Contents

1 Introduction **2**
- 1.1 Notes . . . 2
- 1.2 Computer Languages, Books and Teaching . . . 3
  - 1.2.1 Computer languages . . . 3
  - 1.2.2 Books and Teaching . . . 4
- 1.3 Pedagogy . . . 5
- 1.4 Legal-ish . . . 8

2 First program and printing **9**
- 2.1 Hooking up a simple program . . . 9
  - 2.1.1 Install/configure Unity3D . . . 9
  - 2.1.2 Program set-up . . . 10
  - 2.1.3 Errors in set-up . . . 11
  - 2.1.4 A simple program . . . 12
  - 2.1.5 Running it . . . 14
- 2.2 Statements, `print` . . . 15
- 2.3 Intro to computer math . . . 16
- 2.4 More writing program details . . . 16
  - 2.4.1 White space . . . 17
  - 2.4.2 Style . . . 18
  - 2.4.3 Comments . . . 18
- 2.5 Errors, Syntax errors . . . 20
  - 2.5.1 Compilers . . . 20
  - 2.5.2 making some errors . . . 21
- 2.6 Adding more programs . . . 24
- 2.7 more math . . . 25

3 Variables, computer math **26**
  - 3.0.1 Literals . . . 26
- 3.1 Math with types . . . 27
  - 3.1.1 mixed types . . . 29
- 3.2 Variables, declarations, simple assignment . . . 30
- 3.3 Identifiers . . . 31
  - 3.3.1 Style . . . 31

3.4 Simple assignment, var use . . . 32
3.4.1 More assignment statement rules . . . 33
3.4.2 Initialization, unassigned vars . . . 35
3.4.3 Computer math with variables; types . . . 36
3.4.4 mixed-type assignments . . . 37
3.5 More assign and variable rules and examples: . . . 38
3.5.1 examples . . . 39
3.6 Declare/assign shortcuts . . . 41
3.7 Semi-useful variable and assign examples . . . 42
3.8 More error messages . . . 44
3.8.1 Run-time errors . . . 47
3.9 Doubles vs. floats . . . 47

4 Input, more output, testing 49
4.1 Input . . . 49
4.2 Using Update to make more interesting programs . . . 51
4.3 declare = shortcut and Inspector vars . . . 53
4.4 Replacing constants with variables . . . 54
4.5 Math shortcuts . . . 55
4.6 Fun 3D stuff . . . 56
4.7 more cute cube tricks . . . 58
4.7.1 Colors . . . 59
4.7.2 Scale . . . 60
4.8 Errors, funny stuff . . . 61

5 Using a 3D environment 63
5.1 Making a small room . . . 63
5.1.1 Intro . . . 63
5.1.2 A floor . . . 63
5.1.3 walls . . . 64
5.2 Color, Textures . . . 66
5.3 Adjusting the Camera . . . 67

6 If statements 69
6.1 Comparison Operators . . . 70
6.2 Some semi-useful ifs . . . 72
6.3 Compare precedence . . . 75
6.4 else . . . 75
6.5 And, Or . . . 78
6.5.1 Useful and/or's . . . 81
6.6 Nested ifs . . . 82
6.7 Common errors . . . 85

7 more if tricks **88**
7.1 falling through . . . 88
7.1.1 Cascading range-slicing ifs . . . 89
7.1.2 Do only one of these, none of the above . . . 91
7.2 Swapping . . . 93
7.3 Incremental calculations . . . 93
7.3.1 guessing game example . . . 94
7.3.2 Gold mine problem . . . 95
7.4 Tricks with modulo . . . 97
7.5 Mixed and/or . . . 100
7.5.1 not, DeMorgan's law . . . 101
7.6 Switch Statements . . . 102

8 Unity `if` examples **104**
8.1 bouncing . . . 106
8.1.1 Speed increase . . . 107
8.2 Counting . . . 110
8.3 Delay counter . . . 111
8.4 State variables . . . 114

9 Scope and Namespaces **118**
9.1 Scope . . . 118
9.1.1 More global rules . . . 119
9.1.2 block scope . . . 121
9.1.3 Common errors . . . 121
9.2 Namespaces . . . 122
9.2.1 Notes/rules . . . 123

10 More ifs and randomness **125**
10.1 Random rolls . . . 125
10.2 Random Positions . . . 127
10.2.1 Random size . . . 129
10.2.2 Color . . . 129
10.3 More things to randomize . . . 130
10.4 Coin flips, percents, random tables . . . 131

11 Functions, part I **135**
11.1 Intro Function examples . . . 135
11.2 Function rules . . . 137
11.3 More function examples . . . 139
11.3.1 Longer resetToLeft example . . . 141
11.4 functions with more math . . . 143
11.5 Common Errors . . . 144
11.6 Style . . . 145
11.7 Chains of functions . . . 145
11.7.1 Multiple local scopes . . . 147

12 Functions, part II - parameters **149**
12.1 Parameters 149
12.2 Rules 153
12.3 More rule-showing examples 154
12.4 Fun errors 155
12.5 Multiple Inputs 156
12.5.1 More errors 158

13 Casting and `char` **159**
13.1 casting 159
13.1.1 `(int)` from float 160
13.1.2 `(float)` from int 161
13.1.3 `string` casts, conversions 162
13.2 `char` type 162
13.2.1 `char` examples 163
13.2.2 Casting `chars` 164

14 Functions, part III - return **166**
14.1 Introduction 166
14.2 return values 166
14.3 Using value-returning functions 168
14.4 More math functions 170
14.4.1 Extra return rules 172
14.4.2 Optional empty return 173
14.4.3 `return` can use math 174
14.4.4 Errors 175

15 Constants and enums **177**
15.1 Constants 177
15.2 enumerated types 178
15.2.1 value returning `?:` ifs 180
15.3 More enumerated types 182

16 Overloading and default parms **184**
16.1 Introduction 184
16.2 Default parameters 184
16.3 overloading 186
16.4 Style 188

17 booleans **189**
17.1 Bool rules 189
17.1.1 bool variables in `if` statements 191
17.2 Common bool tricks 192
17.2.1 Pre-computing if tests 192
17.2.2 bool return values 195
17.2.3 Bools for function settings 196

17.2.4 flags 197
17.2.5 Incremental bool calculations 198
17.3 Testing tricks 199

18 Function examples **201**
18.1 Introduction 201
18.2 Change me type functions 203
18.3 Dice functions 203
18.4 More style; types of functions 204
18.5 More Unity Movement examples 206
18.6 Reading keys 210
18.7 Making a real function library 214

19 Structs **216**
19.1 Cow struct example 216
19.1.1 More examples 219
19.2 Common errors 223
19.3 rules 224
19.3.1 namespace vs. struct dots 225
19.4 Builtin structs: Color, Vector3 226
19.4.1 Color 227
19.5 Constructors, struct literals 228
19.6 Nested structs 229
19.7 Structs as function inputs 231
19.8 Structs as return values 232
19.9 Uninitialized structs 234

20 Struct and Unity examples **235**
20.1 More Vector3, Color code 235
20.2 More built-in functions 238
20.2.1 Distance 238
20.2.2 MoveTowards 239
20.2.3 More movement examples 240
20.2.4 Color namespace 241
20.3 Transform struct 242
20.3.1 Special errors; pull-out trick 243
20.4 Rigidbodies 243
20.4.1 Simple physics set-up 244
20.4.2 Playing with velocity 245
20.4.3 Collision callback 246
20.4.4 The Collision struct 247
20.4.5 Misc event notes 249

21 Float problems **251**
21.1 The Problem 251
21.2 Solutions 252
21.3 < vs <= in float compares 254

22 Loops **256**
22.1 Special infinite loop warning 257
22.2 Number-sequence loops 257
22.3 While loop rules 261
22.4 Do it 10 times loops 262
22.5 Loops with floats 264
22.6 Go until done loops 265
22.7 More oddball loops 269
22.8 Infinite/broken loops 270
22.9 Move and count loops 272
22.10 Digit tricks 272
22.11 Fencepost errors 275
22.12 for loops 275
22.13 Count plus formula loops 277
22.14 do-while loop 278

23 Index Loops **279**
23.1 Indexes 279
23.1.1 Length of a string 282
23.2 Variables as indexes 282
23.3 String Loop examples 283
23.3.1 Index inputs/outputs 285
23.4 Odd string loops 288

24 Pointers **293**
24.1 First walk-through 293
24.2 Terms and Theory 295
24.2.1 Pointers 295
24.2.2 `new` vs. Declare, Heap vs. Stack 296
24.2.3 Reference types 299
24.3 Common pointer/class use 299
24.4 Functions and pointers 301
24.4.1 Pointer inputs 301
24.4.2 Pointer outputs 302
24.5 null 305
24.6 Errors 307
24.7 Pointer compare (`==`) 308
24.8 Garbage Collection 308
24.9 Why struct `new`'s? 310
24.10 Old-style pointers 310

25 Pointers in Unity **312**
25.0.1 Pointers to existing Cubes 312
25.1 Instantiate 315
25.1.1 Instantiate as a copy 315
25.1.2 Proper Unity use of Instantiate 317
25.1.3 Pointers for "caching" 317
25.2 New'ing Inspector variables 318

26 Pointer examples **320**
26.1 Text change 320
26.2 More time, and APIs 322
26.3 Three platforms 325
26.4 More prefab use 327
26.5 Multiple scripts 330

27 Using indexed lists **331**
27.1 Intro 331
27.2 Creating Lists 332
27.3 More playing with indexes 334
27.4 List loop examples 335
27.4.1 list-Initializing loops 335
27.4.2 Printing 336
27.4.3 List Searching 337
27.5 Random item 339
27.5.1 A list as a sequence 339
27.5.2 Variable variables 342
27.5.3 Moving parts of a list 343
27.6 Two list tricks 344
27.7 Size 0 and 1 lists 345
27.8 Functions with lists 345
27.8.1 Lists as inputs 345
27.8.2 Returning lists 347
27.9 errors 348
27.10 Assign shortcut 349
27.11 List variables as pointers 349

28 Nested loops **351**
28.1 As a grid 352
28.2 Pattern in pattern 353
28.3 Wedge exercises 355
28.4 break, continue 358
28.5 List nested loops 362
28.5.1 Sorting 363

29 Struct in struct **365**
29.1 Lists of structs 365
29.2 Structs with Lists in them 367
29.3 Lists of strings tricks 368
29.4 2D lists 370
29.5 Larger structures 372
29.6 Giant game board example 374
29.7 List of indexes 376
29.8 Internal list pointers 377

30 Member functions **381**
30.1 Example and motivation 381
30.2 More simple examples 383
30.3 Calling your own member functions 387
30.4 Built-in member functions 388
30.4.1 Unity member functions 388
30.4.2 String member functions 388
30.4.3 List member functions 389
30.5 `this` 390
30.6 Constructors 390
30.7 Scope 393
30.8 Style: member vs. non-member 393
30.9 Special struct in script rule 394

31 Access modifiers **395**
31.1 How `private` variables work 395
31.2 Classes as new types 396
31.3 Interface/Implementation idea 399
31.3.1 Rewriting classes 402
31.3.2 Accessors 403
31.3.3 Special get/set 405

32 Scripts as Classes **406**
32.1 How scripts really work 406
32.2 Pointers to scripts 407
32.2.1 Direct script pointers 407
32.2.2 GetComponent to find scripts 408
32.2.3 Multi-script ball-dropping game 409
32.2.4 Hand-running fake updates 413

33 Nested classes, static **416**
33.1 Nested `enum`'s, classes and `public` 416
33.2 Static / namespaces 418
33.2.1 static functions 418
33.2.2 Fake classes for namespaces 419
33.2.3 static variables 420

33.2.4 File of classes 421

34 Reference parms **422**
34.1 `ref` rules 422
34.2 More `ref` examples 423
34.3 Output parameters 425
34.3.1 `out` shortcut 425
34.4 mixing real return and `ref` parms 426
34.5 Pointers and References 427
34.6 Raycast example 428
34.7 Reference vs. reference 430

35 Arrays **432**
35.1 Basic array rules 432
35.2 Arrays and functions 433
35.2.1 Array work-arounds 434
35.3 Max size / current size trick 435
35.4 creation shortcuts 436
35.5 List / array conversion 436
35.6 OverlapSphere Unity example 437
35.7 2D arrays 437
35.7.1 Ragged `[][]` arrays 438
35.7.2 Real 2D `[,]` arrays 438

36 Efficiency **439**
36.1 Most speed-ups don't work 439
36.2 Premature Optimization 441
36.3 Not all code sections are equal 443
36.4 Is it already fast enough? 444
36.5 Will it cause more errors? 444
36.6 Things that work 444

37 Debugging **446**
37.1 Test as you go 446
37.2 How/what to test 447
37.3 Tracking down bugs 447

Chapter 1

Introduction

This is about basic computer programming. It starts from nothing and covers a little more than one semester of a Programming-I class. It uses C# in the Unity3D game engine, but it isn't about them any more than it needs to be.

A game engine seems like a funny choice. It's more difficult to get started than the traditional "scroll lines against a black screen". But the 3D environment lets us write some nice examples with moving blocks and colors. It also provides a nice way to watch our variables as we run.

C# isn't an especially good first language, but it's fine. The basics are the same as every other language – much of it is exactly the same. And it's not the worst thing to learn a language in common use.

1.1 Notes

Since the programs need to run, there will be certain amount of nit-picky `C#` rules. That's fine – every language has them, so we may as well get a feel now. I'll try to be clear about the parts everyone uses vs. the odd bits that everyone does a little differently.

About half of the examples are moving blocks around and other game tricks. But only because they make good examples. This absolutely isn't going over 3D game skills. It even leaves out some easy 3D tricks which aren't good for examples.

Whenever you teach anything, you lie a lot. In first grade math you learn you can't have 3 minus 5 since you don't know negative numbers yet. Then second grade says you can't divide 9 by 4, since you haven't learned fractions yet.

I'm going to do the same thing. Instead of telling you the Ultimate Rule for something, I'll start with a good-enough-for-now rule, then fill it out later. At

the end that seems to work better.

Many of the examples that do something useful are the same way. You'd probably do that exact thing in a different way. But you won't know that yet, and the exact trick is still good, except as part of something larger and more complicated that would make a terrible example.

Examples that do nothing useful are also good – you have to pay close attention to every step of the rules. I'll try to label these as teaching/useless/silly so you don't waste time trying to figure out what they do, which is nothing.

1.2 Computer Languages, Books and Teaching

This is just for fun, if you wonder about all the various books on programming. These are just my opinions.

1.2.1 Computer languages

This whole thing is based on all computer programming sharing the same ideas, no matter which language you use. Learn the ideas and you easily figure out any language. Here's some background why I claim that:

Computers don't understand *any* programming languages. They only run a set of very simple instructions. The way programming languages work is by translating your code into real computer instructions.

We can write programs directly in computer code (called assembly code.) It's a huge pain, and takes forever, and we invented programming languages so we didn't have to. But all programming languages are shortcuts for the same simple computer commands.

Another thing is that programming languages get revised, and borrow from each other. We started writing them in the 1950's, tried a lot of things, and over the decades figured out the best concepts, rules and even the symbols that seemed the best.

As new things were invented, not that many, we did the same thing. Older languages often added them once all the kinks were worked out. Modern programming languages were created by taking things from the common pool of "programming ideas."

This makes it sound like there should be only one programming language. There are lots of them because there are lots of trade-offs:

Languages good at quickly writing small programs aren't good for huge ones. Some nice features make the program run slower; sometimes a lot slower. Adding too many shortcuts and features makes it hard to read. Some languages

can run with very little memory by cutting to a bare minimum of features. Some great features are error-prone for new users – "safe" languages cut those.

Then, some people just like different sets of optional features, or like commands spelled a certain way.

1.2.2 Books and Teaching

There are maybe three kinds of Computer programming books: basic computer programming, language reference manuals, and Project books.

Reference manuals are great once you know how to code. Obviously they don't tell you anything about how to program, and only have examples for the weird special case stuff. The Unity3D online Scripting Reference is like this. It's incomprehensible if you aren't a programmer who knows game stuff, too. But if you are, it's just fine.

The problem is you can't always tell something is a reference manual. The microsoft online C# docs have some introductory examples – just enough to fool you, but not nearly enough to teach you programming. The site's main purpose is to be a reference manual.

Many books about a particular language are written for programmers wanting to learn it. They have a teaching style, but only about new features in that one particular language. The book will mention all of the basics, but leave out what every programmer already knows. These books are just great, but can be especially frustrating if you don't realize they're not written for beginners.

Project books are things like making a single 3D game, learning the programming as you go. Those are fun, give you a nice overview, and might inspire you to learn more. They seem better since you finish with a free game. But I don't think they work.

A problem is there's too much to cover. For a game you need to learn the engine, 3D models, art, sound, particle systems . . . and coding. Any of those could fill a book. Another problem is the best examples for each coding topic come from all over. And a real game is too complicated - you get lost in the explanation of all the things to make even a simple program to move the player.

If you think you only learn-by-doing, don't sell yourself short. Everyone learns with some practice, some reading, and some explanation. In a textbook it's fine to jump straight to the examples, try to run a few things, and then go back to read the explanation.

Books about learning basic programming have their own difficulties – you need to write what buyers think they want. When someone goes to a bookstore to learn C#, they might see "Learning computer programming", "Learning C#" and "Learning C# with 3.0 features". The first one teaches you how to use loops. The second shows the 4 different kinds of C# loops. The third also

shows the new special-case loops over arrays. Each one is more of a reference manual, and less about how to write programs.

Everyone knows that the manual for the XK-20 nail-gun won't let you make a birdhouse. You need to know basic carpentry for where the nails go and how many. But not many people know that about programming.

Object Oriented Programming books have a similar problem. "Learning Object Oriented C#" sounds pretty good. But OOP is an advanced trick – a way to organize large programs. In an intro book all it does is make the examples longer and harder to read.

For books used by schools, you'd think it would be different, but there's also pressure to teach OOP and the latest hot language. It's also fine for it to be more of a reference manual – the part about how to program will be done in class. And the book selection process for an intro class can be a bit messy. Often you're using whatever book it had before, and only teaching it for a year.

Upper-level Com Sci textbooks tend to be pretty good. The instructor knows the area well, cares about it, and is very interested in choosing the right book.

1.3 Pedagogy

This is a standard part of programming books where you explain to other teachers why you arranged things the way you did.

When I was teaching I gradually realized that the most important thing was convincing students that programming is understandable. They really, really want to memorize things. My job was to convince them they could understand the basic idea of `if`'s, solve lots of problems with them, and the rest was just details.

All I had to do was ruthlessly strip away everything else. Not quite, but I have to keep reminding myself that I'm not writing a manual and they're not going to be reading production code immediately after taking my class. We don't need special cases, alternate syntaxes, or shortcuts.

I learned to avoid lists. Seeing all of the Reserved Words is fun, but I'm trying to show them they *don't* need to memorize things. The same with the traditional 2-page list of every data type. I love `long long int`'s as much as the next guy – but why do we show them a big list of things we're not going to use?

I like to put some of the optional stuff in its own chapter. Not so much to cover it, but to show how a working language fleshes things out. But even then, keep it short. One line with a foreach loop, mentioning how it's a shortcut for an index loop, is plenty. That's enough for an interested student to look it up.

A guideline for me is that pages and space equals importance. Switch statements are individually so long and have so many rules that they take 3 pages. If you spent 2 pages on boolean logic, switch statements automatically seem more important. If most examples also include one bonus rule, the net effect is that

there sure are a lot of rules to remember.

I've noticed that students confuse actual rules with style rules and defensive-coding rules. We think we're giving them 2 or 3 rules in 3 different categories, but it blurs into 8 general purpose rules. Pretty soon they think wrong camel-casing caused a logic error. The simplest way to fix this is to only give real rules.

I think style can be shown by example. The later, instead of "the body of an `if` statement is indented" you have the non-rule rule "the body didn't need to be indented, but it looks nice that way."

Showing proper programming is just too horrendously distracting. This is where every example checks for valid input and uses try-catch blocks.

The thing is, I understand the impulse to hammer away at style from the start. I started college during the structured programming movement. General contempt for style, or readable programs at all, was a glaring problem. You didn't tell true genius programmers how to do their jobs. Job security meant writing code that only you could figure out. And so on. I took the class that was a reaction to that, where 25% of the homework grade was for style.

But that old culture is mostly gone. Python's geek culture is proud of "pythonic" code, which is all about style. Modern languages are more readable, editors push style on you. If anything, style is now probably over-emphasized.

My favorite tricks:

- int, double, string are the best group of types. 3 is the perfect number. String is exotic; int vs. double shows how a type is what we say it is. string/int/double casting is enough to get the idea, and sort of interesting.
- Use strings to teach indexing. People are so much better at looking through letters 0 through 4 of `"zebra"` than the list {`5,8,4,12,7`}. Use strings to teach everything you can about index loops and off-ends. Then move onto formal arrays.

  `C++` is a little better for this, since `w[i]='x'` is legal.
- Arrays of structs, structs with array fields, arrays of nested structs and so on are great. They're good for going left-to-right through each dot and seeing what the current type allows. They're an opportunity for all sorts of interesting nested loops. They're an excuse to write more small classes with a member function or two. Then they're just a nice example of making a data structure.

Covering functions early is great since all of your examples afterwards can be functions. If you want to do something with a name and a number you normally have to declare them and wave your hands about them somehow getting filled. You write things like "now suppose w is `cow` and n is 8".

With functions it's shorter and more clear to write `void story(string name, int count)` at the top. That literally means that they somehow get

filled. `story("cow",8)` is the best way to say "pretend they are ...". Even better, once you have returns you can stop saying "and now `n` is the final result." Just write `return n;`.

Functions are the best way to set up and bracket code snippets.

I'd normally have functions as the first thing. In a class I can talk students through how function don't accomplish anything, but they'll be very useful and will be on the test. In a book it seems important quickly do something useful, so I moved `if`'s as the first thing.

Loops are so far back because of Unity's Update loop. It lets us write loop-like things and use loop thinking very early on.

Reference Types make teaching pointers and the heap a big pain. Here's how I covered pointers in `C++`: first pointers to normal vars, like `int* p1=&n;`. Having everything explicit is really an advantage: `int*` holds an address and `&n` creates an address. There aren't any secret steps, and at this point students understand about types having to match.

Then, once we've used pointers for a bit, we can move on to `new` and the heap. This is the best way to learn pointers, which includes knowing how to use Reference Types.

Learning Reference types first is a mess. In C++ terms you're learning structs, pointers, and the heap, all at once. The way we first hand-wave `Cat c=new Cat();` just makes it worst. The best way I could think of was to avoid classes at first. Cover structs. That gets practice with member variables and dots. Unity examples use lots of `Vector3` and `Color` structs, so that helps. Then, much later, cover pointers and `new` together when you introduce classes.

All of the old textbooks had triangle-printing nested loops. I thought they were self-indulgent and show-offy. Then I realized they were great nested-loop examples, with indexing practice and immediate visual feedback. if you're off-by-one, the picture is off-by-one. In that section, the checkerboard example is original. The rest are from every "Learn BASIC" book written in the 80's.

It's a shame Unity's debug window breaks up the output, but you can still see the star patterns.

I think I inherited my first `C++` class using `Vector` instead of arrays. Or maybe I was smart enough to do it myself. Either way, an array container class is the way to go. Easier to use but still all the fun of indexing. Much later, cover arrays. Students now know indexing so you're just covering the fixed-size rule and the work-arounds for it. Here I've got the the array-wrapper `List` class first, with arrays much later.

Sadly, I don't think the Unity3D API uses a single List – it's all arrays. That makes sense for a game engine. But it's not helping building a case that you should primarily use Lists over arrays.

1.4 Legal-ish

Copyright Owen Reynolds 2016, 2018. Permission is given to print, copy or otherwise distribute however you think might be helpful to you.

Unity3D is a trademark of Unity Technologies, used here without permission.

Chapter 2

First program and printing

This section is mostly about mechanical stuff – things we need to get out of the way before we can get to the programming part. The first part is setting up Unity3D to where we can run a program. The next is the basic rules and parts of a program.

Then the rest is a little about programming – one command, and rules for computer math.

Many of the rules are obvious, but it's nice to have them all in one place, written down. So at least skim them if you already know most.

2.1 Hooking up a simple program

I'm going to describe the quickest way to set things up so Unity3D can run a program. If you've got it set up, or want to set it up in some other way, that's fine. Not much later depends on you having things this exact way.

2.1.1 Install/configure Unity3D

One of the nice things about using Unity3D is that there's been a lot written about getting started. I'm going to list the steps, but not go into detail about the parts that are easy to find a video of.

Obviously, download and run Unity3D. It may direct you to something called UnityHub, or may allow a direct download. The exact version won't matter, but the latest one is fine. There will options to include various extra Assets – we won't need any of them, but they won't hurt. You can always get them later.

Then we'll need the code editor. One used to come with Unity, but no longer. If you already have MS-Visual Studio, you can use that. If not, Unity prefers something called Visual Studio Code. It's not really Visual Studio – it's

an open source C# editor called Monodevelope which microsoft modified and put on their site. Getting it all is a pain.

After installing, you should check it's connected to Unity. With Unity open go to `Unity->Preferences->External tools` and see it's set to Code or Visual Studio.

If you haven't used Unity before, you may want to arrange the panels and play around. In the upper-right corner, where it says `Layout`, my favorite is "Tall."

We'll mostly be typing and running programs, but we'll need to know a little about the areas in the Unity screen:

- The largest part is the Scene/Game area. It's a view of the 3D world. We're not going to need it for a while.
- The Hierarchy panel lists items in the game world. To be nice, Unity premade *Main Camera* and *Directional Light* for us. Those are fine for now.
- The Project panel holds everything else. Our programs, and pictures and 3D models (if we were really making a game.)
  It's set to two-column mode now, which I think looks terrible. In the upper right, next to the "lock" picture, there's a little icon with three bars and an arrow. Clicking on that lets you switch to 1 column mode.
- The Inspector panel holds details about the currently selected object, and gives us a way to make changes to it. For fun try selecting *Main Camera* and *Directional Light* from Hierarchy, and watch the Inspector panel change.
- The Console is a separate pop-up window, currently closed. Anything we print goes there, as well as error messages. The very bottom line in the Unity window shows the last line from the Console. If it's blank, the Console has nothing in it. The simplest way to open the Console window is double-clicking that line.

2.1.2 Program set-up

To get a program we can run, we need to do three things: make a new program file, open it so we can type stuff, and convince the Unity3D system to run it.

To make the file, find the top bar of the **Project** panel and click `Create->C# script` (script means program. It's supposed to be less scary sounding.) A new item should appear below and ask you to name it. Name it anything with no spaces – `testA` is fine.

As soon as you name it, you'll see some code lines in the **Inspector** side-panel. That's just a helpful preview – you can't type anything there.

To bring up the editor, double-click your `testA`. It may take a while, but eventually a new window with your editor will appear, with your program in it. That's where we're going to do all of our typing. We can hide it, for now.

The last step is convincing Unity3D to run it. This is a little funny, so I want to explain the way game engines think:

The system takes care of the 3D world, remembering where everything is, and showing it. If we want a block to move back-and-forth, it expects us to write a back-and-forth program and put it on the block. If we want the light to flicker on and off, we write another mini-program and put it on the light.

That seems funny, writing more than one program, but it's pretty typical. For example, a web page has a mini-program for each button.

The way the system thinks, if you write a program and *don't* put it on anything, you don't want to run it yet. Even a program that only prints to the Console still has to be on something to run.

Right now, we have two things in the game world we can put our program on. *Directional Light* has more room, so we may as well use that. Select *Directional Light* (the one in Hierarchy.) You should see its details in the Inspector. You could safely pop shut the little area with details about how bright it is.

Next drag `testA` from **Project** onto the blank space at the bottom of *DirectionalLight* in the Inspector. If it works, the original `testA` will still be in Project, and *Directional Light* will have a new `testA` section.

2.1.3 Errors in set-up

The system won't prevent you from having more than one running copy of `testA`. If you accidentally drag it twice onto the Light, it makes 2 copies. Or you may accidentally drag another copy onto the Camera.

The system will run every copy of your program at the same time. It won't necessarily cause a problem, but it will print the output twice, possibly in mixed-up order, making it difficult to see if your program worked.

If that happens, you can scroll through Hierarchy and look at the Inspector for each. When you find a duplicate, click the gear on the right side and select `RemoveComponent`.

The obvious other problem is you can have 0 copies running. Make sure it's on the Light or the Camera.

When you try to drag it, you might get an error pop-up "Can't add script." That's probably due to a late name change. When you make it, you can keep the name or change it. But if you rename it later the system gets confused (we'll see why, later). For now, the easiest way to fix "Can't add script, names don't

match" is to delete the script (right click, Delete from the Project panel) and start over.

2.1.4 A simple program

A normal program starts blank, and you type everything. But it's also common for a system to give you some starting code, which Unity does. If we bring back the code editor window we should see these program lines. Don't worry too much about what they mean yet:

```
using UnityEngine;
using System.Collections;

public class testA : MonoBehaviour {

  // Use this for initialization
  void Start () {

  }

  // Update is called once per frame
  void Update () {

  }
}
```

Eventually every symbol will make sense, but for now, we're going to ignore or delete most of it. But it still helps to have a really rough overview:

- The top two lines with `using` just have to be there. There are rules for what they mean, and sometimes you change them. But for now its easiest to think of them as magic.
- Blank lines don't matter. The computer skips them. They're only to space things out so they look nicer. Feel free to add or remove all the blank lines you want.
- The long line `public class testA: MonoBehaviour {` also has to be there, just like that. See how it has `testA` in it? That's the name we picked (it's a 1-time change Unity makes when we name the script for the first time.)
- That same line ends with an open funny-paren. The official name is a curly-brace. Like normal English or math, they always come in pairs. This open { matches the close-} on the very last line. From line 4 on down, the program is one big block named `testA`.
- The two lines starting with `//` are little notes for humans, called **comments**. The computer ignores them. We can delete them.

- There are two similar chunks named `Start` and `Update`. Notice how they're different from the `testA` line, but the same as each other – `void Start() {`. The normal `()` parens just go there and don't mean anything yet. They each have an open/close curly-brace pair.

- We've seen how the {'s and }'s match. But when you see those two close curly-braces together at the end, it looks funny. Don't let them confuse you and don't delete one of them. It's like how $12 - (3 - (2 + 2))$ ends with two close-parens, since both open-parens just happen to end there.

To make our first program easier to read, let's delete down to the bare minimum. We don't need the `Update` chunk or the \\ comment lines. We just need this:

```
using UnityEngine;
using System.Collections;

public class testA : MonoBehaviour {

    void Start () {

    }
}
```

Unity's special rule is that it runs the chunk named `Start`. The open and close {}'s mark off the contents, which is currently nothing. So this is the smallest program which runs and does nothing.

We can finally start writing the real program. I'll have it print three things:

```
using UnityEngine;
using System.Collections;

public class testA : MonoBehaviour {

    void Start () {
        print( "howdy" );
        print( 4+3*10 );
        print( "2+5" );
    }

}
```

Some details about typing this and the symbols (which will be super-obvious once you get used to them):

- Starting spaces don't matter. The computer will probably auto-intent, but any number of spaces or TABs are fine. I thought it looked nice this way.

- When you type the p in `print`, the editor will make a little pop-up of all the legal p-words, trying to guess what you want. You can keep typing, or can use the arrows and select with Enter. Or can press the esc key to get rid of the pop-up. Any way is fine, as long as it says `print` when you're done.

  If you aren't getting the pop-ups, it means the stupid intellisense is broken. See below.

- It's case-sensitive. It must be all lower-case `print`. Not capital-P Print. That seems funny, since the `S` in `Start` has to be upper-case. But that's the rule.

- The `()`'s after print are regular parenthesis.

- The quotes are both the same double-quote key – the one above the comma. There aren't any special start or end quotes.

- Spaces between things don't matter. I used one space in a few places, but you don't need to. Extra spaces at the end of the line don't matter.

- The symbol at the end of the new lines is a semicolon (`;`).

- The symbol between 3 and 10 really is a star (on the keyboard, above the 8.)

Finally, there's one more install problem that may need solving. The editor should highlight the special Unity commands – Monobehaviour, UnityEngine, print. If those are white or don't pop up, or hovering doesn't give a tool tip, it means "intellisense" is broken. The code editor may be asking you to install add-ons. Add anything with Unity in the name. Or search for "Unity Code Intellisense not working".

Broken pop-ups won't stop the code from running – Unity can read it even if the code editor can't. But it will get annoying after a while.

2.1.5 Running it

To run the program, save it, click back to the Unity3D window, and click the **Play** button on top. We should see `2+5` on the bottom of the window. After a little bit, press the `Stop` button.

The program printed three lines to the Console window. If you remember, the bottom of the screen helpfully shows us the last line. Double-clicking brings up the Console with lines: `howdy`, `34` and `2+5`, and some junk between them which we can ignore.

The Console window will always pop-up when you double-click. I usually close it when I'm done reading, since it's so easy to reopen.

Before looking at what those three lines mean, some more notes about the system:

- If you run it a few times, the lines might pile up in the Console. Nothing wrong with that, but it can be hard to see the new stuff. On top there's a "Clear on Play" option. It's probably turned on now (it should be more white than grey.) If it's not, click it to turn it on.

- The code editor has lots of buttons on top, including `Run`. We don't need to use any of those. They're for stand-alone programs. Using them will run the code outside of Unity, which won't make any sense.

- It seems funny to need a `Stop` button, since the program prints 3 things and quits. But the system is set up for programs that run over time, like games. It thinks our program might do something else, eventually, so it keeps running.

  You have to press `Stop` before running it again, so I think it's easier to get in the habit of always pressing `Stop` before doing anything else. But it won't hurt if you leave it running.

- When you added those 3 new print lines and went back to Unity, you probably noticed a small delay. Unity is examining your new program. If there are any errors this is when you'll see them. They'll be a little red ball at the bottom of the main window. We'll look at errors later.

  If you press `Play` when you have red errors, it gives you a big message about not being able to run. Nothing else bad happens (but you still have to fix the errors.)

2.2 Statements, `print`

Computer commands are called **statements**. Not a big deal, but error messages use the word, so you'll be seeing it. Statements run one at a time, top to bottom, left to right. Statements end with a semi-colon, which acts like a period in a sentence.

The part of the program we care about is three statements:

```
print( "howdy" );
print( 4+3*10 );
print( "2+5" );
```

Notice how even the last one needs that semi-colon. It's like how even the last sentence in a book needs a period.

`print` is meant for testing, which is why it prints to the Console, with the extra lines.

The ()-parens are part of the print command, and always have to be there. It prints whatever is inside them. In other words, the print statement looks like this:

```
print( stuffToPrintGoesHere ) ;
```

The contents of those prints are just to get us started. You might notice there are two types:

- Things in double-quotes, which are printed exactly. These are called strings (more later.)
- Math, which is calculated before printing.

The last line `"2+5"` looks like math, but it's in quotes, so uses the print-exactly rule. If you took the quotes off, saved and pressed Play, it would print 7.

2.3 Intro to computer math

Computer math works as much as possible the same as real math. The oddest thing is that the star symbol (`*`) is always used for multiply. In regular math, you can write $3x$ or $4(3+5)$. In a computer, you'd have `3*x` or `4*(3+5)`.

Precedence works the same as normal math: multiply and divide before plus and minus. The second line `4+3*10`, is 34 because 3 times 10 goes first.
Also like regular math, parens go first. `print((3+4)*10);` gives 70.

Required parens and math parens can be a little confusing. In `print((3+4)*10);` the outer parens are required parts of the print. The inner ones, around 3+4, are math parens that we added to change the order.
You can add all the math parens you like, even ones you don't need. For examples `print( 3+(4*5) );` or `print((2+5));` or even `print(((1))+(2));`

Here are a few more, with what they print:

```
print( 5 ); // 5 -- counts as math
print( "5" ); // 5 -- works differently, but looks the same

print( 12/2 ); // 6 -- same division symbol as normal
print( 1+1+1+1+1 ); // 5 -- can be as long as you want
print( "6/3+1"); // 6/3+1 -- not math, because of the quotes

print( 3*2+2*2 ); // 10 -- both multiplies go first, then 6+4
```

2.4 More writing program details

The part about using Unity is now mostly out of the way, and we can focus on writing a program in the editor for a while. But there are still rules about just program writing in general.
This isn't too tricky, but I want to be sure you see the terms and the general idea.

2.4.1 White space

Together, all of the indents, blank lines and extra spaces are known as White Space. The short rule is that none of it matters. Statements work like sentences, with semicolons instead of periods. The long rules are mostly obvious, but here they are:

- Blank lines never matter. Can add as many as you like, or can delete them.
- Indentations (spaces in front of lines) never matter. Can have any amount of spaces or TABs, or both, in front of a line.
- Can add extra spaces and TABs in between parts. For example

```
void   Start (       )   {
  print  (  "howdy"      )    ;
```

- Can remove spaces and smash together things if the computer can tell them apart. You can't smash two words together. For example `voidStart` – the computer will think it's one big word and cause an error. But almost all symbols are safe to smush together.
- Line breaks never matter. You can put several statements on a line, break one across lines, or both. This is another legal way to rewrite the program:

```
void Start(){print("A");print("B");
print(
"C")
;}
```

  It's the same as reading sentences, where the semi-colon is the period. Sometimes you have two on a line, sometimes it goes into the next line.
- You can't hyphenate words like `Sta-rt` across two lines. You also can't break a string (something in quotes) over two lines.
- There's no maximum line length.

Spacing makes absolutely no difference to the program. The computer actually pre-scans your program, making a version with extra spaces removed. Feel free to use or not use white space however it looks nice.

Some more fun examples:

`print( 6 - 3-2 );` is 1. It looks a little like 3-2 goes first, but spaces don't matter, and math goes left to right. `print( 1 * 2+3 * 4);` is really 2 plus 12. The confusing spaces don't matter.

`print(  "cat");` prints cat with no spaces in front. The computer scans them out and sees `print("cat")`. For fun, `print(" cat");` does make a space (more on that later.)

2.4.2 Style

The official word for things you do a certain way, even though the computer doesn't care, is **Style**. The idea is if you arrange your program to be pretty, it's much easier to read. And if you know the normal way people arrange programs, its easier to read other people's code.

A common style rule is how to indent. The usual rule is: everything inside a "chunk" is indented a little extra. That way you can tell what's inside what. Usually the final curly-brace lines up with the start of the chunk.

Some people like to put the beginning curly on a line by itself, so the { and } line up exactly:

```
void Start()
{
  print("A");
  print("B");
}
```

Moving the { to the end of the `Start` line is just a style choice. It makes the program shorter, and looks not-too-bad.

Since the indent rules are so common, the editor knows them. When you make a new line, it tries to indent the normal amount. But there's nothing special about the spaces it makes. You can change the settings if you want.

Blank lines are often added between major parts. In Unity's starter program, it would look funny if the ending } of `Start` was right next to the beginning of `Update`, so a blank line was added. If you had three prints for your name, then three for something else, a blank line between them looks nice.

We rarely write two statements on a line. But sometimes it looks nice if they're short, and, in your mind, they do one thing . Most people break statements over two lines if they go off the page.

Spaces are added to make it look nice. Some people prefer spaces to make the print-me part stand out: `print( "A" );` others don't: `print("A");`. In math, spacing can help show the order, like `3 + 2*4`.

The funniest thing about style rules is how important they are. They don't really mean anything, so a lot of people don't bother making the program look nice. But as soon as you have to track down some error, you get lost. Don't go nuts, but "proper" indents, spacing and blank lines save more time than they waste.

2.4.3 Comments

Almost all programs have ways to put little notes in them, called **comments** The comment symbol is a double backslash, `//`. It counts as one thing, so can't

have a space between.

The rule for comments is: everything after the comment symbol is skipped, until the end of the line. Obviously, comment lines don't need semi-colons.

Comments can be on lines by themselves, or at the ends of real lines. Some sample comments:

```
// ==========
// Program to help cats
// do their taxes
// ==========

using UnityEngine;
using System.Collections;

// this program is named testA
public class testA : MonoBehaviour {
   // inside testA

    void Start() {
        // inside of Start
        print( "A" ); // prints A

        print         // useless comment in the
        ("B");        // middle of a silly statement

        //print("C");
        print("D");

    } // end of Start
}  // end of testA
```

Comment blocks like the top 4 lines are common enough. The two lines full of ='s look like they might mean something, but they don't. They're just cute borders.

The comment about printing A is useless – never clutter up your program with that junk – but it shows adding a comment at the end of a real line.

Comments like on the split-line printing B are rare, but it shows the rules. The computer really will read those two lines as `print("B");`.

The line printing C is a trick called *commenting out*. The program will not print `C`, since it's a comment. It's a neat trick to temporarily take a line out of the program. It's not a special rule – it's just a regular comment – but we're using it in a sneaky way.

The last two "end of" comments were common before editors could quickly auto-display matching curly-braces. They were helpful time-savers reminding us which close-curly-braces were closing which part.

Most of the comments I'm going to write are "teaching" comments – good for me explaining parts, but wastes of space in a real program.

2.5 Errors, Syntax errors

There are two main categories of errors. The second type is when your program is running along just fine, and then crashes. We won't get any of those for a while. The first type are **syntax errors**, which are a little like spelling a word wrong. They prevent your program from starting.

2.5.1 Compilers

To understand syntax errors, it helps to know how the computer really runs your program:

The program you typed is officially called the **Source Code**. The computer doesn't run it directly. Instead, a **compiler** scans your source code to turn it into something the computer can run, called the **executable**. This scanning first skips all the comments and white spaces, then matches all the {}'s to find the chunks, then checks syntax of all the statements (proper semicolons, spelling, proper math, . . .). There's more after that. The goal is to get the smallest, fastest possible program for when we distribute it.

All of this is invisible to us. That little pause when you return to Unity is the compiler running.

If there's even one thing the compiler can't figure out, it can't make the executable, so can't run. For example, you might have ten perfectly good print statements in `Start()`. Then somewhere down below, not even in `Start()`, you have one bad one, misspelled `priint`. The computer will refuse to run anything until you fix that 11th statement.

The compiler lists all syntax errors, with red balls next to them, in the Console. As usual, you only see the last one until you double-click to bring up the window.

Double-clicking an error will bring up the program window and jumps to the line where the computer first noticed it. Reading the error message says what the computer *thinks* is wrong. The messages use good solid computer terms, which should start to make sense as you learn more about programming.

The compiler mostly scans the program from top to bottom, left to right. This means that the first error is almost always the one to check. The ones after it might have been caused by the first one.

We're going to be getting a lot of errors, so we may as well cause a bunch now and take a look at the messages. They don't hurt the computer.

2.5.2 making some errors

Let's make a missing semi-colon error. Take the semi-colon off the end of the first `print`, save and go back to Unity:

```
void Start() {
  print( "howdy" )   // <- missing
  print( 4+3*10 );
  print( "2+5" );
}

// 2 Errors:
testA.cs(5,17): error CS1002: Expecting ‘;’
error CS1525: Unexpected symbol ‘(’, expecting ‘)’, ‘,’, ‘;’, ‘[’, or ‘=’
```

The first error tells us exactly the problem – expecting a semicolon. If we double-click the message, it takes us right to the line.

Just so you know, the error number (CS1002) isn't important, and the `(5,17)` is the line number and how far over.

The second error looks scary, but goes it away when you put the semi-colon back. It's an example of one mistake causing two error messages. If we always try to fix the first error, we'll be fine.

Now, change so the only error is the last `print` missing a semicolon. Save and go back to Unity:

```
void Start() {
  print( "howdy" );
  print( 4+3*10 );
  print( "2+5" )   // <- missing
}

// 1 Error:
unexpected symbol }
```

Even though it's the same error, the message is different. And double-clicking jumps to the wrong line (the one after the error.)

Try removing the final double-quote from Howdy:

```
void Start() {
  print( "howdy );   // <- missing "
  print( 4+3*10 );
  print( "2+5" );
}

// many Errors:
```

```
newline in constant
newline in constant
newline in constant
 ...
Unterminated string literal
Parsing error
```

That's one angry computer! It thinks your program is one giant multi-line quote, with no end. And double-clicking the first error jumps to the wrong line again. It goes to the line after where we made the mistake.

Try misspelling `print`. Try `Print`:

```
void Start() {
  Print( "howdy" );   // print is spelled wrong
}

// one error:
The name Print does not exist in the current context
```

This is the compiler's way of saying "I don't know what that word is." You might think it would try to guess that we meant lower-case-p print, but clearly it doesn't. But it's still a useful message, and takes us to the line.

Try leaving out the { after `Start`.

```
void Start()  // <- missing {
  print( "howdy" );
}

// one error:
Unexpected symbol 'print' in class, struct, or interface member declaration
```

Again, it gets mad at the line after the error. But I think it makes sense why. `void Start()` isn't an error, yet. The open-curly could have been on the next line. The computer is telling us "hey, `print` isn't a curly-brace!"

Try leaving out both ()'s for Start:

```
void Start {  // <- missing ()
  print( "howdy" );
}

// one error:
A get or set accessor expected.
```

That's just a totally confusing message – we definitely do not need a get or a set. A double-click takes us one line below the problem. But that's fine – by now we know to also look at the line above.

Trying adding an extra } after the last line of the entire program. That gives me *Parsing error* with no line number. Flipping the final } to an open-{ says the same thing. Most errors with messed-up {}s do this. The computer matches them all, then, only when it gets to the end, sees you have too many or too few.

Indenting can help spot missing {}'s. Using the little `[+]` hide/show boxes on the left can also help (it uses the {}s as a guide.) Putting the cursor on an open or close will highlight the matching one. It's hard for the computer to always give good messages when {}s are wrong.

Try spelling `Start` wrong – try `Stork`. No error! But when you run it, nothing happens! This is the worst kind of error, since it isn't technically wrong, like misspelling Cat as Car. It turns out you're allowed to name a chunk anything, but only `Start` gets run automatically.

To sum up:

- Look at the first error. Don't be scared by a page of them. The first could be causing all of the rest.
- The real error is often on the line before the one it shows you, or even a few lines before that.
- Read the error message, see if it makes sense, but it might be completely wrong. Sometmes try ignoring the message and just looking at the previous lines.
- The same error can give different messages, depending on where the line is. The error messages are more like clues.
- "unexpected symbol" for the first thing on a line probably means the end of the previous line is bad. Often a missing semi-colon.
- "WORD cannot be found" often means you spelled it wrong. But not always.
- A final "parsing error" or "unexpected EOF" (EOF means end of file) often means there's a missing }, or extra {, which could be anywhere.

You may also see some yellow triangles. Those are officially called Warnings. They won't stop the program from running. Some of them are important, some aren't. We can ignore them for now.

2.6 Adding more programs

In the next chapters, I'll go straight to "now try running this program" or just "try these lines" without saying anything about how to hook it up or play nice with any old programs. There are a few ways to add new programs:

The simplest is to reuse `testA`. Delete the inside of `Start` and write new stuff. Don't worry about losing your work. Pretty much all the programs here are like those worksheets you got in second grade. Once you learn from them, you'll never need them anymore.

Another way is to make a new script (`Create->C# Script`) and also put that one on the Light. An object can have any number of scripts on it, and will run them all. But running two printing scripts will jumble the lines. There are two ways to run just the new one:

The most obvious is to take off the old one: use the Gear symbol on the old `testA` and `Remove Component`. It won't hurt the program, and you can always put it back on.

The other is to disable it. In the Inspector, there's a checkbox to the left of the name. That's the enabled/disabled box. Just be careful, since once you start enabling/disabling things it's easy to say "why isn't this working?!? Oh, right, I disabled it."

For new programs with more major changes, we can make a new **Scene** and start fresh. Click `Edit->New Scene` and you'll get a fresh *Main Camera* and *Directional Light*. Unity runs the current Scene, only.

Unity will ask you to name the old Scene. When you do, it goes in Project as a black&white cube-corner icon. Double-clicking a Scene in Project should switch back to it.

Starting Unity will usually go to the last Scene you were using. But sometimes it will start a fresh one (check the Scene name on the top bar.) If that happens double-click on the correct Scene Icon in Project (or just double-click all the cube-corner icons until you find the one you want.)

For really minor testing and messing around, don't even make a new program. The comment-out trick is great. For example, this runs one line, but we can get back the old ones if we need them:

```
// midway through math-testing:
void Start () {
    //print( "howdy" );
    //print( 4+3*10 );
    //print( "2+5" );
    print(1+3     *2); // <- testing just this
    //print( (1 + 3)*2);
    //print(         "x");
```

```
}
```

2.7 more math

This is just for fun, we'll do more with it later:

Division is a little funny. `print( 14/5 );` prints 2, instead of the correct 2.8. Back in 2nd grade, I learned 14/5 is two, with remainder four. If you use whole numbers, the computer does whole number math, dropping the fraction.

A fun fact, % is the remainder symbol (officially called modulo.) 14%5 is 4 (five goes in twice, with 4 left over.) It's sometimes useful, but also a way to show that computers sometimes care about whole numbers.

We can't write fractions, only decimals. 3-1/2 is just 3 minus 1 divided by 2, which is 3 minus 0 (since 1/2 rounds down to 0.) But 3.5 is fine. Computers can do real math: $4.2 + 0.7/2$ will print 4.55, like it should.

Chapter 3

Variables, computer math

The next thing we want to know is how to use variables. We're going to be able to do things like `x=3; y=5; print( x+y );`. But before we do that, we have to know about **types**. As humans, it makes sense to divide things into words or numbers. The computer is much more strict about it, and has formal rules.

We're going to use three **types** for now: words, whole numbers, and decimal numbers. The official names are `string`, `int` and `float`.

3.0.1 Literals

Constant values in the program, like 3 or `"yam"` are officially called **literals**. Whenever the computer sees a literal, it first decides what **type** it is, based on how it was entered.

`string` literals are what we've seen before. Anything between double-quotes is officially a string. Some more examples of string literals:

```
"duck"  "12"  "12.6"  "3-1"  "Start()"  "}"  ""
```

We knew `"duck"` was a string, from before. We've also seen tricks like how a number or an equation in quotes is a string. To review, as far as the computer cares, `"12"` is no different than `"ab"` – they're both just two-letter words. `"3-1"` is just a three-letter word. The human who typed it clearly put double quotes around it. If they didn't want it to be a string, they wouldn't have done that.

`"Start()"` and `"}"` are also the same tricks as before. They look a little like instructions, but they're in quotes, so are strings.

The last one, `""` is a little special. It's a string with zero things in it. It's like a zero, but for words. It's usually called the *empty string*. You don't need it that often, but you do need it sometimes.

Non-decimal plus/minus numbers are officially called **integers**. The official computer word for them is `int`. Here are some int literals, nothing special:

```
12   4934002   0   -46   +58
```

Some boring rules: There is never a comma for the thousands place, millions place or any other place. One thousand is just 1000. A plus in front for positive ints is optional: 3 and +3 are the same. Most people leave off the starting +.

`float` literals have decimal points, and always have a lower-case f after them. Here are some `float` literals:

```
0.1f  30.1f  -6.001f  0.00000045f  3.0f  -4.0f  0.0f  47000.0f  3f
```

The strangest thing is that they really do all end with an `f`. That seems like a pretty silly rule. There's a reason for it – more on that at the end of the section.

The first four are basic decimal point numbers. Like school math, they can have just the decimal, or before and after the decimal, be negative, or be very small.

The next four are interesting, since they look like integers. This is where the **type** rules matter. 3 is the integer version of three, and 3.0f is the float version of 3. Put another way: if you have the number three, and want to put it in the computer, you have to decide to enter it as an `int` or as a `float`.

The last one, `3f` is just another way to write `3.0f`. Most people write the "point-oh", to make it more obvious, but it's optional.

3.1 Math with types

A way to get a feel for `types` is to see the rules about how math works with them. When the computer sees math, like two things being added together, the first thing it does is look at the **type** of the things being added.

When the computer sees a + symbol with strings, it combines them end-to-end. The official word for that is *concatenate*. Strings don't use any other math symbols. Some examples:

```
print( "a" + "b" ); // ab
print( "b" + "a" ); // ba
print("catcher" + "dog"); // catcherdog
```

Obviously, the order matters for string plus.

Like math, we can use as many plus's in a row as we want:

```
print( "a" + "b" + "c"); // abc
```

It doesn't know or care about what the strings mean. Below, it combines duck and bill to make a fun word, but it doesn't know `thehouse` should have a space. There's a space between "I" and "like" since I put one there, but it still doesn't know that "puppies" should have one:

```
print( "xxx" + "xx" ); // xxxxx

print( "duck" + "bill" ); // duckbill
print( "the" + "house" ); // thehouse
print("I like" + "puppies"); // I likepuppies
```

The rule "if it's in quotes, it's a string" applies here. These are things that look like numbers and equations, but are really strings:

```
print("2" + "7"); // 27
print("10" + "+0"); // 10+0
print( "1+2" + "3"); // 1+23
print( "4" + "5" + "6"); // 456
```

I like those, since they show how "it's a string" is the most important thing for the computer. In `"2"+"7"`, it wasn't tempted even a teeny bit to math-add them and get a 9.

The rule for number math is you get a result of the same type, `int` or `float`. For example `1.5f + 2.5f` gives a nice even 4, but it's float `4.0f` since it came from floats.

Division shows off the same-type rule. `13.0f/5.0f` gives the real answer, `2.6f`. But using integers, `13/5` gives an integer answer of just 2 (it drops the fraction.) More examples:

```
print( 35.0f / 10.0f ); // 3.5
print( 35 / 10 ); // 3

print( 18 / 6 ); // 3 -- if it goes in evenly, we get the normal answer

print( 15 / 20 ); // 0 -- not a special rule. 20 goes into 15 zero times
print( 15.0f/20.0f ); // 0.75 -- back to real math
```

Dropping the fraction seems completely insane. We did that in second grade because we didn't know how to make decimals. We have the rule now because sometimes we want that option. If I have 13 people, I can make 2 5-person basketball teams with 3 people left over. I don't care that technically those extras make 0.6 of a team.

Looking at the division rule another way, if you want to get the "real" answer, divide using floats. If you ever want the "dropped fraction" answer, divide using ints.

The remainder in **int** division is completely gone. It isn't invisibly saved for later. Here are some fun examples showing that:

```
print( 1/2 + 1/2 ); // 0
```

Both parts are integer 1 divided by 2, which is is 0. Then we get 0+0 is 0. It's tricky because it's so simple. The computer doesn't look ahead or try to figure out what you really wanted.

```
print( 2/3 + 3/4 ); // 0
print( 9/10 + 9/10 ); // 0
```

These are both 0+0, for the same reason. Nine-tenths is a lot closer to 1 than to 0, but the rule is to drop the fraction, not to round. `9/10` is still 0.

This next one tries to trick you with white space. Division still goes first, so this is really 1+1:

```
print( 6/ 5+5 /4 ); // 2
```

Then here's an ugly example you wouldn't use for real. Division goes from left to right, so we get `(10/3)` = 3 first, then `3/2` is 1:

```
print( 10/3/2 ); // 1
```

But `10/(3/2)` is 10. The `(3/2)` is first, which makes 1, and `10/1` is 10.

3.1.1 mixed types

The rules so far are what to do with two things of the same type. We need some rules for what happens when you have two things of different types. For example: `"goat" + 7` and `7.0f / 4`.

For the three types we have now, there are two mixed-type rules (the first two are the rules):

- Adding a string to a number (int or float) converts the number to a string. For example, `"goat" + 7` is `"goat7"`.

  floats are printed in a nicer format, since humans will be reading them: the `f` isn't printed, and `.0` is left off. `"a"+4.0f` is `"a4"`

- Mixing an `int` and `float` causes the `int` to be **promoted** to a float, by having a `.0f` temporarily added to the end. For example, `5/4.0f` is `1.25f`.

- It doesn't matter which comes first. `5+"2"` is still a string and an int, so the 5 gets turned into `"5"`. `9/2.0f` and `9.0f/2` are both 4.5, since they both have a mixed float and int.

- Normal precedence rules apply, and conversions don't happen until they need to. For example, `1+2+"glow"` is `"3glow"` (since 1+2 goes first.) But `"glow"+1+2` is `glow12` (since `"glow"+1` goes first.)

Examples using these rules are fun, and give more chances to see how important the type is to the computer:

```
print( 3/2 + 1.5f ); // 1 + 1.5f = 2.5f

print( (0.0f + 6) / 4 ); // 1.5f

print( 12/8/2.0f ); // 1 / 2.0f = 0.5f

print( "abc" + 1 + 2 ); // abc12
print( "abc" + (1 + 2) ); // abc3
print("abc" + 2 * 3 ); // abc6
```

The steps for all of these are the same: figure out which math symbol goes first, find the types of the two things around it, fix mixed-type problems if you need to, and do the math. Then repeat until it's done.

3.2 Variables, declarations, simple assignment

Computers can use x and y somewhat like in algebra. But first, every variable needs to have an official Type – `string`, `int` or `float` Before you can use any variable, you have to say what **type** it will permanently hold.

The command to say "this variable is this type" is called a **declaration** (that word will sometimes appear in error messages, so good to know it.) The rule for **declaring** a variable is simple: write any type, a space, then the variable name. Ex's:

```
int x;
float y;
string z;
```

Those lines declare `x` as an `int`, declare `y` as a `float` and `z` as a `string`. Notice the semi-colons – declarations are statements, so need the ending semi-colon.

It often makes sense to think of the declaration as creating the variable, or setting it up. So the `int x;` line says "please make a little box for variable x, which can hold whole numbers."

Here are some rules for declaring. You can skim them, since you'll pick them up from examples later:

- You have to declare a variable before you can use it. If you write `x=6; int x;` the computer will complain that `x` wasn't declared. The real error is those lines should be flipped.
- The type is permanent. In the above, `x` is locked as an `int`. It can never hold `"cow"` or `3.0f`. Put another way, for every `x` in the program, you won't know the exact value ahead of time, but you know it will always be some integer.

- You can't declare a variable twice. `int x; string x;` gives an error about `string x;`. Even `int x; int x;` gives you an error on the second `int x;`.
- A shortcut: you can declare several variables of the same type, using the same declaration, by putting commas between them: `int a, b;` is a shortcut for `int a; int b;`. You can have as many as you like: `float w, x, y, z;` is fine. The comma is special for just this rule.

3.3 Identifiers

In a computer, a variable name can be just about anything. `x, y, n1, n2` are fine. But also `weight`, `minutes`, `phoneNumber`, `partsPerMillion`, `ppm` . . . . The technical term for "words we can make up" is **identifier**.

Rules for identifiers (variable names you can pick):

- Can use numbers, upper/lowercase letters, or underscores. No other symbols, no spaces. Can mix them in any order, except can't start with a number.

  These are bad names, but legal: `a1b2c3` `a_b` `_1` `___x___` `zero`. These are illegal: `4frog` (starts with a number,) `goat-home` (no dashes – the computer thinks this is goat minus home,) `tail Len` (no spaces.)
- Can't use a built-in computer word (can't make a variable named `string` or `void`.) In a modern editor, it will turn blue, so you'll know. You can vary the case, or include a built-in word. So `Float`, `float1` and `floater` are legal (but not very good) variable names.
- Bad spelling doesn't matter, since the name never means anything. `int totel;` (misspelled total) will work fine (but it might confuse people about what a `totel` is, so try to avoid it.)

3.3.1 Style

Variable names never really matter – you could use `a`, `b` . . . . But good variable names make the program a lot easier to read. Here are some notes:

Normal variables almost always start with a lower-case letter: `n` instead of `N`, or `count` instead of `Count`.

It's common for variable names to be a few words pushed together, like `dogWeight` or `catWhiskerLen`. Capitalizing the start of new words like that is sometimes called camel-casing.

The other way is with underscores for the spaces, like `cat_whisker_len`. But that's less common now.

There's an old style where you put the letter for the type in front, like `int iCats;` instead of just `cats`. That's called Hungarian notation – you can look it up. I mention it to show that people can take variable naming very seriously.

For just a quick name, `w` is usually a string. And if you see `i` and `j`, those are usually integers.

It's common to start with an underscore, like `float _len;`, to mark a variable as being sort-of hidden (we'll use that style in a much later chapter.)

Consistency is probably the most important. Don't use `catSize` along with `szDog` – pick one way or the other.

But it's not too difficult to change the names, once you learn Search and Replace.

3.4 Simple assignment, var use

We can put values in variables with lines like x=7;. The official name is an **assignment** statement. Once we have a value in a variable, we can use it like algebra: when the computer sees the variable, it replaces it with the value.

Here's a small program (inside of Start()) to show this:

```
int n;  n=8;

print( n );    // 8

print( n+1 ); // 9
print( n*2+1 ); // 17
print( 3+n+1 ); // 12
print( n*n ); // 64

print( "n" ); // n
print( "n+1" ); // n+1
```

The first looks up the value of `n`. The next four also look up `n`, then do math. Nothing special. `print(n*n);` is a little different – it looks up `n` twice, to get `8*8`.

The last two are showing how the "quotes make it a string" rule also applies to variables.

This next example has two variables with fun names:

```
int cats;
cats = 6;
int dogs;
dogs=4;
```

```
print("number of cat feet:");
print( cats*4 ); // 24

print("total number of pets:");
print( cats + dogs ); // 10
```

That's a more common sort of program. In your school math class you might have had things like "`x` stands for cats and `y` stands for dogs". In a program we can also do that, but it's easier to have `cats` stands for cats. But `x`, `n`, `cats` – to the computer they're just int variables.

An example with floats:

```
float length;  length = 3.5f;
float width;  width = 9.1f

print("area:");
print( length*width );

print("perimeter:" + (length*2 + width*2) );
```

In this one, `length` and `width` look like they have some special meaning, but they're just normal variables, no different from `x` and `y`.

I printed area on two lines to make it easier to read. Perimeter is jammed into 1 big line using a `+`. That's more common. Notice how the extra parens are required to force it to do all the math first. Without them we'd get `perimeter: 718.2` (7 and 18.2, smashed together).

An example with string variables. This one uses string math to combine the variables with some words:

```
string animal; animal = "mouse";
string name; name = "Henry";

print( "You are a " + animal );
print( "I think I will call you " + name );
```

This is still just math – the computer doesn't know what it's doing. For example, if the animal was `aarvark` it would print simply `"You are a aarvark`, without knowing to change `a` to `an`.

3.4.1 More assignment statement rules

In regular math, x=3 and 3=x are the same thing. In a program, = works a little differently. It says to change the variable on the left, into the thing on the right. This means the thing on the left must be a single variable. The right side

can be any type of math. The computer works the math, then puts the result in the variable, erasing the old value.

That's why we make a point of calling `=` an **assignment statement**.

In the example below, `x` is assigned using a formula. It's no different than a formula inside a `print`. Same as before, the computer works out the math and converts it to a single number before doing anything else:

```
int x;
x = 5+2*2;
print( x ); // 9
```

The same idea using strings. For real, no one would ever do this:

```
string w; w = "cow" + "bell";
print( w ); // cowbell
```

The computer won't do any work to "solve" equations. It will do regular math to work out the right side, but the left side is just where the answer goes. It has to be a single variable. These are errors:

```
int x;
5 = x; // ERROR. Left side must be a variable
x + 1 = 7; // ERROR. Left side must be a variable by itself

int y; y=5;
x + y = 8; // same error. Left side must be a variable all by itself
```

Like other statements, assignments run in order. In algebra, if we see a system of several equations, we know we should move them around, and pick one to solve first. Assignment statements are much simpler than that.

A good way to think of assignment statements, is each variable is a box with its current value. When you have something like `x=4+3*9/2;`, it does the math, `x` becomes 17, and the line is done. All that matters is `x` is now 17. The way it got there isn't important any more. An example:

```
int n1; n1 = 4+2; // 6
int n2; n2 = n1+3 // 6+3 = 9
int n3; n3 = 2*n2-4; // 2*9-4 = 14
```

Line one puts a 6 in `n1`. Line two simply looks up `n1` and sees 6, then adds 3. Line three simply looks up `n2` is 9, then finishes the math. No matter how complicated it was to compute a variable, afterwards it's just a simple number in a box.

Extra assignment statements erase the variable's old value. In this next example, the second equals "wins." `x` is 7 for just a little while, then changes to 2. The line `x=7;` is a waste, except as an example:

```
int x;
x=7;
x=2;
print( x ); // x is 2
```

But it still does things one at a time. This next one prints 7 then 2. Changing `x` to 2 won't go back and change the first print:

```
int x; x=7;
print( x );
x=2;
print( x );
// output is: 7, 2
```

One assignment statement that can fool you is `x=y;`. In Algebra, we're so used to `x=y` and `y=x` being the same thing. In computer assignment statements, they're completely different – copy `y` to `x`, or the reverse:

```
int x; x=4;
int y; y=20;
x=y; // now both are 20
print( x ); // 20 (it was changed to y)
print( y ); // 20
```

If we flipped it to `y=x;`, then it would print 4 both times:

```
int x; x=4;
int y; y=20;
y=x; //  <- this line is different. Now both are 4
print( x ); // 4
print( y ); // 4 (it was changed to x)
```

For the variable in front of the `=`, the old value never matters. If you have `x=3;`, no matter what `x` used to be, it's just 3 now. There's no undo, so the old value really doesn't matter.

Even strings work that way. It seems like the old length might matter, but it doesn't. Say you have `string w; w="abc";` then shrink it with `w="z";`. There's no left-over junk from when it had 3 letters – it's completely just `"z"`.

3.4.2 Initialization, unassigned vars

Variables have no starting value. `int x; print(x);` gives the error *Use of unassigned local variable x.* You get the same error with `int x; int y; y=x;`. Since `x` doesn't have a value yet, we can't copy it anywhere.

You might think `int`s should automatically start at 0. We made it an error on purpose to remind ourselves to always put something in there.

Giving a variable a starting value is often called **initializing** it. That's not an official term – if a variable got up to 100 and you reset it to 0, you could say that you're initializing it back to 0, or not. There's no special initialize statement. We just say the first assignment is initializing it.

In some circumstances, the computer will give a starting value. 0 for ints, `0.0f` for floats, and `""` for strings. There are rules for when it will and won't. But it's usually better not to count on that, and always give it an `x=0;` to be sure.

3.4.3 Computer math with variables; types

We know when the computer sees `3+"6"` it checks the types first, then decides what kind of `+` to use. It works the same way with variables. If we see `w+n` we check the types by looking how they were declared.

That's really why we have to declare variables. If `w` is a `string`, even though we won't know exactly what's in it, we know 100% that `w+n` uses string-plus.

Some examples of mixed math with variables:

In this first example, we divide two variables by 2. The `int` variable drops the fraction, the `float` doesn't:

```
int x;  x=7;
float y;  y=7.0f;

print( x/2 ); // 3 -- both are ints
print( y/2 ); // 3.5 -- float divided by int
```

Here's the same thing, except we divide by `x` and `y`:

```
int x;    x=2;
float y;  y=2.0f;

print( 5/x ); // 2 -- both are ints
print( 5/y ); // 2.5 -- int divided by float
```

This next one compares adding 3 to a `float` and `string` variable:

```
float a;   a=7.0f;
string b;  b="7.0f";

print( a+3 ); // 10
print( b+3 ); // 7.0f3 -- a 5-letter string
```

This next one uses addition where *both* sides are variables. As usual, it checks the type of both; if either is a string, it does string-combine. For real, we wouldn't have such terrible variable names as `a`, `b`, `c` and `d`. But they're good for a do-nothing example like this, which just shows the rules:

```
string a; string b; a="4"; b="6";
int c; int d;  c=4; d=6;

print( a+b ); // "46" -- both are strings, so concatenate
print( c+d ); // 10 -- regular addition

print( a+c ); // "44" -- string+int converts to string
print( d+a ); // "64" -- int+string converts to string
```

There aren't any new rules here. It's just the same old rule: "if one part of a `+` is a string, do string combine."

3.4.4 mixed-type assignments

Assigning the wrong type to a variable is another way to show how much the computer cares about declarations and types. If you try to assign the wrong type, the computer will either fix it, or give an error.

For all these, I'll use `int n; float f; string w;`.

Assigning an `int` to a `float` variable works. It uses the "promote" rule from earlier:

```
f = 4; // legal. counts as f=4.0f
print( f/3 ); // 1.3333 -- f is still a float
```

The interesting thing is `f` is always a `float`, since we declared it as one. In `f=4;`, there's no way the computer is going to try to turn `f` into an `int`.

No other wrong type-assignments work. `float` into `int` is an obvious problem – you'd have to drop the fraction. We decided it's better as an error:

```
n = 3.2f; // ERROR -- can't convert float to int
n = f; // same error, since f is a float variable
```

It's not even legal if the float ends in point-zero. That would be a confusing exception, and would look weird:

```
n = 3.0f; // error - cannot convert float to int
f = 3.0f; n=f; // same error
```

`string`s won't let you assign `int`s directly:

```
w = 5; // ERROR -- can't convert int to string
w = n; // same error
```

That seems funny, since we already know the computer can fix the 5 in `"ant"+5`. But we decided it's better to make you do it yourself. Here's the common, clever trick:

```
w = ""+5; // legal (but it would be easier to write just "5")
w = ""+n; // legal. Turn a number variable into a string
```

Even though `""` has zero letters, it still counts as a string, so it forces `n` to convert. It's a very sneaky use of the empty string.

Assignment also uses the "do things in order" rules. Of course, the assignment always goes last, and the computer won't look ahead to see what type it is. Fun examples:

```
f = 12/5; // 2.0f
f = 2.5f + 8/3; // 4.5 -- 2.5+2
```

In the first line, the computer doesn't care that 12/5 will be going into a `float`. It sees two integers, so rounds the result down to 2. Later, it converts into 2.0.

The next one is the same idea. `8/3` goes first, and is just a 2. It doesn't look ahead to the floats.

3.5 More assign and variable rules and examples:

One of the most common things we want to do is to "work on" a variable.

Typical things are adding 1 to a counter, or adding `n` to a running total. Or doubling a variable, or increasing it by 10%. We can do these with regular assignment statements, with a trick.

`x=x+1;` increases `x` by 1. The trick is to remember it's not algebra. The computer calculates the formula on the right, and puts it into `x` on the left. An example:

```
int x; x=4;
x=x+1;
print(x); // 5
x=x+1;
print(x); // 6
x=x+1;
print(x); // 7
```

You can read it as "the new value of `x` is the old value plus 1." The cool thing is that it's not a special rule. Suppose `x` is 6 right now. `x+1` is 7, so `x=x+1;` makes `x` be 7.

This is one of our main tricks. `x =` *x with some math* `;` is the basic way to change a variable.

The best way to try this is to write `x` on a piece of paper with a line under it, and think of it as a box. Then play computer.

`x=4;` puts a 4 in it. Then run `x=x+1;` in your head. That makes `x` be 5, so cross out the 4 and put a 5. You never have to think about the history of `x`. Whenever you want to know the current value, just look in the box. That's what the computer does.

You usually have to hand-run a few programs this way, to get a feel for the rules.

The same trick can add 5, subtract 1, or double us:

```
int x; x=4;
x=x+5; // 9

x=10;
x=x-1; // 9
x=x-1; // 8

x=12;
x=x*2; // 24
```

It works just as well with `float`s. But I'll mostly stick with `int`s since they're simpler. This increases us by 50%:

```
float f; f=20.0f;
f = f*1.5f; // f is 30
```

3.5.1 examples

These are just traditional exercises to get practice with the idea of changing a variable. They do nothing useful, except to be examples.

The column to the right has the current value (what you'd get if you were crossing out and changing.) For real, you'd use a piece a paper with room to cross out old values.

```
               n
              ---
int n; n=8;    8
n=n+1;         9
n=n+5;        14 <- adds 5 to the previous 9
n=n/2;         7 <- divides the previous 14
n=n*10;       70
```

This is more of the same, but with some subtraction and a fake-out:

```
               a
              ---
```

```
int a; a=8;     8
a=a-1;          7
a=a-2;          5  <- subtracts 2 from the previous 7
a=a-9          -4
a=12;          12  <- just a=12;, not a=a+12;
a=a-1;         11
```

Second from the bottom, `a=12;` is there to trick you. Same as always, that wipes out the work and just makes it 12.

A thing to be careful about is thinking `x=y;` creates a "link". It's easy to think `x=y;` means that `x` is now tracking the value of `y`. But assignment is always a 1-time thing.

Here's an example where I copy `a` and `b` around to show that:

```
             a   b
            --- ---
int a,b;
a=7; b=a;    7   7
a=11;       11   7  <- b didn't follow a
a=a+3;      14   7  <- b still didn't change

a=b;         7   7
b=0;         7   0  <- a didn't follow b

print("a=" + a + "b=" + b); // a=7 b=0
```

That last line is a typical testing print. If I was running this for real, I'd paste it after each change.

This is more copying two variables back and forth, but with extra math:

```
              a   b
             --- ---
int a, b;
a=5; b=20;    5  20
b=a+1;        5   6
a=9;          9   6  <- b stays 6. didn't follow a
a=b*2;       12   6
b=a*3;       12  36
a=a-b/10;     9  36  <- b/10 is 3, a is 12-3
```

Then one more example, nothing special, but with three variables:

```
  int a, b, c; a=0; b=0; c=0;

           a   b   c
```

```
         --- --- ---
  c=5;      0   0   5
  a=c+9;    14  0   5
  b=a-2;    14  12  5
  c=a/2;    14  12  7
  a=c*3;    21  12  7

  print("a="+a+" b="+b+" c="+c);
```

Later on, we'll be using things like these where they actually do something useful.

This "change me" trick also works with strings, but all we have is +. An example, `w=w+"s";` adds an `s` to the end of `w`. Even more fun, we can also add to the front. Examples:

```
string w; w="snake";
w=w+"s"; // snakes
w=w+"h"; // snakesh
w="Go "+w; // Go snakesh
```

3.6 Declare/assign shortcuts

As mentioned before, you can combine declares of the same type, using commas. `int a; int b; int c;` can be shortened to `int a,b,c;`.

You're also allowed to combine a declare and an assign. `int frogs; frogs=7;` can be combined to make `int frogs=7;`

Examples:

```
int n=5;
n=n+1; // 6

string w1="shark";
string w2="fish";
w1 = w1+w2; // "sharkfish"
```

The starting value is in no way special. `int n=5;` doesn't make the 5 any more sticky or special than regular `n=5;`.

You're also allowed to combine these shortcuts, doing multiple declares and assigns at once:

```
int n=0, maxCows=20;
// long version:
// int n; n=0; int maxCows; maxCows=20;
```

```
string a1="cat", a2, a3="duck";
// a2 is still empty. We don't have to use equals on them all.
```

Some people make a big declare over several lines with comments in the middle. This next thing just declares six variables (and assigns four of them.) I just made up the meaning (but real programs have comments like this):

```
float g1=1.4f, g2=-0.43f,  // cougar coefficients
      h1, h2=1.0f, h3,     // harfle values
      glom=0.0f;           // standard glom factor
```

You're also allowed to chain assign statements. It makes everything equal to the last item:

```
a=b=0; // same as a=0; b=0;

w1=w2=w3=""; // same as w1=""; w2=""; w3="";
```

About the only time you see this is for a "reset," like the examples above.

This next thing is just a trick. I thought this was a good spot for it:

Suppose you want to divide 2 int variables `a/b` to get a decimal answer. For example `a` is 6, `b` is 5 and you want to get `a/b` is 1.2. With numbers you can write 6.0f. But you can't write `a.0f`. The trick is to combine one with a float – multiply by `1.0f`:

```
int a=6, b=5;
print( a/b ); // 1, since they're both ints
print( 1.0f*a/b ); // 1.2
```

I like this trick because it's just being clever with what we have – it's not a new rule. `1.0f*a` goes first, turning 6 into 6.0f.

For fun, `a/b*1.0f` won't work – `a/b` goes first, and drops the fraction. But `a/(b*1.0f)` works. You could also add 0, like `(a+0.0f)/b`.

3.7 Semi-useful variable and assign examples

Our programs can't do much so far – we can't even read input yet. So the best we can do for real programs is pretty fake. But these give an idea how a real program would look.

As usual, they go in `Start()`, inside the {}'s.

The first one is just the area/perimeter example, with extra variables and a longer print:

```
float wide=6, high=8;

float area = wide*high;
float perim = wide*2 + high*2;

print("A " + wide + " by " + high + " box has:");
print("area of " + area + " and perimeter of " + perim);
```

It's a common trick to compute the answer into a variable, then print. It's easier to focus on just the math first, and focus on printing the variable second. It also allows us to pick helpful variable names.

For this next one, I wanted something that computes things in steps, and the best I could do was the damage-per-second calculations in games. In this one, a fireball shoots every 2 seconds, doing 40-60 damage. The average damage is 50, which is 25/second:

```
int dMin=40, dMax=60; // fireball min/max damage
float castTime=2.0f;

float dAvg = (dMin + dMax)/2.0f; // 50 average each fireball

float dps = dAvg/castTime; // damage-per-second
```

To print it, I'm going to go a little crazy and "compute" the output sentences into strings first. That's overly complicated for just printing it, but I wanted to show computing a string:

```
string line1, line2;
line2 = "Average damage is " + dps + "/second";

// line1 is the raw stats for a fireball:
string wRange, wTime; // description of damage range and cast time
wRange= dMin + " to " + dMax + " damage";
wTime= "every " + castTime + " seconds.";
line1 = wRange + " " + wTime;

print(line1); // 40 to 60 damage every 2 seconds
print(line2); // Average damage is 25/second
```

I computed `line2` first, since it's easier. That's another small advantage of putting things into variables before printing them.

This next example uses the "work on a variable" idea to compute 10% compound interest for three months. My plan is to just walk `months` and `cash` through each month. This computes for one month, then uses cut&paste for the next two:

```
float cash=100; // our starting cash. Anything works
int months=0; // how many months have gone by, so far

print("starting with $" + cash);

cash=cash*1.1f;
months = months + 1;
print("Month " + months + ": $" + cash); // Month 1: $110

cash=cash*1.1f;  months = months + 1;
print("Month " + months + ": $" + cash); // Month 2: $121

cash=cash*1.1f;  months = months + 1;
print("Month " + months + ": $" + cash); // Month 3: $133.1
```

A drawback of growing a variable is you can't look at old values. For example, we can't print them all in one big chart at the end. Just to show we can, each month could have it's own variable:

```
float cash0 = 100; // starting money, 0 months interest
float cash1, cash2, cash3; // cash after that many months

// each month is 10% more than the last:
cash1 = cash0 * 1.1f;
cash2 = cash1 * 1.1f;
cash3 = cash2 * 1.1f;

print("One month: " + cash1 + ", second: " + cash2 + ", third: " + cash3);
// One month: 110, second: 121, third: 133.1
```

3.8 More error messages

The same as last chapter, let's preemptively make some errors and see what messages they give us.

A funny thing about many of these errors is they seem like things the computer could fix for us, without giving an error. But getting more errors is better. All these nit-picky errors are the computer's way of making sure your finished program is crystal clear, with no misunderstandings.

Here's the message if you forget to declare a variable:

```
void Start() {
  gpm=6;
}
// The name 'gpm' does not exist in the current context.
```

I think that's a pretty good message. It doesn't say "forgot to declare" because you might have just spelled it wrong:

```
void Start() {
  int gpm;
  gmp=6; // oopps! flipped the m and p
}
// The name 'gmp' does not exist in the current context.
```

Using a variable before declaring it gives a straight-forward error message:

```
void Start() {
  gpm=6; // too soon
  int gpm;
}
// A local variable ‘gpm’ cannot be used before it is declared
```

That's one where it didn't need to be an error. But using before declaring looks so funny that we decided to make it illegal.

Here's the error again for using a variable that doesn't have a value yet. Most people would say `gpm` was uninitialized, but the computer uses *unassigned* (which is the same thing):

```
int gpm;
print("value is " + gpm ); // ERROR - gpm has no value
int x; x=gpm; // same ERROR - gpm has no value

// Use of unassigned local variable ‘gpm‘.
```

Here are the errors from assigning the wrong types. A funny thing is that the exact wording changes, but they're all just a way of saying "type mismatch":

```
int x; x=1.2f; // ERROR float into int
// Constant value ‘1.2‘ cannot be converted to a ‘int’

string w; w=6;
// Cannot implicitly convert type ‘int‘ to ‘string‘.

int y; y="12";
// Cannot implicitly convert type ‘string‘ to ‘int‘.

float f = 1.3f;
int x=f;
// Cannot implicitly convert type ‘float‘ to ‘int‘
// An explicit conversion exists (are you missing a cast?)
```

Different wording aside, I think they all give you the idea. I didn't make up that last 2-line error for float into int. It really does have that annoying, condescending second line.

If you forget the `f` on the end of a `float`, the exact error you'll see is another cannot-convert:

```
float f=2.5; // forgot the f on the end
// Literal of type double cannot be implicitly converted to type 'float'
```

Now for some different errors. A common mistake is to forget the `x=` in front. That gives a funny-looking error:

```
int x=6;
x+1; // oops! meant to write x=x+1;

// Only assignment, call, increment, decrement, and new object
// expressions can be used as a statement.
```

The words in this one don't mean anything. It's a generic message saying you've completely confused it. It sometimes means you forgot something.

Declaring a variable twice is an error. Just so you know, "local" and "scope" are good technical terms, and we'll see them later:

```
int x;
int x; // ERROR

// A local variable named 'x' is already defined in this scope
```

The commas trick is only for the declare-and-assign shortcut. Using it in normal assigns is an error:

```
int a=4, b=8; // this is fine

a=6, b=10; // ERROR. change to semi-colon after a=4

// Unexpected symbol ',', expecting ';'
```

I usually get this error when I cut&paste that top line somewhere below. I usually remember to take out the `int`, so I don't double-declare, but then I leave in the stupid commas.

Here's the error for using a built-in word for a variable name. *Unexpected symbol* means it knows what `void` is, but not what's it's doing here. Then it gives you a bonus error for the `=`, since it didn't even know this was suppose to be a declaration:

```
int void=7; // ERROR

// Unexpected symbol ‘void’
// Unexpected symbol ‘=’
```

3.8.1 Run-time errors

All the errors so far happen right away, while the program is scanning the code. There's a different type of error you can only get when the program is running, called a **run-time** error.

For real, we won't get these until later. But we can make one now by dividing by zero. Just `4/0` won't fool it – the computer spots that in advance, as a regular error:

```
print( 4/0 ); // ERROR
// Division by constant zero
```

But if you use variables, the computer can't tell ahead of time. The program will run. But then crashes on that line with an error:

```
int a=4, b=0;
print( a/b ); // not an error, yet
print("this will never print");
```

This will run. But when you press Play it crashes on `a/b`:

```
//DivideByZeroException: Division by zero
//testA.Start () (at Assets/testA.cs:12)
```

It still gives you the problem line and what the problem was, and double-clicking jumps there. *Exception* is just the technical term for an error while it's running.

We won't see many of these yet. But eventually reading syntax errors and fixing them will be no big deal. These run-time errors will be the real problems.

3.9 Doubles vs. floats

Having to put an `f` on the end of 2.5 seems like a bad way to make a computer language. If you want to know why, this section explains it. But you don't need to know. I'm not going to use this part later.

The short version is: most programs use just regular `2.5` for numbers, with no `f` on the end. They write decimal numbers as `double n; n=3.2;`. `floats` are a special type, made to save space.

The basic problem is that computers can't store `1/3`rd. It's a repeating value, so they store it as 0.333333 with the rest chopped off. Storing only a certain number of places is called **floating point math**

Way back we decided the basic floating point type held 6 digits, which was plenty. If you really, really needed extra accuracy, a special one held 12 digits. We named it `double`, for double-sized-float. That's a bad name, but almost no one ever used them, so it was fine. The rule was that `3.5` was a regular float, and `3.5d` was a double. Math with doubles took twice the space, but was more accurate past the 6th decimal place.

This was back when computer memory was measured in kilobytes. When memory got cheap and was sold in megs, things flipped. We didn't change the names, but normal programs started using `double` for everything. Floats were only for extreme space-saving. It was too late to fix the weird names, but we flipped the rule for constants. Now 3.5 is a double, and 3.5f is a shorter float. The weird letter is still for the one no one ever uses.

But graphical games, especially ones for cheap, power-hungry cell-phones, are one of the few places where space is vital. Unity's position, color, size, everything, use `float`'s for speed and size. Since we're going to be setting those, we'll need to use floats to do it.

We can declare and use doubles. `double a=0, b=1.5; print(b);` is totally legal for us. But it seems easier to stick to just floats.

Now onto the error in `float f = 1.5;`. It's really the same error as `int n=1.5f;`. 1.5 is a double, with 12 digits, and `f` only holds 6 digits. We're chopping off the last 6. In this case, it's only 6 zero's, but it could have been something important, so we always get the error.

Chapter 4

Input, more output, testing

This section starts with Unity3D tricks: a simple way to get input, another way to get output, and how to make the program keep running and doing things. But that won't take long, and we'll be able to use them to write some more interesting programs.

Most of this chapter will be about how we can use variables and the `n=n+1;` trick to make the computer do things.

Then, near the end, we'll move a Cube around on the screen, by adding just one extra command.

4.1 Input

Unity uses an old trick where it can show you the "box" for a variable. Typing into it is a shortcut to give that variable a starting value. Then, while the program runs, we can watch our variables change, or even cheat by hand-changing them.

This seems like cheating, but the secret is all real input is about putting a value in a variable. For example, suppose we had a drop-down with five options. There would be rules to set it up, but the end result is an `int` changing from 1 to 5. Output is the same way. If we have an auto-centered text-box with a pretty font, we'd still create a `string` and assign it to the text-box.

So changing and looking at variables directly is a completely fair way to do input/output. It's ugly and confusing if you're not a programmer. Luckily we are.

To do it we have to declare the variable outside of `Start()` with the word `public` in front. Here's a silly sample program that lets us see `x`, `w1` and `w2`. I named it `testB`, but you could just as easily reuse `testA`.

Note how the declarations are outside of Start, but are still inside the big {}s for `testB`. The indentation (even with Start) is also to help us see that:

```
Using UnityEngine;
using System.Collections;

public class testB : MonoBehaviour {

    public int x; //  outside of Start, public in front
    public string w1; //  same

    public string w2; // this one, too

    void Start() {
      x = x + 1; // just for fun
      w2 = w1 + x + "ous";
    }
}
```

Once you have this script set up (dragged onto the Directional Light) you'll see the cool part. Select the Light and look at the Inspector. If it isn't already, click the arrow to "pop open" the tab with the script. You should see new lines with `x 0`, `w1` and `w2`. You can click and type in new values (the boxes for strings are sometimes further right than you think, so keep clicking along the row.)

When you press Play (don't press Stop yet) you should see `w2` and `x` change. Pressing Stop makes them snap back (it just does.) Try entering `"ham"` for `w1`. Pressing Play again should show `w2` is `"ham1ous"`.

Not super exciting, but it's a quick, easy way to test this program for lots of different inputs and see the results. Try `3+4` for `w1`. You should see `3+41ous` for `w2`, proving it's using string-add.

Here are some notes, to explain this more:

- Using this trick to set starting values is a cheat – don't let it confuse you about how programs work. When you press Play, the Unity system jumps in and slaps those Inspector values into the variables. Then it runs Start() as normal.
- The rule about variables snapping back when you Stop is suppose to make things easier, for when you test real programs. Don't worry too much about it.
- Typing new values *during Play* won't recompute anything, yet. Entering `cow` for `w1` won't update `w2`. It would be nicer if it did, and later programs will.
- Fun fact: the system saves what you type in there. Even if you quit Unity and come back, whatever you entered for `w1` will still be there.
- Putting variables outside of `Start` with `public` in front is a normal C# rule. We'll use it for its real purpose later on. Unity happens to piggy-back on it to decide what variables to show you.

- Notice how we didn't need quotes for strings in the inspector. It already knows everything you type in `w1` is one string. We only need quotes inside the program.

- Whatever you enter for `w2` doesn't matter, which should make sense. The program only uses `w2=`, and we know that completely erases the old value. In out minds, `w2` is for output.

Here's a slightly different "Mad Lib" example. `animal`, `food` and `place` are the input variables:

```
Using UnityEngine;
using System.Collections;

public class testB : MonoBehaviour {

    public string animal, food, place;
    public string w;

    void Start() {
      w="The " + animal + " eats " + food + " in the " + place;
      print(w); // as a double-check
    }
}
```

We can enter values, Play, and check `w` to see how it looks. The neat thing is, a real program would compute `w` just like this does. The only difference is it would then copy it into a fancy text-box somewhere on the screen.

4.2 Using Update to make more interesting programs

A traditional program runs once, outputs, and is done. `Start` in Unity works this way.

There are a few tricks to making a program that runs over time. Unity uses a common one. You write a regular run-once program, but you tell the system to run it over and over. That's what `Update` is for.

The original starter program looked something like this (showing the important parts):

```
  void Start() {
  }

  void Update() {
  }
```

Anything you put inside the `Start` curly-braces will run once. Anything in the `Update` curly-braces runs over-and-over. You're allowed to use either one by itself, or both.

Here's an example showing how they work together. I'm putting them each on one line, to save space:

```
void Start() { print("begin now"); }

void Update() { print("x"); }
```

If you press Play, wait a little, then Stop, you"ll see one "begin now" followed by x's blasted down the page (you may have to scroll up.) If you only see a single `x`, you probably have **collapse** checked on the Console, which won't show duplicate lines. Just uncheck it, at the top.

Notice how the inside of `Update` doesn't have anything about repeating. It's completely normal. The system just knows it should run `Update` over and over.

Here's a program taking advantage of Update(). It adds one to `x` over-and-over and lets us watch:

```
public int x;

void Update() { x=x+1; }
```

This is fundamentally no different than when we had several `x=x+1;`'s in a row inside Start(). We're just using a trick to make it run lots, spread over time.

If you know games, you may notice it's going too fast. Update should be running 60 times each second. A quirk of Unity is when you run things this way it goes as fast as possible, usually a few hundred times per second. That's fine for our purposes.

If you don't know games, each time Update runs is usually called a `frame` or a `tick`.

We can use the Update trick to improve the Mad Lib program. Moving it there makes it so changes while running immediately update the answer (this version is a little different than before, but the same idea):

```
  public int eatAmt;
  public string animal, food;
  public string w; // output

  void Update() { // <- in Update instead of Start
    eatAmt = eatAmt + 1;
    w = "The " + animal + " eats " + eatAmt + " " + food + ".";
}
```

I had `eatAmt` keep increasing just so we can see this really is running over-and-over. It's not super exciting, but we can now change `animal` or `food`, while running, and have `w` change.

Here's a little more interesting program. It computes hours and seconds based on minutes:

```
public float minutes; // input
public float hours, seconds; // outputs

void Update() {
  hours = minutes/60.0f;
  seconds = minutes*60.0f;
}
```

If we type (while this is running) 120 into the minutes box, we should see hours snap to 2, and seconds snap to 7200.

The Update trick works with any other "change me" assignments. This next program makes `y` go down very slowly, and computes 5% interest on `x`:

```
public float x, y; // <- shortcut, but we still get a box for each

void Update() {
  y = y - 0.01f; // <- floats let us use a very small number
  x = x * 1.05f;
}
```

At first `x` won't change, since anything times 0 is 0. But if you change it to 1, it will go up slowly, then faster and faster, like compound interest. That's not really useful yet – just showing how the "change me" assign tricks from the last chapter work here.

4.3 declare = shortcut and Inspector vars

We can use the declare-and-equals trick with `public` "Inspector" variables, but there are some tricks. This will start `x` at 100, for real and in the Inspector:

```
public int x=100;

void Start() {
  x=x+1;
}
```

But what happens if you change Inspector `x` to -999? The rule is: the Inspector wins, and `x` starts at -999, even though the code still says it starts

at 100. That's because Unity jumps in and changes it to -999 before running Start.

This is horribly, horribly, horribly confusing if you try to read the program and forget. For real, variables never just magically change their values. But it makes testing programs much easier – what you see in the Inspector is always what you get.

One way not to be confused by this rule is not to use the assign trick. But I do it anyway, to give the Inspector that starting value. If you use Unity for real, this trick is handy.

4.4 Replacing constants with variables

Here's a program that goes up by 1 each tick:

```
public float x;
public float xSpeed = 1.0f;

void Update() { x = x + xSpeed; }
```

The old version used `x=x+1;`. The new version uses `x=x+xSpeed;`, but `xSpeed` is always 1, so it's the same.

It seems overly complicated, but there's one advantage: we can change `xSpeed` while we're running.

Press Play, let `x` go up by 1 for a while, then type 0.1 for `xSpeed` – `x` will go up 1/10th as fast. Try -1 and `x` starts going down. 0 for `xSpeed` makes it stop (it's still adding, just adding 0.)

Here's an even more interesting version. It starts out increasing by 1, but then increases by a little less each time (it has one new line):

```
public float x;
public float xSpeed = 1.0f;

void Update() {
  x = x + xSpeed;
  xSpeed = xSpeed - 0.005f; // slow down a tiny bit
}
```

`x` will go up, slow down, almost pause, then go back down. That's kind of neat, for a 2-line program.

This is one of the Big Ideas in programming: they're your variables, and they mean what you say they mean. `xSpeed` is how fast `x` changes, because we made it that way – because of the first line. `xSpeed` of 1.0 means `x` is going up quickly, 0 means it isn't changing, negative means `x` is going down. Those

numbers would mean different things for some other variable, but they mean this for `xSpeed`.

In the second line `xSpeed=xSpeed-0.005f;` means "make `x` slow down." It's also just a regular line that subtracts a little from a variable. But it means "slow down" in this program.

4.5 Math shortcuts

`numOfCats = numOfCats + 1;` adds 1 to how many cats we have, but it seems like a waste having to write `numOfCats` twice.

There's a family of shortcuts for modifying a variable, so you only have to write it once.

`numOfCats += 4;` is a shortcut for `numOfCats = numOfCats + 4;`. The `+=` counts as one symbol, so it can't have spaces or be `=+`. You can think of it as saying "numOfCats gets increased by 4."

It also works for floats and strings:

```
x+=0.1f;  // same as x=x+0.1f;
w+="ly"; // same as w=w+"ly";
// no shortcut to add to the front of a string
```

The same shortcut works with other math symbols: `-=`, `*=` and `/=`:

```
numOfCats *=2; // numOfCats = numOfCats * 2;
cows -=6; // cows = cows - 6;
q /=2; // q = q/2;
```

These tricks are very common. You almost never see someone write this out the long way. But they're only shortcuts – they don't even run any faster than if you use the long way. Use them when you get sick of writing the same variable twice.

There's one tricky part – the entire right side is always computed first. Ex:

```
x *= 1+1; // same as: x = x * 2;
cows *= n+1; // same as: cows = cows * (n+1);
```

You usually don't use the shortcut for anything complicated like this. I just write these out the long way.

There's a super shortcut for just adding or subtracting 1. That seems excessive, but we do it a lot. `x++;` adds 1 to `x`. `x--;` subtracts 1 from `x`. There isn't even an `=`. Just `x++;` all by itself adds one:

```
// all do the same thing:
x++;
x += 1;
x = x + 1;
```

The official name of `++` is **increment** (coming from the root word, increase.) And `--` is called a **decrement**.

4.6 Fun 3D stuff

With just a little extra work, we can take a program that moves `x` around, and have it also move a Cube on the screen around, using `x`. All we need is a rule to apply `x` to the position of some Cube.

Here are the steps, then an explanation later. Make sure it isn't in Play mode for any of this:

- Add a Cube: click GameObject (top bar) slide down to 3DObject, then select Cube (or any shape, really.) Don't worry about where it is or whether you can see it now.
- The camera should already be in the right spot. But check and change it if isn't. Select the camera, and look in the Inspector, at the top part labeled Transform. It should have (0, 1, -10) for the x,y,z under Position. And (0, 0, 0) for the x,y,z in Rotation. If it doesn't, type those values in yourself.
- Use this script (as usual, I'm leaving out the `using` lines on top.) It's a small rewrite of the changing-`x` code. If you reuse it, change `x` and `xSpeed` to those new values in the Inspector:

  ```
  public float x=-7.0f;

  public float xSpeed=0.1f;

  void Update() {
    x = x + xSpeed;

    // this is the new part:
    transform.position = new Vector3(x, 0, 0);
  }
  ```

- Find the Cube you made (in the Hierarchy panel) and drag this script onto it (select the Cube, drag the script into the blank space at the bottom of the Inspector.) It should look like the usual script – should see `x` and `xSpeed` variables displayed.

  If it's on the Directional Light, Remove it from that one.

- Play should make the Cube move from left to right (then off the edge forever.)

Here's the explanation of the parts:

The things in the Scene are officially called `GameObjects`. The `GameObject->3Dobject` menu has simple testing shapes, like a Cube or a Sphere. Any of them are good enough.

When you make it, the system puts it "in front" of you, which could be anywhere. Since our program will position the object, it doesn't matter where it starts.

Way back, I mentioned how scripts are intended to be on their Cube, which they move around. We're using that rule now. The last line of that script says "move me," the Cube it's on counts as the "me".

This really is the entire movement command:

```
transform.position = new Vector3(x, y, z); // moves us to (x,y,z)
```

If you look in the Cube's Inspector, in the Transform block, in the Position slot, you'll see it changing.

For now, it's "just because." All we need to know about it is that x, y and z can be any `float` or float variable.

There's no natural size of the screen. It depends on the position of the camera and how it's aimed and even the particular size of your Scene/Game area. With the camera in the starting position, the left side is about x=-7, which is why the code starts us there. The right side is about x=+7.

Since the screen is only 14 across, a speed of 0.1 feels about right.

To get a feel how that last line works, we can change things around:

- Start x at -8 (in the Inspector.) That should make the Cube come in from off-screen.
- Change `xSpeed` to 0.005 (half speed) or 0.5. From before, we know this will just make `x` go slower or faster. But it's pretty cool seeing the Cube do that.
- In the last line, change the middle 0 to a 3: `Vector3(x,3,0);`. It should go across more near the top of the screen. So x is across and y is up. `Vector3(x,-4,0);` should make it go more near the bottom.
- Change `x` to 7 and `xSpeed` to -0.1. We know this will just make `x` start at 7 and go down, but now it also makes the Cube go from right to left.
- Change the last line to `Vector3(0,x,0);` (`x` in the second slot instead of the first.) That should make the Cube go from bottom to top. `x` isn't

automatically sideways. What it does depends on how we use it. Putting it in the middle slot makes if be up/down.

- Change the last line to `Vector3(0,0,x);`. The Cube should come right at us, then fly underneath us. The last slot is for how far/close it is from us.

I think of this as two parts. The new part is the set position command – the name `transform.position=`, what the three slots mean and where the edges of the screen are. The other part is what we've been doing – making `x` go up and down

Most cool-looking programs break down that way. Once you understand the rules for using some neat thing, the rest is regular programming to make the numbers change how they should.

4.7 more cute cube tricks

Before we knew how to move Cubes, we could make more than one variable change at once but we didn't have a reason. Now, with Cubes, we do.

If we change `x` and `y` at the same time, the Cube can move at an angle:

```
public float x=-8.0f; // just off the left side
public float y=-4.0f; // near the bottom

public float xSpeed=0.1f;
public float ySpeed=0.1f;

void Update() {
  x = x + xSpeed;
  y = y + ySpeed;

  transform.position = new Vector3(x, y, 0);
}
```

That should make the cube fly from the lower-left to the upper-right.

We can change the angle by changing either speed. Or have it go lower-right to upper-left (by making `x` go from 7 backwards.) And so on.

We already know how to make `x` go up, slow, then back down. We can reuse that for a Cube that goes right, slows, then goes back left:

```
public float x=-7.0f;
public float xSpeed=0.1f;

void Update() {
  x = x + xSpeed;
```

```
  xSpeed = xSpeed - 0.0002f; // <- new line

  transform.position = new Vector3(x, 0, 0);
}
```

It's the same idea, except in the old version we didn't care about the exact value where `x` turned around and started getting smaller. Now we have to play with the math and trial&error so it changes direction somewhere around 7. But otherwise it's the same idea.

We can use the "variables are better than constants" rule again, by turning that -0.0002 slowdown into a variable. I'm going to name the variable `xAccel`:

```
public float x=-7.0f;
public float xSpeed=0.1f;
public float xAccel=-0.0002f; // new. x acceleration variable

void Update() {
  x = x + xSpeed;
  xSpeed = xSpeed + xAccel; // using the new variable here

  transform.position = new Vector3(x, 0, 0);
}
```

Now we can just adjust the slowdown in the Inspector, instead of having to go back to the code each time.

A fun trick is to start `xSpeed` at 0, and `xAccel` at a very small positive number. The cube will speed up from a standstill. Or, set `xSpeed` negative and `xAccel` positive – it will go left, slow, then right.

4.7.1 Colors

If we want to change the Cube's color, we have to know the command, and what the values mean. This makes the Cube turn red:

```
GetComponent<Renderer>().material.color = new Color(1, 0, 0);
```

The first part is very strange-looking, but it works and changing color is fun enough to be worth it. The pop-up will even help us. Note the `R` in Renderer is capital, and those really are a less-than and greater-than (they're being used as angle-brackets.)

Like before, we need to know what the 3 values in parens mean. They stand for red, green and blue, in that order, and go from 0 to 1. For example, `Color(0,0.3f,0);` makes dark green.

Now we can use the same tricks to move the color. This slowly colors the Cube from black to red (if the Cube is off the screen, you may have to manually enter 0,0,0 for the position in Transform in the Inspector.):

```
public float r = 0;
public float rSpeed = 0.005f; // how fast it changes

void Update() {
  r = r + rSpeed;

  GetComponent<Renderer>().material.color = new Color(r,0,0);
}
```

We can try the usual tricks:

Start `r` at 1 and have `rSpeed` be negative (so it starts red and goes to black.) Use the second slot – `Color(0,r,0)` – to make green change, or `Color(0,0,r)` to change blue. Or use something besides 0's – `Color(1,r,1)` goes from purple to white.

Another interesting one is putting the variable in two places – `Color(r,r,0)`. The same variable controls red and green (which happens to make yellow.)

We can change red, green and blue at once by making a variable for each. These particular numbers change them at different speeds, making kind of an interesting pattern. You're probably sick of seeing the long way, so I'll start using the `+=` shortcut:

```
public float r=0.0f, g=1.0f, b=1.0f;
public float rSpeed = 0.002f, gSpeed = -0.002f, bSpeed=-0.001f;

void Update() {
  r += rSpeed; g += gSpeed; b += bSpeed;

  GetComponent<Renderer>().material.color = new Color(r, g, b);
}
```

4.7.2 Scale

To make us twice as big, use `transform.localScale = new Vector3(2,2,2);`. As usual, we need to know what the numbers mean.

You may have guessed from the `Vector3`, that they stand for x, y and z. 3D programs don't use just one number for size. They let you stretch it wider, taller and deeper (along the x-axis, y-axis and z-axis) in different amounts. If you want to just make the whole thing bigger or smaller, you have to use the same number for all 3.

Here are a few uses of it:

```
transform.localScale = new Vector3(1, 2, 1); // twice as tall
transform.localScale = new Vector3(1, 0.5f, 1); // half as tall
transform.localScale = new Vector3(2, 1, 1); // twice as wide
```

```
transform.localScale = new Vector3(5, 0.2f, 1); // long, sideways pole
transform.localScale = new Vector3(0.1f, 0.1f, 0.1f); // 1/10th size mini-Cube
```

The numbers aren't officially the size – they're a multiplier. But Cubes' base size is 1x1x1, so for Cubes, the scale is also the size.

This next program makes the Cube get wider and taller, in sort of a fun pattern. It starts not as wide, but it gets wider faster than it gets taller. I used `xSizeSpeed` for how fast the x-size changes since it was the best variable name I could think of:

```
float xSize=1.0f, ySize=2.0f; // our x and y scale
float xSizeSpeed = 0.05f, ySizeSpeed = 0.02f; // how fast scale changes

void Update() {
  xSize += xSizeSpeed; ySize += ySizeSpeed;

  transform.localScale = new Vector3( xSize, ySize, 1);
}
```

This would start taller, grow with time, but eventually get wider than it is tall.

As usual, we could enter a small negative number for one of the speeds, to see it get smaller. Or 0 for one speed to have that size stay the same. Negative sizes are allowed, but aren't good – it looks inside-out.

This, and the color change, and the movement are really the same thing - just some variables that get bigger or smaller, then one special Unity command saying what they control.

But that's the point. A lot of things that look different are mostly the same old lines playing with numbers.

4.8 Errors, funny stuff

When using the assign shortcuts it's easy to forget the `=` in `+=`, or to add an extra `=`, or an extra `x`. Examples:

```
x+3;  // oops! meant to write x+=3;
// Only assignment, call, increment, decrement, and new
// object expressions can be used as a statement

x ++ 2; // meant to use x+=2;
// Unexpected symbol '2'
```

Some mistakes aren't errors, they just do wrong stuff or look funny:

```
x += x+3; // add x+3 to x. Yikes!! Meant to write x+=3;

x = x+= 3; // Works! Adds 3 to x, but meant to use x+=3;

x = x ++; // Also works! Add 1. But meant to use x++;
```

I try to remember that `+=` and `++` are shortcuts. Using them should make a line shorter, not longer.

A script changing color needs to be on a real colorable shape, like a Cube. On a Light it gives the run-time error *"Missing Component Exception: there is no Renderer ...*. It's looking for the color slot and can't find it.

The scripts that move or change size won't give errors on a non-Cube, but you won't see anything happen.

Leaving out the `public` in front of a variable isn't an error. You won't see a slot for it in the Inspector, but it will work as normal. This prints 10, 11, 12 . . . :

```
int x; // forgot public, but not an error

void Start() { x=10; }

void Update() { x=x+1; print(x); }
```

And, to repeat, remember the Inspector wins. If you write `public int x;` an inspector slot with a 0 appears. If you later change it to `public int x=7;`, `x` will still start at 0 until you change it inside the Inspector.

Chapter 5

Using a 3D environment

This is an optional chapter about using the 3D area to make a little nicer set-up. I feel like knowing a little more makes the programming part easier to understand. And Unity3D uses most of the same concepts as any other 3D non-game program, so not a waste to learn even if you never plan to use Unity. And sometimes playing with the system can give you the idea for more mini-programming projects later.

But you can skip this section. Much later, when we bounce things around, We'll need some of this (a floor and walls to bounce from.) But not for a while.

5.1 Making a small room

5.1.1 Intro

Like many 3D programs, Unity starts with a completely empty game world – like outer space. The horizon you see is fake – there really isn't even a floor or ground. I think it's helpful to make a little room for our Cube – a floor and three walls, so maybe more like a diorama.

Doing that will involve learning just a little about how to use the controls, and what some of the numbers mean.

I'm going to explain the steps, but if you get lost or just feel like it, the Internet has a lot about simple Unity3D controls. I won't be offended if you stop and look something up.

5.1.2 A floor

We can make a floor using a stretched-out Cube:

Make another Cube (`GameObject->3DObject->Cube` again.) You might want to rename it "floor," to avoid confusion later (the usual way: click to select it, then click again.) To put it where a floor should be, go to its Inspector

position slot and enter (0,-5,0). You don't need to, but the TAB key jumps to the next – you can enter 0 for X, then tab to Y.

If you look at it in the big Game window, it should be lower-centered, but still just a small cube. To stretch it out, find Scale in the Inspector (two spaces below position.) A clever way is to move the mouse left of the X box (over the letter X.) It should turn into a double-ended arrow. That's an invisible slider for the x-scale. Left-click-drag will let us stretch the Cube wider. The same thing with Z will make it deeper, and more floor-like. Adjusting those can give it a nice floor-like look.

The Esc key will cancel a drag-slide (while you're holding the button down,) or the usual ctrl-Z will undo it. Or you can just type in numbers for X and Z, like we did for position. Drag-slide is just a nice short-cut.

The camera probably cuts off a lot of our floor. For fun, we can take a look at the whole thing using **Scene** view. The biggest window has tabs **Game** and **Scene**. Game is locked to what the game camera sees, while Scene lets us move around.

To move around: select the **Scene** tab. You should see a view of the floor from a different angle. Then double-click the floor in Hierarchy – the Scene window will zoom-center on the floor. For fun, you can double-click the Light or Camera or the other Cube and watch it zoom there.

If you select MainCamera, the Scene window will show a little white pyramid coming out of it – four lines shooting out from it at angles. Those show you the camera view – that's what we've been seeing while it runs, and what the Game tab always shows.

If you move the mouse over the Scene window, the scroll wheel moves you in/out. Right-click-drag spins around what you're aimed at (called orbiting.) The arrow keys will slide left/right/up/down (the Scene window has to be active. You may have to click on it.) Clicking the green/blue/red gizmo on the upper-right snaps to that angle.

Don't worry about getting lost. That's what double-clicking is for. It takes a while to get used to moving around in a 3D program. Scene's only purpose is to let you walk-around and adjust things, so don't worry about breaking it or putting it back the way it was.

A fun trick with Scene view is you can use it while running the program. If the Cube goes off the edge, you can switch to the Scene tab, scroll-zoom out, and watch it keep going. You can even double-click the moving Cube to follow it.

5.1.3 walls

We can make the walls the same way as the floor: make another Cube, move it where the wall should go, and scale it.

The moving cube is going from x= -7 to 7, so that's where the side walls should go. We can make a Cube, then hand-enter (-7,0,0) for Position, then slide on Scale Y and Z to grow it tall and deep.

Doing that in Scene view will let us get a good angle on it. But doing it in Game view will let us see the real way it will look. I usually go back and forth.

For the wall on the right, we can use Copy/Paste on the first wall, and change position-X from -7 to 7.

The back wall is just for looks, so not as important, but fun to make. We can start its Cube at maybe `(0,0,-6)`. The idea is, we know our moving Cube goes through 000. So z at -6 will put the back wall 6 units behind that, which seems far enough. Then we can slide-stretch X and Y to make it wall-sized.

For a test, enter (-7,0,0) for the position of the moving Cube. That might move it partway *inside* the left wall. Gah. Maybe that's fine (the computer won't care about one thing inside another.) Or maybe the walls should move a little further apart.

Just so you know, the Cube is 1x1x1 and centered. When its at -7, it's left edge is at -7.5.

After, if you didn't do it already, go to Scene view and look around.

Notes:

- I put the floor at -5. Why not 0? The system doesn't have any special meaning for particular numbers, so a floor can be anywhere. The moving Cube we have now stays at a height of 0, so -5 seems like a good place for a floor.

- The size of the window can also be anything. We could position the camera and resize it so x goes from exactly 0 to 100 across the screen. But that's a pain: the camera starts at -7 to 7, and those numbers are as good as any others.

- The axis also depend on where the camera is aimed. Our camera is looking from the front, so x is sideways and y is up. A topdown camera, looking down at the floor, would still have x going sideways, but z would be up and down.

- In the upper-right corner of Scene view, you should see a little XYZ gadget, showing the view angle. The three colored ends point in the positive direction of that axis. You can also click on any of those cones, and it should snap views (it stays centered on the item you double-clicked, so it's kind of neat.)

5.2 Color, Textures

Coloring the shapes seems frivolous. But in my test scenes I've found that colored stuff really helps me see what's happening. Plus it looks really neat, and might give you some ideas for things to program.

The standard way color and pictures are placed on 3D objects is using one extra thing, called a **Material**. If you want a red cube, you make a Material, color it red, and place that Material on the cube.

Having to make a Material first seems like a pain. One reason for it is, you can set the color once, then use that Material on all the objects you want. The other is there are also lots of settings, and a Material is a good place to store them. For example a picture of a cat can be doubled in one Material, set to a green tint in a second, and normal in another.

Here are the steps to turn the floor red:

- In the **Project** panel select `Create->Material`.
- Enter a name for it. You should have a little ball icon with "New Material" written, waiting for you to enter a better name. You can easily rename it later, so anything is fine.
- The Material should appear in the Inspector. If it doesn't, select it again. The top item should say **Albedo**. That's a fancy word for "the basic color." Click the color picker and find one you like.
  If you've never used one before, the *top* bar is the color. The long right-side bar and the big middle square are used to adjust it.
- Drag the material into the Scene. As it goes over each object, you should see it change color. Drop it on the one you like.
  Just so you know, you can also drag the Material to the name, in the Hierarchy panel. The real spot, if you need to check later, is in the Inspector for the item inside the `MeshRenderer` section. Look where it says `Materials`. Pop that open if needed, and look in `Element 0`. That's the real place Materials go (don't worry about element 1 or 2. Only specially made objects ever have more than one.)

Using a few Materials we can now turn the ball yellow, or make the back wall blue. The ball should stand out more. Plus it's fun.

Some notes: notice how the color-picker, in the lower-right, shows red, green blue numbers as 0-255. That's the other standard color range. From last chapter we know Unity prefers 0-1. The color picker secretly converts the ranges.

The line changing color should make a little more sense now. The one: `GetComponent<Renderer>().material.color=`. The color is stored in a material (we still don't know what a Renderer or a Component is, but 2 out of 4 is

pretty good.)

A fun thing is to add a tiled picture (we call pictures *textures*) to the floor. My floor is 14 wide and 7 deep, so, I'll get a picture to repeat that much. The steps are:

- Find any picture in your computer. Try to get one that you can tell when it repeats (so, not a solid color.) Drag it into the Project panel in Unity. It will be copied.
- Have one Material already created and on the floor.
- Select that Material and go to Albedo again. We're going to put the picture here. One way is to drag the picture into the square. Another is to click on the small circle in a circle. That's the "show me my choices" icon. It should open a window that lets you pick an image.
- To make it tile, still on the Material, look down for *Tiling*. There are two – make sure it's the first one, under "Main Maps." Enter 14 for X and 7 for Y. You should see the image repeat that many times.

This can be nice for measuring. If your object is exactly 12 wide, and a picture tiles 12 across, then each picture is exactly one unit.

One thing to watch for, you can reuse a material for several Cubes and they *share* it. If your floor material is also on something else, it will also get the picture. If that's a problem you can use Edit->Copy on a Material.

5.3 Adjusting the Camera

For fun, we can move and re-aim the camera and change the "perspectiveness" of it. Sometimes a better view can make testing easier.

Suppose we want to move the Cube from X=0 to 10, which means we need to adjust the camera to see that range. As a guide we could change the floor to run 0 to 10: make it 10 units wide (just type a 10 into the x-scale) and change the X in position to 5.

Or, an even better guide, we could create and place dummy cubes on each side: x=0 and another at x=10. Together with the new floor, that should give a pretty good indicator for when the camera is aimed correctly.

Obviously the camera should be at x=5. Since it uses perspective we can slide the camera in and out (using it's Z position) until the edges are correct.

We can change the edges of the camera another way. The amount of perspective is controlled by FieldOfView, in the Camera section of the Camera's Inspector. It's starts at 60, for no reason. We can slide that around to adjust the edges. If you want a mostly flat perspective, pull the camera way back and

narrow FieldOfView.

Another fun thing is to angle the camera down a little. You can click-slide any of the camera's rotations. Y pans (looks sideways,) Z cocks it, and X angles up/down.

You can eventually get a nice downward view by going between angling x-rotation down a bit and sliding up the y-position. Putting the Cube at 000, and those dummy cubes on the sides, can help you get it right.

Again, none of this really matters. It's just nice knowing where x= -7 to 7 came from, or why changing z position just changes size a little. And sometimes it helps with testing to be able to make some landmarks, like walls or things at some exact position.

Chapter 6

If statements

So far, our programs can do math, but they can't really "think." They should be able to look at the variables and somehow decide what to do next. Using one very simple new rule, we can add the ability to make all sort of complex decisions.

I'll start out by just showing the first half of the rule, then the mechanics of how it works, then the second half. After that, we can program a crazy amount of stuff, just by being clever.

The basic decision statement is an **if**. Here's a sample `if` statement:

```
public int x; // input

void Start() {
  print( "A" );

  // the interesting part:
  if( x < 10 ) {
    print("x is small");
    print("It's less than 10");
  }

  print("B");
}
```

`x` is the input, using the usual trick – change `x` in the Inspector, then run. The middle chunk of Start is a single `if` statement. What it does is probably obvious: it only prints the middle two lines if you entered 9 or less for `x`.

If you just looked at that and guessed the rules, you'd probably get most of them right. But, just to be complete, here they are:

- The syntax (the things that just need to be there) of an `if` are: the word `if`, in lower-case; the `()`'s afterwards; the {}'s after that.

A basic if looks like:

```
if( trueOrFalse ) { stuffToDo }
```

The parens are like the required ones in a print statement. They aren't math parens. They just have to be there as part of the `if`.

- There's no semi-colon at the end. That's a special rule, since the close-curly brace already marks the end.
- The part in ()'s is a true/false value – the test. Rules for what you can write, mostly stolen from math, are later. `x<10`, in the example, does exactly what it looks like – true when `x` is less than 10.
- The {} curly-braces belong to the `if` (called the body.) Inside go normal program statements.

  This is a new use of {}'s, but not really. The old {}'s group statements together for Start or Update. This new version also groups statements, just for a different reason.

  Notice how each print still has it's own semicolon at the end. These aren't special `if` semicolons – they're just regular statement semicolons, for the prints.
- The semantics (what it does) are: check the test for true or false. If true, run the body (same as usual: run statements top to bottom.) False means to skip the entire body – jump to just after the ending curly-brace.
- The way we indent the body a little is just style. Like anything else, an `if` can have spaces and line-breaks however you like. But it looks nicer this way – it's easier to see that those two lines are inside of an `if`.
- The curly-braces are optional if the body has only one line (more later.)

Just in case, a run through of the example: the program always prints `A`, then it might print the things in the `if`, then it always prints `B`.

If you enter 14 for `x`, the program only prints `A` then `B`. The `if` is false, so it skips the lines in the middle. If you entered 6, it prints `A`, the `if` is true so it prints `small` and `it's less than 10`, then it prints `B`.

The program will always print one of those two ways: either only A then B; or else all four lines.

6.1 Comparison Operators

Less than, <, is the same as regular math. So is greater than, >. But the rest of the math symbols we use aren't written on our keyboard, so need rules for how to write them. Here are all of the compare symbols, and some rules/comments:

Computer math compare symbols are:

```
<  less than
>  greater than
<= less than or equal to
>= greater than or equal to
== equals
!= not equals
```

String compare symbols are the last two: equals and not-equals.

Less-than and greater look like regular math:

```
if(x<0) { print( "x is negative"); }
if(x>0) { print( "x is positive"); }
```

The last four take some explaining. In real math, we write "or equal to" by adding an extra line underneath the symbol. But we can't type that, so we add an equals after it. `<=` counts as one symbol – no spaces, must be in that order.

Flipping them to make `=<` is an error. I remember equals comes last because in school I learned to say it in that order: "less than or equal to". Ex's:

```
if( x <= -1 ) { print( "x is negative"); }
if( x >= 1 ) { print( "x is positive"); }
```

The second-to-last `==` is the compare-equals symbol. Together both =’s count as one symbol. The reason for this is that single = is already an assignment statement. We don't want `=` to mean different things in different places, so we invented a special double-equals to mean compare-equals. Ex's:

```
if(x==5) { print("you have five"); }
if(w=="cat") { print( "meiow" ); }
```

The compare for string `==` is case-sensitive. If `w` is `"cat"` then `if(w=="CAT")` is false.

Finally, `!=` is "not-equals". In regular math, we put a slash through an equals, but we can't type that. We could move it in front, like `/=`, but we used that up in the special divide-and-assign rule. It turns out, real mathematicians have been using an explanation point to mean **not** for a long time (they call it a bang,) so we stole it from them.

Like `<=`, `!=` counts as one symbol, and must be in that order. The way I remember is `!` means not, so `!=` means not equal. Exs:

```
if(x!=0) { print( "not zero" ); }
if(w!="abc123") { print( "password incorrect" ); }
```

6.2 Some semi-useful ifs

There are a few ways to think of an `if`, and a few tricks we can do with them. One thing `if`'s can do is keep numbers in range:

```
// x can't be more than 10:
if(x>10) { x=10; }
```

This uses the rule that any normal statements can be inside of an `if`. Assignment statements are normal statements, so they can be in the body.

We can grow that idea to make sure `x` is between 1 and 10:

```
// make sure x is 1-10:
if(x<1) { x=1; } // fix too small
if(x>10) { x=10; } // fix too big
```

In this case, we can think of the `if` as detecting and fixing bad things. True means the `if` is unhappy. False means we "passed" and don't need to do anything.

The same idea can be used fix bad strings:

```
// fix common spelling errors
if(w=="teh") { w="the"; }
if(w=="thier") { w="their"; }

// if they leave the name blank, use jane:
if(name=="") { name="jane"; }
```

Notice how this mixes == and =. They can be easy to confuse. Just remember that == means compare and = means change. It works out the == goes inside the parens, since that's where you do the comparing.

I want to add a note here: `if`'s follow the same "do it and move on" rules as everything else. They aren't sticky. For example, this code ends with `x` as 11, even though it has a "can't go past 10" `if` in it:

```
int x=17;
if(x>10) { x=10; } // fixed to not be past 10
x=x+1; // it's 11. The if was a 1-time thing
```

We can also use an `if` to make tweaks. The way it works is the same as fixing bad values, but it feels like a different thing. In this example, `"Beth"` and `"Dick"` aren't bad, but we prefer the long versions:

```
// undo common nicknames:
if(name=="Beth") { name="Elisabeth"; }
if(name=="Dick") { name="Richard"; }
```

`if`'s can be used to make table lookups. For example numbers to number words. I'm using the shortcut where we can leave out the body curly-braces if we only have one line:

```
// get words for numbers:
string numWord = "unknown num";
if(num==1) numWord="one";
if(num==2) numWord="two";
if(num==3) numWord="three";
if(num==4) numWord="four";
if(num>=5) numWord="many"; // <- everything else
```

We can use this trick for any made-up tables, like hurricane names:

```
// get hurricane names:
if(num==1) hName="Ana";
if(num==2) hName="Bill";
if(num==3) hName="Claudette";
 ...
```

Or, this is a completely made-up table of monsters and weapons they use, which looks up on strings:

```
public string monster; // enter this

  string weapon = "";
  if(monster=="pirate") weapon="rusty cutlass";
  if(monster=="ninja") weapon="poisoned shuriken";
  if(monster=="alien") weapon="zap ray";
```

Then there are some common miscellaneous uses of an `if`:

If we have a word, like `duck`, and a count, we can add an "s" for plurals:

```
if( howMany != 1 ) animal=animal+"s"; // ex: duck becomes ducks
print(howMany+" "+animal); // 1 duck, or 4 ducks
```

The last print is a little interesting. In something like `"The "+w` we sneak the space into the end of `The`. In `howMany+animal` there's nowhere to sneak in that space, but we can still add it using `+" "+`.

We can "take off the minus sign" from a number by flipping negatives to positive:

```
if(x<0) x=x*-1; // make negative #'s positive
```

This turns -6 into 6, but leaves +4 alone. `-x` is a shortcut for `-1*x`, so `x=-x;` also would have worked.

We sometimes use `if`s as a guard. In our minds, this adds 1 to `x`, but not if it would bring us past 10:

```
// add 1 to the score, but not if it would go past 10:
if(x<10) { x=x+1; }
```

We can also add first, then fix it later:

```
x=x+1;
if(x>10) x=10;
```

In real life, you wouldn't want to add 11 gallons to a 10-gallon bucket, then take out a gallon. But in computers it's often easier to go past, then fix it.

For fun, we can use both ways to add 3 to the score, but not past 10. Here's the method where we check before adding:

```
// add 3, not past 10:
if(x<10) x+=3;
```

There's a problem. This will add 3 to 8, giving 11; or 3 to 9, giving 12. We could fix that with `if(x<8)`, but now it won't add anything to 8 or 9. Gah! We can do it correctly by using several `if`s:

```
// add 3, not past 10, 2nd try:
// handle overshoot cases. They go straight to 10:
if(x==8) x=10;
if(x==9) x=10;
// now handle non-overshoot:
if(x<=7) x+=3;
```

That's a bit long. The "fix afterwards" method works with no changes:

```
x+=3;
if(x>10) x=10; // 7,8,9 all become 10
```

Moving on, we can use a pair of `if`s to find the largest of two variables:

```
// find the largest:
if(n1>=n2) largest=n1;
if(n2>=n1) largest=n2;
```

The two `if`'s work together – one of them has to be true.

An interesting thing is that this gives the largest value, but doesn't tell us which variable it came from. Usually it won't matter – just knowing the number is good enough. Like if we order pizza and meatballs. Maybe the pizza takes 10 minutes and the meatballs take 15 – they tell us it will be a 15 minute wait, and that's all we want to know most of the time.

Another interesting thing is if the numbers are equal, both `if`'s are true, and that's fine.

6.3 Compare precedence

At some point, you might put some math inside of an `if`, and it might start to look funny. For examples:

```
if( x>y+1 ) ...
if( x + y<10 ) ...
```

The rule is: math goes first, compares go last. The official rule is that compares count as math, but all have lower **precedence** than plus, minus, multiply and divide.

This is the rule you want. It means you almost never have to put extra grouping parens inside of an `if`. But, same as before, you can if you want:

```
if( x > (y+1) ) ...
if( (x+y) < 10 ) ...
```

As with `print`, the outer parens are required, and the inner ones are just math parens.

As usual, you can still use parens to change the order of `+` and `*`. This adds 1 to `a` first:

```
if( (a+1)*b < 10 ) ...
```

This is really just the rule that things work the same, no matter where they are in the program. Any place you have math, you can use all the regular math rules.

6.4 else

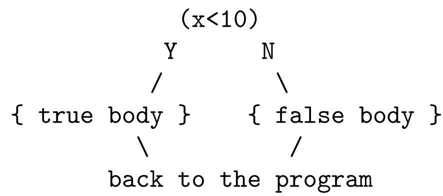
A regular `if` either runs the lines, or does nothing. A little different version can choose between two things. It adds the word **else**, and is usually called an `if-else` statement. An example:

```
if(x<10) {
  print( "less than 10" );
}
else {
  print( "10 or more" );
}
```

It's just a regular `if`, with `else` { `doThisWhenFalse` } jammed at the end. Notice how there are still no special `if` semicolons. All six lines count as one giant if-else statement.

Also notice there isn't a test on the else. It's just `else`, then a curly brace. We usually think of it as two bodies with a test to pick one.

Sometimes we call them **branches**. Here's a picture of how we sometimes think about it flowing through an if-else, picking a branch:



You can also think of an if-else as a shortcut for two “opposite” `ifs` in a row. Here are two ways to write small or big. The if-else version is shorter, and it’s easier to see what it does:

```
// 2 opposite ifs version:
if(n<100) { w="small"; }
if(n>=100) { w="big"; }

// else version:
if(n<100) { w="small"; }
else { w="big"; }
```

Here’s “find the largest” both ways. I think the if-else version makes it more obvious we’re picking between `n1` and `n2`:

```
// 2 opposite ifs largest:
if(n1>n2) largest=n1;
if(n2>=n1) largest=n2;

// if-else:
if(n1>n2) largest=n1;
else largest=n2;
```

Choosing between two equally good options is one way to use an if-else. Another way is just to tack-on a little extra. Two old examples with an extra `else` added:

```
if(x<10) x=x+1;
else { print("sorry - can’t add past 10"); }

if(password != "abc123") print("bad password");
else print("password accepted");
```

More `else` notes:

- There’s no test in the else. I wrote this before, but it’s a common mistake. Besides not needing it, putting one there is an error. Ex. of an extra else-test:

  ```
  // common error putting a test with the else:
  string sound;
  if(w=="cat") { sound="meio"; }
  else (w!="cat") { sound="unknown noise"; } // this is wrong!!
  ```

- The `else` body is usually indented the same as the true body. This goes with the idea that true and false are just two branches of the same `if`. Here's a silly example where both have two lines and are indented the official way:

  ```
  if(x>=0) {
    print("positive");
    n=+1;
  }
  else {
    print("negative");
    n=-1;
  }
  ```

- The leave-out-{}s-for-one-line rule can also be used for the `else`. Examples:

  ```
  // both can be 1-line and skip {}:
  if(x<100) w="too small";
  else w="too big";

  // 1-line true skips, else doesn't:
  if(z==0) z=1;
  else {
    print("you picked a good z");
    print("thanks");
  }

  // true uses {}, else skips:
  if(w=="cat") {
    print("changed it to kitty for you");
    w="kitty";
  }
  else print("not changing it");
  ```

- An `if-else` can flip the test and flip the true and false parts. Sometimes that makes it look nicer. These are the same:

  ```
  if(password != "abc123") print("bad password");
  else print("password accepted");

  if(password == "abc123") print("password accepted");
  else print("bad password");
  ```

  Both versions only accept `abc123`.

This comes in handy when you're having trouble writing the test – maybe the opposite way is easier. Another example:

```
if(n<100) { w="small"; }
else { w="big"; }

if(n>=100) { w="big"; }
else { w="small"; }
```

The second version might look nicer if we "like" big numbers.

- It's possible to put "nothing" after an `if` by writing {} (an empty body.) Here's a strange "don't let `x` get past 10":

  ```
  if(x<=10) {} // everything is fine, don't do anything
  else { x=10; } // too big, lower it
  ```

  The `else` can have an empty body, which is the same as not having it at all:

  ```
  if(x>10) { x=10; }
  else {} // oops! don't need this. But legal
  ```

  I do it rarely, but sometimes empty bodies work as placeholders:

  ```
  if(animal=="dog") print("arf");
  if(animal=="elephant") {} // nothing, yet
  ```

6.5 And, Or

The test is allowed to use combiners *and* and *or*. This `if` is true for yellow dogs:

```
if(animal=="dog" && color=="yellow")
  print("You should name your yellow dog Sparky.");
```

I'm just going to write all the rules, then examples for them later:

- The symbol for *and* is `&&`. The symbol for *or* is `||`. Both are two of the same thing in a row, no spaces, and count as one symbol. The *or* bar is the line above the backslash on most keyboards.
- They go between legal if-compares (things that can be in an `if` by themselves.) In the example above, `animal=="dog"`, and `color=="yellow"` could both be in regular `if`s.

- `&&` means they both have to be true. `||` means either can be true, or both.

  The "or both" rule seems like it might be confusing, but it's not once you start using it for real things. A computer `or` is always something like "13 years old or at least 4 feet tall to go on this ride." Being both is clearly fine.

- You can use `&&` or `||` multiple times. It means they all have to be true, or at least one has to be true (examples below.)

- If you mix `&&` and `||` in a big compare, `&&` has higher precedence. You can think of `&&` being like multiply, and `||` like plus. You can use parens to group. This rule can be pretty tricky. We won't use it right away. I just wanted to get all the rules in one place.

- `&&` and `||` have lower precedence than compares do. In other words, they work the way you think. It does all the math, then all the compares, then the and/or checks. You don't have to add parens, but you can.

Suppose we want to know if `animal` is a cat or a dog. We have to write a legal check for a cat, then a legal check for a dog, and combine them:

```
if(animal=="cat" || animal=="dog")  print("common housepet");
```

You can think of this as the "each has to be legal" rule, or as the "and/or always goes last" rule. Suppose we wrote `if(animal=="cat" || "dog")`. The computer sees 2 things joined by an `or`, and the second one is `if("dog")`, which makes no sense (it gives a compile error.)

Suppose we want to check for discount tickets (very old or very young.) We have to write each part out, like this:

```
if(age<=12 || age>=65)  print("discount tickets");
```

If we tried the shortcut `if(age<=12 || >=65)` the computer would see `if(>=65)` for the second part (obviously an error.)

If we need `x` and `y` to both be 0 or more, we also have to write it the long way:

```
if(x>=0 && y>=0) print("both 0 or more");
```

The shortcut `if(x && y>=0)` would see `if(x)` for the first part (also a compile error.)

A good thing about each part being separate is you can mix and match any kind of variables and tests. For example, a string and int check:

```
// young cats are kittens:
if(animal=="cat" && age<=3)  print("kitten");
```

The way the computer sees it, `animal=="cat"` is true or false. Then `age<=3` is true or false. By the time the `&&` happens, everything is down to only true's and false's.

With string comparing, it's fine to mix `==` for one and `!=` for the other:

```
// non-brown cows are fancy:
if(animal=="cow" && color!="brown") print("fancy cow");
```

The multiple compare rule says we can check for baby fancy cows by adding one more:

```
if(animal=="cow" && color!="brown" && age<=3) print("young fancy cow");
```

This is a little bit like `3+7+4`. There's no special 3-way rule. Technically you're adding a pair at a time: `(3+7)+4`. Also technically, the computer *and*'s the animal and color first. But it works out the same either way - they all have to be true.

It can be helpful to look at some broken code. These are legal, but pointless. Just say them out loud and it's obvious way:

```
if(age>=18 && age>=21)  print("same as if(age>=21)");
if(age>=18 || age>=21)  print("same as if(age>=18)");
```

For the second one, `age` of 21 or more seems like it might be "double-true," but there's no such thing. Everything inside of an `if` always comes out to be just regular true or false, no matter how complicated it is.

We can make some other wrong example `if`s that are always false. A number can't be less than 5 and more than 10 at the same time, and a word can't be `"cow"` and `"horse"` at the same time:

```
if(num<5 && num>10)  print("always false");
if(animal=="cow" && animal=="horse") print("always false");
```

Here are some useless always true `ifs` using an *or*. They're always either true or "double true":

```
if(num>7 || num<15)  print("always true");
if(animal!="cow" || animal!="horse") print("always true");
```

In the first, all small numbers are less than 15, all larger numbers are greater than 7, and 8-14 are "double true" (which is the same as normal true). In the second, cow is true since it's not horse, horse is true since it's not cow, and everything else is true since it's not cow or horse.

This next example shows how precedence normally does the right thing (the math is made-up. Just think about the grouping):

```
if( x + 3 < y || 3 < x || y == 9 ) ...
// how the computer sees it:
if( ((x+3) < y) || (3<x) || (y==9) ) ...
```

It looks a little like the computer might try `y || 3`, or maybe `x || y`, but it knows *and*'s and *or*'s go last.

6.5.1 Useful and/or's

Now onto tricks and examples people really use.

We check "if it's one of these" by using `==`'s chained together by *or*'s:

```
if(n==2 || n==4 || n==6 || n==8)
  print("single digit even");

if(w=="bill" || w=="willy" || w=="billy")
  print("william");
```

It might seem like there would be a shortcut where we didn't have to write `n==` or `w==` over and over. But there isn't.

Range-checking is common. In math, we'd write 1 to 10 as: $1 \leq n \leq 10$. That's really a shorthand for two compares. In a program, we have to write them out:

```
if(n>=1 && n<=10) print("n between 1 and 10");
```

A way to think of this is it can't be too small (`n>=1`) and can't be too big (`n<=10`). Both parts have to be true, so it's an *and*. It we used an *or* by mistake, the whole thing would always be true (everything is either bigger than 1 or less than 10. Some numbers are both.)

We can do some cute things with a range check. Say we want to check how many digits a number has. There isn't a special way, but 0-9 is 1-digit, 10-99 is two digits, and so on (pretend we know the number isn't too big):

```
// how many digits does n have:
int digits=0;
if(n>=0 && n<=9) digits=1;
if(n>=10 && n<=99) digits=2;
if(n>=100 && n<=999) digits=3;
```

We can mix ranges and exact numbers. Suppose we win if we get 7 or 10-15 or 22. That's like checking for three exact numbers, using `||`'s. Except the middle one is a range:

```
if( n==7 || (n>=10 && n<=15) || n==22 ) { print("You win"); }
```

Suppose instead we win on 5-9 or 18-21. That's "this range or that range":

```
if( (n>=5 && n<=9) || (n>=18 && n<=21) ) { print("you win differently"); }
```

Mixed and's/or's can be confusing. Sometimes you can just see what they do. Other times you have to trace them. Convert each part to T/F, then combine them a little at a time:

```
// suppose n is 9
if( n==7 || (n>=10 && n<=15) || n==22) { print("You win"); }
1)   F   || (  F   &&   T  ) ||   F
2)   F   ||        F         ||   F
3)               F
```

Often we want to check whether two `float`s are within 0.01 of each other. If we're sneaky, we can find the difference, then use a range-check to see it that's small:

```
float diff = n2-n1;
if(diff>-0.01f && diff<0.01f) print("close enough");
```

But it's fun to try to write it directly. We can do a range check comparing `n1` to "a little less than `n2`" and "a little more than `n2`":

```
if( (n1 > n2-0.01f) && (n1 < n2+0.01f) ) print("close enough");
```

Checking not-in-range isn't as common. It looks like a range-check, but it has to be too low *or* too high:

```
if( n<1 || n>10 ) print("n is NOT 1-10");
```

If you use an *and* by mistake it will always be false (nothing can be less than 1 and more than 10 at the same time.)

6.6 Nested ifs

`if` and `else` bodies can have more `if`s inside. We usually call them **nested** `if`'s (after those nested Russian dolls.) We don't need any new rules for this, but we do need practice thinking about how to use this trick.

Suppose we have a truck fee: 1000 pounds and under is free, up to 5000 pounds is $10, and past that is $20. This nested `if` prints the fee:

```
int fee=0;
if(wt<=1000) { fee=0; print("No fee"); }
else {
  if(wt<=5000) fee=10;
  else fee=20;

  print("Truck fee is $" + fee );
}
```

I think this seems pretty natural. The first `if` divides it into free and pay. If it's free, we print that and are done. Inside the pay part, we need one more `if` to check the cost. Notice how the inside `if` is indented to look nice in the `else` body.

A trick is to check it by "exercising" each part. Run it in your head with a number from each category, like 500, 1500 and 7000, which should give every possible result, and make sure it does.

Here's a longer one that finds the prices for animals. Most animals cost 3, except for cats and dogs, and red dogs and white and black cats are exceptions:

```
int cost=3; // "other" animals all cost 3

if(animal=="dog") {
  cost=8; // most dogs are 8
  if(color=="red") cost=16; // red dogs are rare
}

if(animal=="cat") {
  cost=10; // most cats are 10
  if(color=="white") {
     cost=14; // white cats are valuable
  }
  if(color=="black")
    cost=5; // no one likes black cats
}
print("Cost of " + color + " " + animal + " is $" + cost);
```

Hopefully this reads pretty well. The dog `if` says dogs cost 8, then does one extra check for red dogs. Notice how it only checks the color – we already know it's a dog. The cat `if` is the same, but longer (and I changed-up the paren style for fun.)

If you remember from school, the x/y graph is broken into quadrants around (0,0) (like the picture below.) To find the quadrant of an (x,y), we could brute force with four lines like `if(x>=0 && y>=0) quadrant=1;`.

But a nicer way is to make a plan: divide it into top/bottom half, and go from there:

```
// Standard quadrants (0,0) in the middle:
// 2 | 1
// -----
// 3 | 4

if(y>=0) { // top half
  if(x>=0) quadrant=1;
```

```
  else quadrant=2;
}
else { // bottom
  if(x>=0) quadrant=4;
  else quadrant=3;
}
```

It could also be written starting with a left/right check.

We can make several plans to find the largest of three numbers. This one checks whether A is the largest. If not, it must be B or C:

```
// find largest of a,b,c
if(a>b && a>c) largest=a;
else {
  // it wasn't A, so check B and C:
  if(b>c) largest=b;
  else largest=c;
}
```

A different plan, trickier than the first way, starts by comparing A and B. The smaller one can't be the answer. Compare the larger one to C:

```
// find largest of a,b,c (version 2)
if(a<b) {
  // A is too small, must be B or C:
  if(b>c) largest=b;
  else largest=c;
}
else {
  // B is too small. Must be A or C:
  if(a>c) largest=a;
  else largest=c;
}
```

A simple way to test is using every combination of 1,2,3 (there are 6 of them.) Not that it matters, but this way is a little faster. It always uses two compares. The first way sometimes uses three compares.

This next one is simpler. It checks for positive, negative or 0. I thought it made sense to "knock out" 0 first, then check for +/-:

```
if(x==0) print("zero");
else {
  if(x>0) print("positive");
  else print("negative");
}
```

6.7 Common errors

- It's very easy to forget/mistype one of the =’s in a compare:

  ```
  if(n=3) { print("perfect number of cats"); } // ERROR
  // Cannot implicitly convert type int to bool.
  ```

  The error is the computer way of saying that assigning `n=3` isn't a true/false. But even if you didn't know that, double-clicking will give you the line. When you look it over, you'll spot the bad single-equals eventually.

- It's easy to become "`==`-happy" and accidentally use them for assignment:

  ```
  x==5; // ERROR. Wanted x=5;
  // Only assignment, call, increment, ... can be used as a statement.

  if(animal=="cat") // this == is correct
    color=="red"; // ERROR (same message) Wanted =
  ```

  It's the general purpose error, but at least it gives you the line. The computer is really telling you that comparing `x` to 5, just by itself, is useless, so you must not have meant to do it.

- Adding an extra semi-colon in the middle makes a very hard-to-find non-error error. This will always print `"n is positive"`, no matter what `n` is:

  ```
  if(n>0); // <- oh no!!
    {
      print("n is positive");
    }
  ```

  The semicolon legally ends the `if`. There's no error, but there's also nothing inside the `if`. Here's how the computer sees it:

  ```
  if(n>0) {} // empty body. 100% useless if
  print("n is positive"); // out of the if, runs every time
  ```

  This can cause hours of frustration if you don't know to look for that extra semi-colon.

  Putting an extra semi-colon after the {} isn't a problem. Don't do it on purpose, but it's fine if you do:

  ```
  if(n>0) { print("moo"); }; // useless ending, but runs the same
  ```

- The rule where you can leave out the {}'s for one statement can cause some serious problems if you aren't careful. What happens is, you start with this, which is fine:

  ```
  if(n>10)
    n=10;
  ```

  But then you add another line to the body, which causes a problem:

  ```
  if(n>10)
    print("too big");
    n=10;
  ```

  The indenting makes it look like both are in the body (and that's what you meant.) But the missing{} rule takes only 1 line. The computer reads it like this:

  ```
  if(n>10)
    { print("too big"); }
  n=10; // uh-oh. out of the if. n is always 10!
  ```

  Non-error errors caused this way can also cause really frustrating, hard-to-find wrong code. Here's a version of the same thing, on one line:

  ```
  if(a=="cow") sound="moo"; eats="grass"; // Not what it looks like
  ```

  In our minds, the whole line is the `if`. But the short-cut is only for one statement. The computer sees this:

  ```
  if(a=="cow") { sound="moo"; }
  eats="grass"; // always does this line
  ```

  We'll drive ourselves crazy trying to figure out why the stupid computer thinks tigers and bears also eat grass. Of course, writing it on one line with curly-braces is fine:

  ```
  if(a=="cow") { sound="moo"; eats="grass"; }
  ```

  Some people think it's just easier to always put {}s, even for very short bodies. That way you can never have this sort of error. That's partly why some editors always add {}'s after you type the `if`.

- When comparing the same thing several times, it can be tempting to leave out the variable, like this:

```
if(n==3 || 6 || 8) // ERROR
// Operator '||' cannot be applied to operands of type 'bool' and 'int'

if(animal=="goat" || "pig" || "cow") // same error
```

There's no shortcut like this. You have to write out each part.

- Nested if-else-ifs and missing {}'s can trick you into not knowing which `if` goes with the `else`. Suppose you have this:

```
// runs wrong:
if(n<10)
  if(n>0) print("between 1 and 9");
else
  print("big");
```

Indenting makes it look like the `else` goes with the first `if`. But it really goes with the second one. The computer sees this:

```
if(n<10)
  if(n>0) print("between 1 and 9");
  else print("big");
```

It will print `big` for negative numbers and do nothing for ten or more.

Writing out the curly-braces can fix these problems (another reason some people always write them):

```
if(x<10) {
  if(n>0) print("between 1 and 9");
  // an else for the inner if would need to be here
}
else
  print("big");
```

Chapter 7

more if tricks

Now that we have ifs, there are lots of interesting problems we can solve, and tricks we can do. This section won't have any new rules, just lots of examples.

7.1 falling through

Sometimes you need a few `ifs` to try to compute a value. One trick is, pick one of the values to start with. If all of the ifs are false, we say it "falls through" and keeps that starting value. It's like having a "none of the above" (except it's none-of-the-below.)

This is just a cutesy, not-really-useful way to rewrite an if-else, but it shows how the trick works:

```
w="big";
if(n<100) w="small";
```

Here's the same trick to to make a "which is larger" `if`:

```
// alternate way to find largest
largest=n1;
if(n2>n1) largest=n2;
```

A more interesting example is to rewrite finding positive, negative, or 0. In this next example, `"zero"` is the fall-through:

```
string w="zero";
if(n>0) w="positive";
if(n<0) w="negative";
```

Then, here's a made-up example, where most insurance costs $100, but helicopters and canoes are different:

```
float cost = 100;
if( vehicle == "helicopter" ) cost=250;
if( vehicle == "canoe") cost=85;
```

A trick to these is to think of it as a *process*. We're not saying that `cost` is definitely 100 – we're just starting it off as 100. The other trick is that `int`s aren't like mashed potatoes. Adding or subtracting to a pile of mashed potatoes makes a mess, but we can easily adjust `int`s all we want.

7.1.1 Cascading range-slicing ifs

We often want to program a lookup table with ranges and cut-offs. For example the standard grade 90=A, 80=B, 70=C rules. An obvious, awkward way is to hand-check each range, using this ugly, too-long code:

```
// bad, ugly grade-checking code:
if(score>=90) grade="A";
if(score>=80 && score<90) grade="B";
if(score>=70 && score<80) grade="C";
if(score>=60 && score<70) grade="D";
if(score<60) grade="F";
```

Notice how each breakpoint is repeated twice. 70 is in the 3rd `if` and 4th `if` (70-80 and 60-70.) And I had to be careful with `>` and `>=`. It would be easy to miss a number (like using `>80` and `<80` in both places, so we never give a grade to 80.)

A nicer way is using `if-else`'s and nested `if`s. This works, and follows the by-the-book rules for indenting and nesting `if`s, but is still long and confusing:

```
// better grade-checking, but still not done:
if(score>=90) grade="A";
else {
  if(score>=80) grade="B";
  else {
    if(score>=70) grade="C";
    else {
      if(score>=60) grade="D";
      else grade="F";
    }
  }
}
```

The `&&`'s are gone, which is nice, but there's a lot of confusing {}'s and indents. It's also harder to see that this is really just a table look-up.

For the final version, we'll re-arrange it to look better. This is the most common way people write a range-slicing cascading `if`:

```
// final version of grade-checking:
if(score>=90) grade="A";
else if(score>=80) grade="B";
else if(score>=70) grade="C";
else if(score>=60) grade="D";
else grade="F";
```

Again, this isn't a new rule, just a clever way to rewrite things. It's still one giant `if` (it's exactly like the thing above.) Moving those `if`'s to the line right after each else is just a way to make it look better.

It runs line-by-line, quitting when it sees one it likes. One way to figure it out is to imagine it running for every possible number. The first `if` "slices off" everything 90 or more for an A. The numbers that get to the second line are the left-overs, all less than 90. The second line slices off 80 or more for a B. 79 and less goes on to the third line, and so on.

Looking at it another way, the line `if(score>=70) grade="C";` would tell you 97 was a C *if it was by itself*. But it's not by itself. It's part of an assembly line. 97 never makes it past line 1.

This idea works going down or going up. Here's the grade-checker starting from F and going up:

```
if(score<60) grade="F";
else if(score<70) grade="D";
else if(score<80) grade="C";
else if(score<90) grade="B";
else grade="A";
```

We can analyze it the same way: line one slices off 0-59 for an F, sending 60+ on. Line two slices off 60-69 for a D, sending on 70+.

Here's a different one. The old text-based dungeon MUDs didn't tell you the damage as a number – that was considered to be lazy and mood-breaking. Instead it turned the damage amount into a word. If you hit for 8, it told you "you cut the orc." Here's a damage-to-word table:

```
// damage-to-word table (damage is an int, and always 0 or more):
// 0-2: tickle
// 3-5: scratch
// 6-9: cut
// 10-15: hurt
// 16-25: wound
// 26+: mangle
```

This cascading `if` runs it (assume `hit` has the damage). Notice how each important number from the table occurs once:

```
string ouch;
```

```
if(hit<=2) ouch="tickle";
else if(hit<=5) ouch="scratch";
else if(hit<=9) ouch="cut";
else if(hit<=15) ouch="hurt";
else if(hit<=25) ouch="wound";
else ouch="mangle";

print("You "+ouch+" the "+monsterName);
```

A common error in a cascading-if is to forget an else in the middle. The program will still be legal, but it will run wrong. Here, I left out the else on the D line:

```
// bad code, missing an else:
if(score>=90) grade="A";
else if(score>=80) grade="B";
else if(score>=70) grade="C";
if(score>=60) grade="D"; // <-- oops, no else
else grade="F";
```

This is now *two* ifs. The first is a 3-line A/B/C if. The second is a simple if/else checking for D/F. Suppose grade is 93. The first `if` gives an A. But the second `if` gives a D. Forgetting the `else` makes it so everyone has a D or an F.

7.1.2 Do only one of these, none of the above

When you have several `if`'s in a row, each one could be true or false – all true, all false, or a mix could all happen. Often that's what you want. This example could say a number is only positive, or only 1-digit, or positive and 1-digit, or positive and close to 10, or all three:

```
// ex of how more than one if might be true:
if(n>0) print("positive");
if(n>-10 && n<10) print("one digit");
if(n>=8 && n<=12) print("close to 10");
```

But sometimes, it works out that only one of them can happen. Here's my "number to word" from before (written before we had `else`s):

```
if(n==1) w="one";
if(n==2) w="two";
if(n==3) w="three";
```

Logically only one of those is true, but you can't tell that right away. Rewriting it as a cascading `if` makes it more obvious that this is really a table look-up, and only one will be true:

```
if(n==1) w="one";
else if(n==2) w="two";
else if(n==3) w="three";
```

We didn't need to write it like that, but most people think it looks nicer.

Cascading `if`'s can also be used to make a "none of the above," by adding a final `else`. This example has a look-up table for what sound an animal makes, with a default sound "rrr-rrr":

```
if(ani=="cat") w="meio";
else if(ani=="crab) w="clickity click";
else if(ani=="sheep") w="baa-baa";
else w="rrr-rrr";
```

The other way to do this is with the fall-through trick, setting `w` to `"rrr-rrr"` at the top.

When most people see a cascading `if`, they know a final `else` is "none of the above," and no final `else` means the whole thing could do nothing.

Sometimes we use a cascading `if` to make sure that only one `if` can fire. This is a rare case, and is a little tricky, Here's a made-up example:

Suppose some formula adds 5 to numbers under 10, doubles numbers from 10-100, and divides larger numbers by 3. It's just a nonsense formula, but for real you sometimes have things like that.

My first try of putting that in the computer is to check each thing with an `if`. Here's the code, but it's got a sneaky logic error:

```
// wrong way to run the silly formula:
if(n<10) n += 5;
if(n>=10 && n<=100) n *= 2;
if(n>100) n = n/3;
```

The problem is that it might incorrectly do two of them. Say the number is 7. Line one adds 5 to get 12, like it should. But now stupid line two sees `n` is 12 and incorrectly doubles it! It doesn't know 12 isn't the input `n`, it's the answer `n`. We get the same problem with 60: line two correctly doubles to 120, then stupid line three incorrectly divides it!

A clever person could rewrite it to maybe avoid double-ifs. But an easy way is to turn them into a cascading `if`:

```
// correct silly formula, that never double-changes the input:
if(n<10) n += 5;
else if(n>=10 && n<=100) n *= 2;
else if(n>100) n = n/3;
```

7.2 Swapping

This isn't really an `if` problem, but it always happens inside of an `if`. Sometimes we want two variables, say `n1` and `n2`, to be in order. If `n1` is 5 and `n2` is 3, we want to switch them to be 3 and 5.

Switching the values of two variables is a little trickier than it seems. Suppose you write `n1=n2; n2=n1;`. That just makes two copies of `n2`.

To correctly switch variables, we need an extra spot, and a little dance. We usually call that a swap. This sorts `n1` and `n2` in order, if they weren't already:

```
int temp; // 3rd variable used for the swap
if(n1>n2) { // out of order
  // swap n1 and n2:
  temp=n1; // save n1
  n1=n2;
  n2=temp; // n2 gets saved copy of n1
}
```

You can hand-trace this for fun – make boxes for `n1`, `n2` and `temp`, and run the 3 lines. Changing the order usually makes it not work, but there is one other way to write it.

7.3 Incremental calculations

Often your instinct is to want a formula that just gives the answer. Like `n=` (insert equation here.) A lot of problems are easier to solve a little bit at a time.

Often we want to count up how many items fit some rule, like how many of `a`, `b` and `c` are positive. Here's my first try, which is long and not done:

```
if(a<=0 && b<=0 && c<=0) positiveCount=0;
if(a>0 && b>0 && c>0) positiveCount=3;
// check if only one is positive:
if(a>0 && b<=0 && c<=0 || a<=0 && b>0 && c<=0 || // I give up!!
```

A neat trick is, pretend you're counting on your fingers. Start the total at 0, look at each item, and add 1 if you like it:

```
int positiveCount=0; // we have 0, so far
if(a>0) positiveCount++; // add 1 shortcut
if(b>0) positiveCount++;
if(c>0) positiveCount++;

print( positiveCount + " are positive" );
```

We just push the answer a little at a time. By the time we finish, it's correct.

Here's the same idea, but it counts how many are between 40 and 60:

```
int inRange=0; // we have 0, so far
if(a>=40 && a<=60) inRange++;
if(b>=40 && b<=60) inRange++;
if(c>=40 && c<=60) inRange++;

print("There are "+inRange+" between 40 and 60");
```

Finding the largest of four numbers can also go a little at a time. It works like the "Price is Right" prize spin-off.

One person spins the wheel and steps into the winner's circle. They haven't won, but they're the winner so far. The next person spins and has to beat them. And so on. Each person only has to beat the previous best.

Here's how it looks in code, finding the largest of 4 numbers:

```
// find the largest out of a,b,c,d:
int largest=a; // a is the one to beat
if(b>largest) largest=b;
if(c>largest) largest=c;
if(d>largest) largest=d;
```

The trick is, `largest` is always the largest so far.

Just for fun, here's finding the largest of a,b,c,d using just a big if-else. It's longer and uglier:

```
if(a>b && a>c && a>d) largest=a;
else if(b>c && b>d) largest=b;
else if(c>d) largest=c;
else largest=d;
```

This version gets even worse with 5 numbers. In the incremental methods, we just need one more `if` for each extra item.

7.3.1 guessing game example

This is a more oddball example of computing something a little at a time.

Suppose we have some sort of darts or guessing game. 70 is the target, worth 5 points. We get 3 points for being within five, 2 points within ten, and 1 point within thirty (the exact numbers aren't really important.)

One way to solve it is with an ugly "hammer&tongs" approach, where we just brute figure the ranges and check them: "within 5" is 65-75, but not 70; "within 10 but not within 5" is 60-64 and 76-80; "within 30 but not 10 is" 40-59 and 81-100; and 0 points for "more than 30 away" can be the fall-through.

That's a lot of numbers. I'd probably make a number-line and a little chart, to keep track of them. Then we write an `if` for each. This works, but is hard to read:

```
// ugly range example:
int points=0;
if(n==70) points=5;  // exact
if( (n>=65 && n<=75) && n!=70) points=3;  // within 5, but not exact
if(n>=60 && n<65 || n>75 && n<=80) points=2; // 6-10 lower or higher
if(n>=40 && n<60 || n>80 && n<=100) points=1; // 11-30 lower or higher
```

One obvious way it seems like we could improve this is with a range-slicing cascading if. The trick to making a good one is figuring out what to slice off first. Usually you go low to high, or high to low. But for this one, it makes more sense to start at the target, and to go away from it. I think this looks not-too-bad:

```
if(n==70) points=5;  // exact
else if(n>=65 && n<=75) points=3;  // within 5
else if(n>=60 && n<=80) points=2; // within 10
else if(n>=40 && n<=100) points=1; // within 30
else points=0; // too far
```

This uses the same slicing principles. The second `if` is really "65 to 75, but not 70," because 70 was sliced off in the first line.

Finally, here's what I think is the best way to solve this problem. The really neat thing about some computer code is you can think "how would a human solve this?" and then program the computer to do that.

As a human, I'd first see that the points table is written as "how much away." So, I'd figure out how far `n` was away from 70, as a positive number. Then I'd look up that number in the "points for how far away" table.

Here's how that plan looks in the computer:

```
// figure out how far we are away from the target:
int missedBy = n-70;
if(missedBy<0) missedBy = missedBy * -1; // always positive

// table look-up:
if(missedBy==0) points=5;
else if(missedBy<=5) points=3;
else if(missedBy<=10) points=2;
else if(missedBy<=30) points=1;
else points=0;
```

It's easier to see the plan, and the cascading `if` is easier to read (it's the very first way I described it, before I confused myself with all the math.)

7.3.2 Gold mine problem

We have several gold mines. Each can have 1 or 2 workers. With 1 worker, they produce 2 gold. With 2 workers make 3 gold. We want to know how much total gold we can mine.

There are plenty of computer science word problems like this. They might be math, or might be about using `if`'s. They're probably not difficult, once you figure them out.

My first thoughts are to list facts: if we have more mines than workers, then mines don't matter – everyone works by themself, getting 2 gold. Workers past double the mines won't matter – every mine produces 3 gold. We can never have a mine with 2 workers, and another empty – we want a worker in each mine before we start to double-up.

From the darts example, I know writing code based on that will turn into a mess. An old trick is to pick some "typical" numbers, get the answers by hand, and see it that leads to a formula:

```
Mines  Workers  empty one two
  7       5       2    5   0
  7      34       0    0   7
  7      12       0    2   5

  5       6       0    4   1
 20      32       0    8  12
```

Without thinking about it, I computed only the workers in each mine. That sounds about right. We can easily compute the gold from that. It also seems as if there are 3 cases. Workers <= mines; workers between mines and double mines, and workers >= double the mines.

This partly done cascading `if` should properly select between the cases:

```
public int mines, workers; // input

  int m1=0, m2=0; // how many mines have 1 or 2 workers
  // NOTE: we don't care about mines with 0 workers, since they make no gold

  // testing to choose category::
  if(workers<mines) print("everyone works, alone");
  else if(workers<mines*2) print("all mines used, some are doubled-up");
  else print("all mines have 2 workers");

  // normal mines make 2 gold, doubled-up mines make 3:
  int totalGold = m1*2 + m2*3;
```

I should test it (maybe just in my head) with some of those sample numbers.

Then I'll have to work out the equations for each one. The middle is the most difficult. With 7 mines and 12 workers, the first 7 workers go to each mine, then the next 5 double-up. 5 came from 12-7. So `workers-mines` is how many doubled-up mines we have. All three equations, inside the `if`:

```
  // compute m1=mines w/1 worker, m2=mines w/2 workers
```

```
if(workers<=mines) { m1=workers; m2=0; }
else if(workers<mines*2) { m2=workers-mines; m1=mines-m2; }
else { m1=0; m2=mines; }

//print("m1="+m1+" m2="+m2); // testing
```

7.4 Tricks with modulo

Integers have a remainder operation named **modulo** (we stole the idea and the name from math.) It can be used in some sneaky ways. First, how it works.

We know integer division drops the fraction. 16/5 is 3, not 3.2. If you think of it this way, division really has two answers – the number of times it goes in, and the remainder. From second grade, 16/5 is 3, remainder 1. Both answers are perfectly good numbers.

The / symbol means to divide and give me the first answer – how many times it goes in. % means to divide and give me the second answer – the remainder. Obviously, if you divide by 5, the remainder is between 0 and 4. It looks like this:

```
print( 87%5 ); // 2. Fives up to 85, remainder 2

print( 35%5 ); // 0. Five goes evenly into 35, No remainder

print( 13%3 ); // 1. Threes up to 12, one left over

print( 7%10 ); // 7. Tricky. Ten goes in zero times. Everything is the remainder
```

No one ever uses modulo in big equations, like x+y%5*2. But, just so you know, modulo counts as division, so has the same precedence as `*` and `/` (it happens before plus and minus.) Also, modulo with negative numbers works funny (does 5 go into -8 minus one or minus two times?) People rarely use modulo on negative numbers.

Here are some common tricks with modulo:

If you divide by 2, the remainder is 0 or 1. Remainder 0 meant that 2 went in evenly, 1 means it didn't. So, checking even/odd integers works like this:

```
if(n%2==0) print("even");
else print("odd");
```

This is also a fun example of math inside an `if`. Try it with 7: Two goes into 7 three times, remainder 1. So `7%2==0` is false.

You can think of the `if` as "if 2 goes into `n` with no remainder."

Another fun trick with modulo is breaking a number into digits. `n%10` is always the one's place, just because of how base-10 numbers work. Try it: take `569%10`. Ten goes into 569 fifty-six times, with 9 left over.

We can get the 10's and 100's places by shifting them into the 1's place. Since integers drop the fraction, dividing by 10 merely shifts: `569/10` is 56. The 10's place was moved into the 1's place. In use:

```
print("Ones place is " + n%10 );
print("Tens place is " + (n/10)%10);
print("Hundreds place is " + (n/100)%10);
```

Now we can do some fun modulo examples, where we grow the answer.

A fun one is converting pennies into quarters, dimes, nickels and left over pennies. As a human, you take out the quarters, then take dimes out of the remainder, and so on. We can do that in a program. For simplicity, say we always have 99 or less pennies. The first part:

```
public int pennies; // input, in Inspector
public quarters, dimes, nickels; // output (we will see them change)
```

Finding quarters is just dividing by 25 and dropping the remainder, which is exactly how integer math works (you knew there was a reason.) The left-over pennies are the remainder, which **modulo** will get:

```
void Start() {
  quarters = pennies/25;
  pennies = pennies%25; // always 0-24 left over pennies
}
```

Testing just that, if we enter 93 for pennies, running gives us 3 quarters and 18 pennies (in the Inspector.) Not completely correct, but correct so far. Using 12 for pennies gives 0 quarters and 12 pennies, which is also correct, so far. The pennies variable is now the 0-24 left-over pennies.

Then we do the same thing for dimes and nickels:

```
void Start() {
  quarters = pennies/25;
  pennies = pennies%25;
  // pennies is now 0-24, after taking out quarters

  //print("after Q, pennies="+pennies); // debug

  dimes = pennies/10; // will be 0, 1 or 2
  pennies = pennies%10; // will be 0-9, after taking out dimes

 // print("after D, pennies="+pennies); // debug

  // only makes 0 or 1 nickel, but shorter than an if!!
  nickels=pennies/5;
  pennies=pennies%5; // 0-4 left over pennies
}
```

Those two prints labelled *debug* are typical left-over testing lines, commented out instead of deleted until we're completely sure this is working correctly. With it all done, 63 for pennies, should give quarters=2, dimes=1, nickles=0 and pennies=3.

Here's a similar example, converting seconds to hours, minute & seconds like "00:05:32". It starts with the modulo trick:

```
public int seconds;
public int minutes, hours;

void Start() {
  hours=seconds/(60*60); // 60 times 60 seconds in an hour
  seconds=seconds%(60*60); // left-over seconds is 0 to 3599

  minutes=seconds/60;
  seconds=seconds%60; // 0-59 left-over seconds
}
```

Leaving seconds-per-hour as `(60*60)` is a common trick to make programs easier to read (3600 looks funny to me, but I can remember there are 60 seconds per minute and 60 minutes per hour.) It feels like you should pre-do as much math as possible, but the compiler does that for you. Trying to keep formulas readable is more important.

For more fun with ifs, we can try to print it as 00:00:00. We turn each number into a string, then add a `"0"` in front if needed. This would come at the end of `Start`:

```
string wHours, wMinutes, wSecs; // will be things like "00" "45" "04"

wHours=""+hours; // using the int-to-string trick
if(hours<10) wHours="0"+wHours; // turn "9" into "09"

wMinutes=""+minutes;
if(minutes<10) wMinutes="0"+wMinutes;

wSecs=""+seconds;
if(seconds<10) wSecs="0"+wSecs;

// combine, with colons:
string wTime = wHours+":"+wMinutes+":"+wSecs;

print("time left is "+wTime);
```

This would print 8 hours, 31 minutes and 4 seconds as 08:31:04.

7.5 Mixed and/or

Long expressions using AND and OR mixed-up can be difficult to read. Like everything else, they usually make sense when you know what they're for. But sometimes trying to figure that out can be a pain.

Suppose I like any color cats and white dogs. This `if` correctly checks that:

```
// any cat, white dog
if(a=="cat" || a=="dog" && color="white") print("good pet!");
```

It works because `&&` has higher precedence than `||`. In other words, the computer puts invisible parens around the "dog and white" part. It's like `4+3*5`. You could put parens around them yourself, if you think that looks better.

Here's a longer one that likes cats, white dogs, goats or red skunks. For real, I might add parens on white dog and red skunk, and break it over two lines, as in the second one:

```
// any cat, white dog, goat, red skunk
if(a=="cat" || a=="dog" && col=="white" || a=="goat || a=="skunk && col=="red")

//maybe nicer way?:
if(a=="cat" || (a=="dog" && color=="white") ||
   a=="goat" || (a=="skunk" && color=="red") )
```

Back to just dogs and cats, if you wanted white no matter what, and a dog or cat, you could write it like this, with parens around "cat or dog":

```
if((a=="cat" || a=="dog") && color=="white")
```

We need the parens. Without them, it would be a cat or a white dog.

Some longer ones; this is "white or black" and "cat or dog" (so white dogs, white cats, black dogs and black cats). The parens around the OR groups are required:

```
if( ( (color=="white" || color=="black") && (a=="cat" || a=="dog") )
```

For fun, if we forgot the parens, we'd get something completely different:

```
if( ( color=="white" || color=="black" && a=="cat" || a=="dog" )
// what the computer sees:
if( ( color=="white" || (color=="black" && a=="cat") || a=="dog" )
```

That would accept lots of things it shouldn't: anything white and any color dog.

7.5.1 not, DeMorgan's law

This is a little tricky, and not vital. I just think you should see it. It's about tricks using not (`!`).

The not symbol (`!`) flips between true and false. For example `if(!(x>0))` is "if `x` is not greater than 0." It counts as a real math symbol, so it goes inside the official `if` parens, with all the other math. It has higher precedence than almost everything, so you usually have to use parens with it.

It can often make things easier to write. For example, suppose I want to check if `x` is *not* between 10 and 20. Here are two ways to write that:

```
if( x<10 || x>20) print("not 10-20"); // old "too small or too big" way

if( !(x>=10 && x<=20) ) print("not 10-20");
```

The second one is longer, but I'm used to seeing "between 10 and 20", and the `!`-bang means not, so I can directly read it as "not 10 to 20."

Suppose I *don't* want a white dog or cat. Writing it directly is tricky (I'll do it later.) Writing "white dog-or-cat" and not-ing it might be easier:

```
// anything but a white cat or white dog:
if(( !( (a=="cat" || a=="dog") && color=="white") )
```

The inside is "either a cat or a dog, has to be white," which is a white cat or dog. Then the `!` in front make it the opposite. This might look terrible to you now, but once you get more practice using `if`'s it starts being a neat trick.

In general, the trick is: whenever you're writing the inside of an `if`, see if writing the opposite might be easier. If it is, write that, and put a `!` in front of it.

You can also use the `else`. This looks silly, but it works:

```
// very silly "anything but a white cat or white dog":
if( (a=="cat" || a=="dog") && color=="white" ) {} // do nothing
else print("not white cat or dog");
```

For fun, here's a direct "not a white cat or dog." Notice the `||`'s and `!=`'s:

```
if( a!="cat" && a!="dog" || color!="white")
```

The official rule for bringing a *not* through an AND/OR grouping is **DeMorgan's Law**. In algebra, you can take `a(b+c)` and distribute the `a` to get `ab+ac`. DeMorgan's law says you can do something similar with true/false and *not*. Except it's more complicated. You have to `!`(not) each part *and* flip the AND/OR. Here's an example:

```
// not 1 to 10, converted using DeMorgan's:
if( ! (a>=1 && a<=10) )

// bring the not through:
if( !(a>=1) || !(a<=10) ) // && flipped to an ||

// turn "not >=" into <; turn "not <=" into >:
if( a<1 || a>10)
```

Here's an example with `||`, showing that it flips to `&&`. The inside part is "5, 9 or 16" and then I not it, to say "anything except 5, 9 and 16", then I work out it using DeMorgan's:

```
// not 5, 9 or 16:
if( !( a==5 || a==9 || a==16 ) )

// Convert using DeMorgan's. Bring the not through, flipping || to &&:
if( !(a==5) && !(a==9) && !(a==16) )

// ! == is !=:
if( a!=5 && a!=9 && a!=16) print("a is not 5, 9 or 16");
```

We can test this one – the `&&`'s really are the correct symbol (`||` would always be true – 5 isn't 9 or 16.)

The most useful things to remember about this are 1) slapping a big `!()` around everything is fine, but moving the `!`s into the parts is extra tricky, 2) if you really, really need to convert true/false formulas using `!`s, there's an equation for it. If you only remember "French pirate sounding name," that should be enough for you to find it years from now.

7.6 Switch Statements

There's a limited version of an `if` that isn't useful for anything, and is kind of complicated to learn. But it's semi-common, so you should at least see it.

Here's an example `if` and then the replacement `switch`

```
if(n==5) { a="cow"; }
else if(n==12) { a="dog"; }
else if(n==19) { a="goat"; }
// and so on
```

A `switch` statement to do the same thing looks like this:

```
switch(n) {
```

```
  case 5: a="cow";
    break;
  case 12: a="dog";
    break;
  case 19: a="goat";
}
```

It looks so strange that you figure it must run faster, or be better in some way, but they aren't. In the old days, we tried a lot of crazy ways to write computer languages. If you're curious, reading about a `switch` is like going into computer language history.

Chapter 8

Unity `if` examples

Now that we can "think" with ifs, we can improve the moving, coloring, and resizing examples from before. Even if you have no interest in making games, this is a pretty good way to practice writing `if`'s.

Here's the old "move across the screen" code with only `x` changing:

```
public float x=-7.0f;
public float xSpd=0.1f;

void Update() {
  x = x + xSpd;
  transform.position = new Vector3(x, 0, 0);
}
```

Instead of shooting past the right side, we could have it wrap around like the old Asteroids game. We just need to figure out how to say "when it goes past the right side, return to the left."

That's two separate things to figure out. Going to the left side is easy, we've done it before: `x=-7;`. You might think that can't be right – putting `x=-7;` in Update will keep us glued to the left edge. But that's the other part – how to do it only at the correct time.

Checking past the right side is `if(x>7)`. Combining them is: `if(x>7) x=-7;`. Move with wrap-around looks like this:

```
public float x=-7.0f;
public float xSpd=0.1f;

void Update() {
  x = x + xSpd;
  if(x>7) x=-7; // new wrap-around to left side
```

```
  transform.position = new Vector3(x, 0, 0);
}
```

That new line is pretty short. A chapter ago it would have been a really bad example – why would we want to change from 7.1 to -7? But now that little `if` does exactly what we need.

It probably looks a little "snappy". It will look better if the pop-out and pop-in were both off the edge of the screen. Tweaking +7 and -7 could make this look really nice. The easiest way to do that is to turning them into variables. Then we can adjust them as the program runs:

```
public float x=-7.0f;
public float xSpd=0.1f;

public float xMin=-7.0f, xMax=7.0f; // the new left and right side variables

void Update() {
  x = x + xSpd;
  if(x>xMax) x=xMin; // replaced -7 and 7 with variables

  transform.position = new Vector3(x, 0, 0);
}
```

The `if` is even a little bit easier to read this way. It says what it does "if you're past the maximum, drop down to the minimum".

We can use the same trick to change the size. This grows from normal to double-wide then snaps back and repeats (remember 1 means normal-size):

```
public float xSz=1.0f;

void Update() {
  xSz += 0.01f;
  if(xSz>2.0f) xSz=1.0f; // wrap

  transform.localScale = new Vector3( xSz, 1, 1);
}
```

Wrapping-around color is about the same. To make it more interesting I'll have it go down (remember color numbers are from 0 to 1):

```
public float g=1.0f; // start full green

void Update() {
  g -= 0.01f; // go down to 0
  if(g<0) g=1.0f; // wrap 0 back to 1
```

```
  GetComponent<Renderer>().material.color = new Color(1, g, 0);
}
```

Notice it checks for *less than* 0, since it's going down. I kept the red slot as always 1 for fun. It goes from (1,1,0) to (1,0,0), which is yellow to red (red and green makes yellow).

8.1 bouncing

The other fun thing we can do when we hit the edge is bounce. This will flip the speed when we hit the right edge:

```
public float x=-7.0f;
public float xSpd=0.1f; // <- speed is a variable, so we can change it

void Update() {
  x += xSpd;
  if(x>7) xSpd = -0.1f; // start moving left

  transform.position = new Vector3(x, 0, 0);
}
```

Again, pretty cool for just one `if`. It looks even cooler if you have a wall and tweak the edge so it looks like it's bouncing off it.

An improved version would bounces off both sides, and keep whatever speed it started with. If we use `xSpd *= -1` we can turn a forward speed of 0.08 (or whatever) into the same speed going backwards, then positive again when it hits the left side:

```
public float x=-7.0f;
public float xSpd=0.1f;

void Update() {
  x += xSpd;

  if(x>7) { xSpd *= -1; }
  if(x<-7) { xSpd *= -1; }

  transform.position = new Vector3(x, 0, 0);
}
```

You should be able to watch see `xSpd` flip between +0.1 and -0.1 as it moves back and forth.

There's also a cool bug. When we go past 7, it should be because we were going forward and took 1 step too far. Flipping the speed will have us step back in bounds on the next move. But if we start way off the edge, like at 12, that's not true. We'll be past 7 every time, the speed will flip back and forth and the cube will just vibrate.

Size change with bouncing works the same, except the numbers are smaller (1 to 2.) This makes a square grow to 2, then shrink back down to 1, repeating:

```
public float sz=1.0f, spd=0.005f;

void Update() {
  sz+=spd;
  if(sz>2 || sz<1) spd*=-1; // flip speed when passes min or max
  transform.localScale = new Vector3(sz, sz, 1); // x&y grow the same
}
```

For fun, we can rewrite our movement bouncing using a different plan. We can have a variable `dir`, will be always be +1 or -1 for the direction we want to move:

```
public float x=-7.0f;
public float xSpd=0.1f; // always positive
public int dir=+1; // +1 is right, -1 is left

void Update() {
  if(dir==1) { // we're going right:
    x += xSpd; // add the speed
    if(x>7) dir=-1;
  }
  else { // we're going left:
    x -= xSpd; // subtracting the speed
    if(x<-7) dir=+1;
  }

  transform.position = new Vector3(x, 0, 0);
}
```

That's not quite as nice, but it fixes the off-edge bug, and it's got a nested `if`.

8.1.1 Speed increase

In older movement examples, I had the speed get faster and faster without limit. Now we can use an `if` to add a limit. Or to reset the speed, or whatever else can can think of.

I'll go back to the simple wrap-around Cube. This starts slow and speeds up. The new part is a maximum speed, using a simple `if`:

```
// wrap-around, increasing speed, to a maximum:
public float x=-7.0f, xSpd=0; // speed _starts_ at 0

void Update() {
  x = x + xSpd;
  if(x>7) x=-7; // wrap-around

  xSpd += 0.001f; // very slow speed increase
  if(xSpd>0.8f) xSpd = 0.8f; // speed limit

  transform.position = new Vector3(x, 0, 0);
}
```

Once it reaches a cruising speed of 0.8, it's only a simple wrap-around.

We could just as easily have it slow down to a minimum. We'd start `xSpd` high, subtract a tiny bit, and use a "can't get below this" `if`:

```
  xSpd += -0.001f; // a little slower
  if(xSpd<0.05f) xSpd = 0.05f; // lowest speed limit
```

If we copied these over the old lines, it would move quickly, but eventually settle to be a simple, slow wrap-around.

You might remember the trick where I started the speed negative and always increased it. The Cube went backwards, slowed and then reversed direction. If we do that for `y`, it's like we're making gravity – a ball going up, slowing, then falling. We may as well also add a bounce off the floor:

```
public float y=3.0f; // nearer the top, giving it room to fall
public float ySpd=0; // any speed is fine, but 0=not moving seems obvious

void Update() {
  y+=ySpd;
  ySpd -= 0.003f; // gravity

  // bounce off bottom:
  if(y<-3) {
    ySpd*=-1; // bounce up by flipping speed
    y=-3;  // just in case, don't let it fall below the floor
  }

  transform.position = new Vector3(0, y, 0);
}
```

We've seen a bounce, and we've seen the slow-down-then-change-direction trick. But combined they look pretty cool – like a real ball bouncing.

Gravity and bouncing off the floor looks even cooler if we add the old `x` movement, bouncing off the sides. This cube will bounce in realistic-looking arcs:

```
public float x=2.0f, y=3.0f; // set these to anywhere interesting
public float xSpd=0.1f; // always goes this fast, left or right
public float ySpd=0.1f; // a little bit up, for fun

void Update() {
  x += xSpd; y+=ySpd;
  ySpd -= 0.003f; // gravity

  // bounce off left, right and bottom:
  if(x>7 || x<-7) xSpd*=-1;
  if(y<-3) { ySpd*=-1; y=-3; } // bounce off bottom

  transform.position = new Vector3(x, y, 0);
}
```

It's just a few simple tricks, combined, but it should look a lot like a ball bouncing around in a box.

We can try a resetting speed increase using color. The code below starts with the yellow-to-red code. But it makes green go faster and faster, then resets it to change slowly again:

```
public float g=1.0f, gSpd=-0.01f; // green, green speed change

void Update() {
  g += gSpd; // green goes down
  if(g<0) g=1.0f; // wrap around back to 1

  gSpd-=0.0005f; // the speed change increases
  if(gSpd<-0.03f) gSpd=-0.01; // reset to starting speed when too fast

  GetComponent<Renderer>().material.color = new Color(1, g, 0);
}
```

Hopefully this isn't too bad. It's a simple wrap-around, except going down, not up. Then another wrap-around for the speed. Then we had to figure out good numbers for the speed. 0.0005 seems too small, but we're only going from 1 to 0, and the frames happen quickly.

Even so, it's easy for a few simple tricks, combined, to start to look confusing.

8.2 Counting

We can do some interesting things if we count how many laps the cube makes. Making a lap counter is easy. Make an int variable, and add 1 on each reset:

```
// partial code for a lap counter:
public int laps=0;

void Update() {
  x += xSpd;
  if(x>7) {
    x=-7;
    laps=laps+1;
    // do special lap stuff here
  }
```

If we watch in the Inspector, `laps` will go up each time, but nothing else interesting will happen.

The next part is using the counter to do something. I'd like it to turn red after 3 laps, and green after 7 laps, then back to white on lap 8. All of that goes in the "do special lap stuff here" area:

```
// change color after laps 3, 7 and 8:
public float x=-7.0f, xSpd=0.1f;
public int laps=0;

void Update() {
  x += xSpd;

  if(x>7) {
    x=-7;
    laps += 1;

    // color change checks:
    if(laps==3)
      GetComponent<Renderer>().material.color = new Color(1,0,0); // red
    if(laps==7)
      GetComponent<Renderer>().material.color = new Color(0,1,0); // green
    if(laps==8)
      GetComponent<Renderer>().material.color = new Color(1,1,1); // white
  }

  transform.position = new Vector3(x, 0, 0);
}
```

This is probably the largest body of an `if` we've had yet. But I think it seems natural. When you get past the end you should: 1) wrap around, 2) count the lap, and 3) use `if`s to check for the three special laps.

The color-changing `if`'s are a little new, but not really. We've used the line to change color before, just never guarded by an `if`.

A possibly confusing thing about this is what color is the 4th lap? The code doesn't say. Of course it's red since we turn it red on 3 and stays that way until we change it again. Another confusion is I said it resets after the 8th lap, but the code simply turns it white. That's because there's no "reset" command. Since it started out as white, turning it white feels like a reset.

A little tricker use of a lap counter, we can do something every 3rd lap, using modulo. `laps%3` (the remainder after dividing by 3,) will be 1,2,0,1,2,0 ... after each lap. So `(laps%3==0)` is true every 3rd lap.

This sets the speed to "slow", every third lap:

```
if(x>7) {
  x=-7;
  laps++;

  // set speed for next lap:
  if(laps%3==0) xSpd=0.03f; // slow
  else xSpd=0.1f; // otherwise normal speed
}
```

This sets us to slow at the end of lap 3, meaning that lap 4 is slow (and laps 7, 10, 13 and so on). That's still every 3rd lap, so fine. We could adjust the test to `if(laps%3==2)` if we absolutely needed lap 3 to be the slow one.

8.3 Delay counter

If we want to wait for a second or two, we can do that with an `int` counter, an `if` and a plan.

The simplest plan is having the counter be how many Updates we should skip, count down, and zero means we're done waiting. If we want to wait for 60 Updates (about a second,) we set the counter to 60.

This moves a Cube in big steps, with an 80 update delay in-between each move. It should hop from left to right:

```
public float x=-7.0f;
public int delayCounter=0; // how many Updates to skip

void Update() {
```

```
  if(delayCounter>0) { // still waiting:
    delayCounter -= 1;
  }
  else {
    x = x + 1; // one big step
    transform.position = new Vector3(x, 0, 0);

    delayCounter=80; // wait for one more second
  }
}
```

The movement code is now guarded by an `if`. That's what `if`'s do – they say that we *might* do something. An `if-else` is good for this: either we're waiting on the countdown, or we're not. You could watch `delayCounter` spin down to 0 (the true part of the `if` is running,) then see the Cube pop forward and the counter reset (the `else` ran one time.)

When the delay ends, we don't have to restart it right away. This version moves smoothly with a wrap-around, using the delay only when we hit the edge (it waits a little bit before moving across again):

```
public float x=-7.0f;

public int delayCounter=0; // how many Updates to skip

void Update() {
  if(delayCounter>0) { // still waiting:
    delayCounter--; // subtract 1 shortcut
  }
  else {
    x += 0.1f; // standard small move
    if(x>7) { // wrap and signal for a pause:
      x=-7;
      delayCounter=80;
    }
    transform.position = new Vector3(x, 0, 0);
  }
}
```

Essentially, `delayCounter` is there for us whenever we need it. We want to delay after a lap, so we set `delayCounter`. The `it` takes care of making it actually delay us.

It might look funny having delay come first in the `if`. We can flip the order to put moving first and waiting last:

```
  if(delayCounter<=0) { // not waiting
```

```
    x+=0.1f;
    if(x>7) {
      x=-7;
      delayCounter=80;
    }
    transform.position = new Vector3(x, 0, 0);
  } // end of movement
  else delayCounter--;
}
```

This next example uses a delay with a size change. The basic code makes the Cube be size 1, 2, 3, 4 then back to 1. A delay counter makes it do that once every second. We should see it gradually pop larger:

```
public float sz=1.0f;
public int delay=0;

void Update() {
  if(delay>0) delay--;
  else {
    // sz goes 1,2,3,4 then back to 1:
    sz+=1;
    if(sz>4) { sz=1; delay=120; }
    else delay=60;

    transform.localScale = new Vector3(sz,sz,sz); // all-around larger
  }
}
```

Notice how I got tricky with setting the delay. When it wraps around to 1 I give a longer 120-tick delay.

We could write the 60 or 120 delay a different way: always give a delay of 60, with an extra 60 for a wrap-around:

```
  if(delay>0) delay--;
  else {
    sz+=1;
    delay=60; // base delay
    if(sz>4) { sz=1; delay+=60; } // extra delay for wrapping

    transform.localScale = new Vector3(sz,sz,sz); // all-around larger
  }
}
```

What I like about this is the best way depends on how you think about it. Maybe you feel like it's two different delay times. Or maybe it makes more sense

as a base delay with extra for wrapping.

Finally, I mentioned counting down was the simplest plan for a delay. Another plan would be counting up to the total. We'd need an extra variable. To wait for 20 Updates we'd set the total to 20 and the counter to 0. Here it is with the movement wrap-around:

```
  int x=-7;

  int delayCount=0; // during a delay this counts up to the total, from 0
  int delayTotal=0; // no delay right now

  void Update() {
    if(delayCount<delayTotal) // delayCount goes up, delayTotal stays the same
      delayCount++;
    else {
      x+=0.1f;
      if(x>7) {
        x=-7;
        delayTotal=80; // how long to wait
        delayCount=0; // reset the count
    }
    transform.position = new Vector3(x, 0, 0);
  }
}
```

This is more complicated, which usually means worse. But a possible advantage is if you want to show the delay as a percent – like one of those bars that fills in. It would be `1.0f*delayCount/delayTotal`. We can't compute that with a single count-down, since we don't save where it started.

8.4 State variables

In the second version of bouncing, with `dir`, my idea was the program has two stages – moving right and moving left. `dir`'s job was to remember which stage we were on. I used -1 and +1 as easy-to-remember values, but it could have been 1=left and 2=right.

Sometimes we have a program with more complicated stages, and we formalize the idea of a stage variable. Here's the whole trick:

Draw out a picture with your stages and the rules for going to the next. Number them – the exact numbers don't matter; the numbers are like their names.

To program it, make an `int` to remember which stage you're on. Inside, use a big "do only one of these" cascading if-else-if-else. Make one part for each

stage. Whenever you want to go to a stage, like stage 3, write `stage=3`

Here's an example. The comments under `stage` explain what it does:

```
public int stage=0;
// stage 0 = move right fast until we get to x=6
// stage 1 = grow until we get to size 2
// stage 2 = slowly move up to y=4
// stage 3+ = done, do nothing

float x=-7, y=0; // position, for stage 0 and 2
float sz=1.0f; // Cube size, for stage 1

void Update() {
  if(stage==0) { // move right until past 6
    x += 0.1f;
    if(x>=6) stage=1; // done moving right
   }
  else if(stage==1) { // grow until size 2
    sz += 0.02f;
    if(sz>=2.0f) stage=2; // done growing
  }
  else if(stage==2) { // move up until past 4
    y += 0.03f;
    if(y>=4) stage=3; // done going up
  }
  else if(stage==3) {} // stage 3 is do nothing

  // simpler to do these always, even if some stages don't need them:
  transform.position = new Vector3(x, y, 0);
  transform.localScale = new Vector3(1, sz, 1);
}
```

The trick is it only does one part each Update. For stage 0 it only runs those two lines: move a little right, past 6 means done – set `stage` to 1. Then, starting on the *next* Update, run only the two lines for stage 1. `stage` is the control for what Update does.

This one changes color in stages, using a delay counter so it's not a blur. Also, instead of stopping at the last stage, it repeats to stage 0:

```
int colStage=0;
// 0 = red for a long time
// 1 = bright blue quickly
// 2 = yellow for a medium time, then back to 0
```

```
int delay=100; // wait a little before starting, for no reason

void Update() {
  // whole thing is in a basic delay if-else:
  if(delay>0) delayCount--; // delayed: count down and wait
  else {
    // state machine ifs:
    if(colStage==0) {
      // red
      GetComponent<Renderer>().material.color = new Color(1,0,0);
      delay=150;
      colStage=1;
    }
    else if(colStage==1) {
      // bright blue:
      GetComponent<Renderer>().material.color = new Color(0.5f,0.5f,1);
      delay=30;
      colStage=2;
    }
    else if(colStage==2) {
      // yellow
      GetComponent<Renderer>().material.color = new Color(1,1,0);
      delay=70;
      colStage=0; // back to red
    }
  }
}
```

This spends almost all of it's time subtracting from `delay`. Each stage runs for only one update: it changes color, sets the new delay, and goes to the next stage (for when the delay is finally over.)

The important thing is how we're still using our extra made-up `stage` variable to remember which step we're on.

Not really important for these examples, but the official name for this sort of thinking is a `State Machine`. You can find state diagrams (circles with arrows,) or GUI's (Unity's built-in Mechanim animation system is a drag&drop state machine.)

For fun, why wouldn't we use a word for the state, making it easier to read? Here's what that would look like:

```
public string stage="move right";
// will be "move right" "grow" "move up" or "done"

...
```

```
if(stage=="move right") {
  x+=0.1f;  if(x>=6) stage="grow";
 }
else if(stage=="grow") {
  sz+=0.02f;  if(sz>=2.0f) stage="move up";
}
else if(stage=="move up") {
  y+=0.03f;  if(y>=4) stage="done";
}
```

It's easier to read, but it's so easy to make a mistake. If we wrote `"go up"` or `"Move up"` (capital M) the code would stall, with no errors. It turns out that using strings to control what happens almost always causes hard-to-spot bugs. Later we'll see a trick to make the numbers easier to read in a safer way.

Chapter 9

Scope and Namespaces

This chapter is sort of a break. Just some rules about different ways and places to declare variables. We'll use them a little, and every programmer needs to know them, eventually. But they won't hurt your head as much as those `if` examples.

9.1 Scope

So far, we've been declaring variables in two places: inside of Start and Update, or outside of them (the Inspector variable trick.) Both ways are legal, but they have different rules for how long the variables live and who can see them.

Variables declared outside Start or Update are called **global**. Global variables live for the entire program, and everyone can see and use them.

Variable declared inside Start or inside Update are **local**. Local variables can only be used in the part where they were declared. When that part finishes, local variables are destroyed. A local variable in Update is created and destroyed each time Update runs.

The technical term for where a variable exists is **scope**. The scope of global variables is the entire program. The scope of local variables is inside the {}'s where they were declared. Being destroyed by leaving the {}'s is called *going out of scope.*

Here's a small example of local variables not existing outside where they were made. `n` is declared in Start, so doesn't exist in Update:

```
void Start() {
  string n = "Nif";
  print("n="+n); // n=Nif
}
```

```
void Update() {
  print("update n="+n); // ERROR -- no such variable n
}
```

Local variables also don't "use up" the name outside their scope. This is a good thing. It means we can declare one `n` in Start, and a different `n` in Update:

```
void Start() {
  string n = "Nif"; print("n="+n);
}
void Update() {
  int n=5; // legal. We can reuse the name
  print(n*2); // 10;
}
```

This is why local variables were invented. Imagine you have a very large program, with Start, Update and piles of other things like it. When you declare local variables in one section, you never have to worry about whether someone also declared them in some other section.

You could declare everything as global. But programs are usually easier to read if you use global only for things you need to keep around between Updates.

Here's an example using move-and-wrap-around. It snaps `y` lower as `x` crosses thresholds. `x` has to be global, since it's remembering where I am. But `y` is better as a local, since I recompute it each time:

```
public float x=-7; // <-global

void Update() {
  x+=0.1f;
  if(x>7) x=-7;

  // compute y based on x. 1.5, 0 or -1.5 as we move:
  float y; // <-local
  if(x<-2) y=1.5f;
  else if(x<2) y=0;
  else y=-1.5f;

  transform.position = new Vector3(x,y,0);
}
```

`y` is really just a temporary, used to help place the Cube at the correct height. Declaring it as a local, right before we use it, makes it easier to see that.

9.1.1 More global rules

You're allowed to declare globals anywhere in the big curly-braces. It doesn't have to be at the top. They all get declared first – the compiler scans for global

as an early step. The rule about needing to declare a variable before you use it is real, but it's only for local variables.

Just to show it can be done, this silly program declares n1, n2 and n3 in funny but legal places:

```
int n1;

void Start() {
  n1=1; n2=7; n3=45; // legal to use n2 and n3
}

int n2; // <- also a legal place to declare a global

void Update() {
  n3++; // this counts as declared
}

int n3; // a legal place to declare a global
```

Most people declare them at the top. But I sometimes declare a global just before the part that uses it – like right before Update.

Friendly languages auto-initialize all variables: numbers to 0 and strings to "". Languages that want to run faster never auto-initialize. C# splits the difference – it auto-initializes global variables, but not locals. This can be confusing:

```
int n1; // global, this is 0

void Start() {
  int n2; // local, not initialized
  n1++; // legal, increases it from 0 to 1
  n2++; // error
}
```

There's a special rule allowing declare-and-init for global vars. It's a special rule since global variables aren't run like regular lines – they're magically declared all at once. This means you can't give them values based on other globals:

```
float wide=10; // fine
float tall=wide*1.5f; // error -- wide doesn't exist yet

 void Start() {
   tall=wide*1.5f; // set complicated globals here
 }
```

A traditional program set every global's value as the first thing it does. In Unity, that would be in Start.

9.1.2 block scope

You're allowed to declare variables inside of an if/else {}. They're local to that little area and go away after it's done. That's called **block** scope since the thing the {}'s make is officially called a block.

The idea is the same as global vs. local – keep variables from cluttering things up by getting rid of them when you're done. Here if I have more cats than dogs I'll temporarily compute `extraCats`. It's gone when the {} block ends, which is what we want:

```
  if(cats>dogs) {
    int extraCats=cats-dogs;
    if(extraCats>2) { ... }
    cost = extraCats*3;
    ...
  }
else {
  int extraDogs=dogs-cats;
  ...
}
```

It works in either place. `extraDogs` only lives inside of the else brackets.

A short, real use is for a variable swap. We need an extra variable to make the switch, but not afterwards:

```
  if(x>y) {
    int tmp=x;
    x=y;
    y=tmp;
  }
  // tmp no longer exists
```

It looks nice to declare `tmp` exactly when we need it, and it won't interfere with the rest of our program, since it's deleted right away.

9.1.3 Common errors

Sometimes when you want to change a global, you accidentally redeclare it, like this:

```
int cats;

void Start() {
```

```
  int cats=10; // Oppss! But not an error. Hides the global cats
  // cats=10; // We meant to do this
  cats += 5; // even this is messed up -- changes the local cats
}
```

This isn't a red-dot error – it's a much worse non-error error. The warning is *local cats shadows global cats*.

We now have a local `cats` which hides the global one we meant to use. That code sets local `cats` to 15. But when Start ends local `cats` is thrown away. Global `cats` stayed 0 the whole time.

Whenever you use a variable, just think "Am I changing an existing global, or making a new local variable?"

A similar error can happen with block scope. This accidentally makes `food` only exist inside the `if`:

```
if(ani=="cat") string food="mouse";
else string food="canned";

print(ani+" eats "+food); // error -- no variable named food
```

We have to declare it outside the `if`:

```
string food;
if(ani=="cat") food="mouse";
else food="canned";
```

9.2 Namespaces

Most systems have lots and lots of built-in globals. A standard computer trick is to group them into things like folders, called **namespaces**.

For example, Unity has global variables for the screen's height and width, which way a mobile device is being held, and so on. These are all in a **namespace** which they named `Screen`.

The rule for looking inside a namespace is to use a dot. If you type `Screen`-dot (screen and then a period) the pop-up will show the options. `Screen.width` is the width, in pixels, of the window. It's really the `width` variable in the `Screen` namespace.

For fun, you can test by having Update copy `Screen.width` into an Inspector global. While running this, you can resize your window and watch `ht` change:

```
public float ht;
void Update() { ht=Screen.width; }
```

You might notice that the pop-up for `Screen` says `public sealed class`, and not namespace. **namespace** is just the general term. Some languages use the actual word `namespace` in the language. Others use the term *package*. C# uses a few different ways. But no matter what, they all work like `folder.variable`.

Here are more namespaces and stuff in them, just for fun examples. You won't need to know them:

- `Mathf` holds math stuff. `Mathf.PI` is 3.1415. `Mathf.Floor` is a built-in that rounds 3.6 down to 3 (we haven't seen the rules for this yet.)
- `Time` holds info about game-time stuff. `Time.deltaTime` is the number of seconds between the last Update and this one (it's usually about 0.02.) `Time.time` is how many seconds since you pressed Play.
- `Debug` holds testing commands. `print` is really a shortcut for the `Log` command in the `Debug` namespace. `Debug.Log("abc");` prints `abc`.
- `System.Math.PI` also holds PI. It's a namespaces in a namespace (think of it like a folder in a folder.) `System` is the standard C# namespace. If you're wondering, `System.Math.PI` is a `double`. `Mathf.PI` is a `float`.

`Time.time` is one of my favorite examples of how namespaces work. The `Time` namespace holds a lot of things about time and timing things, so `Time` is a good name for it. The most commonly used thing in it is how many seconds the game has been running. That variable should have an easy, short name, so we picked `time`.

Together, it's the `time` variable in the `Time` namespace – `Time.time`.

9.2.1 Notes/rules

You can use the same variable in different namespaces. That's really a version of the scope rules. That's why `System.Math.PI` and `Mathf.PI` are both legal. You could have `Screen.height` and `Raccoon.height` and also name one of your variables `height`.

In the editor, typing the dot triggers the pop-up. Suppose you type `Screen.wi`, click away, come back, and the pop-up is gone. To get it back, delete down to just `Screen` and retype the dot.

Later on, C# re-uses the dot in a different way. When we come to it, I'll write this again and explain it. I want to sort of pre-warn you about it, since it's one of those things that can be really confusing.

The `using`'s at the top of the program are about namespaces. You don't need to know this – only keep reading if those two lines at the top of every

program are bugging you. `UnityEngine` is the master namespace for all the Unity built-ins. `Screen` is really inside of `UnityEngine`. The real width of the screen is `UnityEngine.Screen.width`.

`using UnityEngine;` at the top lets you skip it everywhere else. it says: if you can't find something they typed, try looking for it in `UnityEngine`. But, again, not vital to know just now.

Chapter 10

More ifs and randomness

This section is about using random numbers. But, really, it's more examples using `ifs` and programming tricks. Random numbers are to give us more things to do. And they're just fun to use.

10.1 Random rolls

The command to roll a random number is `Random.Range`, and then the numbers you want to roll between:

`n = Random.Range(1.0f, 10.0f);` rolls a random number between 1 and 10. It's a `float`, so it could be 1.35 or 6.93201.

The program below tests this in Update. It sprays out random numbers between 2 and 6:

```
void Update() {
  float n = Random.Range(2.0f, 6.0f);
  print( n );
}

// Possible output:
5.573
4.152441
2.80017
4.376728
```

For now, the command just works that way. The only thing we need to know is that we can change the two numbers inside.

Here are more of the rules:

- The command is really `Range` in the `Random` namespace. That's why it's `Random.Range`.

- The parens are like the ones for `print`. They have to be there. The insides are: the low number, a comma, and the high number. The comma is special, and just goes there.
- Random numbers naturally get funny streaks – it's normal to get lots of high numbers in a row, or lots of low ones. Don't worry if it does – just let it run for a while and things will even out.

This example shows how variables and assignment statements work the same as always. We roll a random number once, using Start, then print the variable over and over in Update:

```
float n; // global

void Start() {
  n=Random.Range(100.0f, 200.0f);
}

void Update() {
  print(n);
}

// Possible output:
106.5
106.5
106.5
```

All of the magic is the way `Random.Range(100.0f, 200.0f)` could be anything. But once it picks a number, it's just a number. `n` doesn't change by itself, since variables never do that. It's not "infected" with randomness.

And, of course, if you ran this over and over it would print different numbers, but always the same number repeated.

This next example is the same idea. We roll a number at random, print it, then add 1 and print it. The second number is always 1 more than the first. The trick is, don't think "one more than a random number." Think "one more than `n2`":

```
float n2;

void Update() {
  n2 = Random.Range(50.0f, 60.0f);
  print("A " + n2);
  n2 = n2 + 1;
  print(" B " + n2);
}
```

```
// Possible output:
A 52.6
 B 53.6
A 59.23
 B 60.23
```

This last example shows we can roll any range we want at any time, and they don't interfere with each other or "carry over":

```
void Update() {
  float a, b;
  a = Random.Range(3.5f, 4.5f);
  b = Random.Range(-1000.0f, 1000.0f);
  print(a+" / "+b);
}

// Possible output:
3.78 / -765.4
4.36 / 350.4563
4.145 / 204.5
4.0 / 4.0 <-- odds are super-slim, but it could happen
```

`a` is always between 3.5 and 4.5, and `b` is always something in-between plus and minus a thousand.

10.2 Random Positions

We can make the moving, wrap-around code more interesting with random rolls. If have make a variable for the `y` position, we can roll it randomly after each lap:

```
public float x=-7;
public float y=0;

void Update() {
  x+=0.05;
  if(x>7) { // wrap-around:
    x=-7;
    y=Random.Range(-4.0f, 4.0f); // <- new line
  }
  transform.position = new Vector3(x,y,0);
}
```

It rolls random `y` once at the start of each lap. This means it stays perfectly level as it moves right. `y` won't change again until the next lap.

Because of how random works, it might be near the bottom for a few laps, or appear to restart in the same place a few times. But if you let it run (increase the speed if you want) you should see it jump around.

A fun thing is we know our Cube is just snapping to a new `y`. But if the edges are set right, it looks like a new Cube coming in on a different row.

Another thing we can do is pick a completely random spot on the screen, by randomly picking `x` and `y`. This uses a delay counter to pop anywhere at random about once a second:

```
public int delayCount=0; // no delay for first time

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

Suppose we want the random position to alternate left side / right side. If we save the `x` position, we can check which side we're on now with `if(x<0)`, then roll the opposite:

```
public int delayCount=0;
public float x=0; // global so we can look at the old x

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float y=Random.Range(-4.0f, 4.0f);

    // left or right. Do opposite of old x value:
    if(x<0) x=Random.Range(1.5f, 7.0f);
    else x=Random.Range(-7.0f, -1.5f);

    transform.position = new Vector3(x,y,0);
  }
}
```

I just picked 1.5 as a number far enough from 0 that we can easily tell which side we rolled. Also note how -7 is first in `Random.Range(-7.0f, -1.5f)`, since it's the smaller number. That's easy to mess up with two negative numbers.

10.2.1 Random size

This randomly rolls the width and height of a Cube, about once a second. It's really no different than rolling a random x/y position, but it looks different:

```
int delayCount=0;

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float wd=Random.Range(0.4f, 3.0f);
    float ht=Random.Range(0.4f, 3.0f);
    transform.localScale = new Vector3(wd, ht, 1);
  }
}
```

This might make a square, if it happened to roll width and height close together, but it will usually make rectangles.

If we wanted a random *square*, we could roll one size and use it for width and height:

```
    // replace last three lines of "random-size cube" with this:
    float sideLen=Random.Range(0.4f, 3.0f);
    transform.localScale = new Vector3(sideLen, sideLen, 1);
```

I like this example, because it shows that even random numbers work the way we program them. We can roll once for each side, or once for both, or anything else we can think of.

10.2.2 Color

This rolls a completely random color. 0-1 for red, green and blue:

```
void Start() {
    float r=Random.Range(0.0f, 1.0f);
    float g=Random.Range(0.0f, 1.0f);
    float b=Random.Range(0.0f, 1.0f);

    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

A completely random color is usually an ugly grayish brown. A trick is to pick a good color, and roll close to those values.

This rolls a random "orangey" color by picking a smaller range for each slot:

```
void Update() { // every frame, so will flicker
    float r = Random.Range(0.8f, 1.0f); // big red
```

```
    float g = Random.Range(0.3f, 0.7f); // medium green
    float b = Random.Range(0.0f, 0.4f); // low blue
    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

Another method is to pick a random "brightness", and multiply the original orange color by it. We always get the same shade of orange, but brighter or darker:

```
void Update() { // every frame, so will flicker
    float bright = Random.Range(0.3f, 1.0f); // "brightness" 1=100%

    // using base orange of (1, 0.5, 0.1)
    float r = 1.0f*bright;
    float g = 0.5f*bright;
    float b = 0.1f*bright;
    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

Colors are harder to tell apart than sizes or positions. In the first program, I can see some red and yellows pop up. The second one has more light/dark difference, but always looks pretty orange, to me.

I use tricks like this to give every brick in a wall a slightly different color. Sometimes one way just doesn't look right, and the other way does.

10.3 More things to randomize

Way back, we could take x,y movement code and change `ySpd` to make it go diagonal. But we had to do it by hand. Now we can roll it randomly. Then we also randomize `xSpd` and have it randomly travel only 1 to 8 across. It rerolls all three things each time it ends a lap. It seems complicated, but watching it run should clear it up:

```
public float x=-7, y=0; // the position, starts center-left
public float xSpd=0.1f, ySpd=0; // straight and slow, for now
public float endXthisLap=7.0f; // randomly changes each lap

void Update() {
  x+=xSpd;
  y+=ySpd;
  if(x>endXthisLap) { // wrap-around and re-roll everything:
    x=-7; y=0;

    // randomize speed, slope, and distance we go:
    xSpd=Random.Range(0.05f, 0.15f); // always forward
    ySpd=Random.Range(-0.08f, 0.08f); // up or down
```

```
    endXthisLap=Random.Range(-1.0f, 8.0f); // x for when lap ends
   }
  transform.position = new Vector3(x,y,0);
}
```

The trick is it only rerolls the speeds at the end of a lap. While it moves in one lap, `xSpd` and `ySpd` don't change, so it gets a straight diagonal for each one.

The examples using a delay counter always used 80. We can roll that randomly. This pops the Cube to random spots, with a random delay between pops:

```
float delayCount=0; // changing to float, since random rolls are floats

void Update() {
  delayCount--;
  if(delayCount<=0.0f) {
    delayCount=Random.Range(30.0f, 150.0f); // the new part

    // same random position code from before:
    x=Random.Range(-7.0f, 7.0f);
    y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

10.4 Coin flips, percents, random tables

There's no built-in heads/tails coin flip, but we can make one. The usual way is to roll a random 0 to 1 and check for above 0.5.:

```
int n=Random.Range(0.0f, 1.0f);
if(n>0.5f) print("heads");
else print("tails");
```

Here's the code to pop us around randomly, but it flips a coin for each delay. This always waits either 60 or 140 between pops:

```
public int delayCount=0; // trick it into moving right away

void Update() {
  delayCount--;
  if(delayCount<=0) {

    // next delay: heads=short, tails=long:
    float delayFlip=Random.Range(0.0f, 1.0f);
```

```
    if(delayFlip<0.5f) delayCount=60;
    else delayCount=140;

    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

I like this because there's no built-in way to randomly roll 60 or 140. But, we can make a plan: "flip a coin, heads=60, tails=140." If you watch it run, you really can see that it's using 2 delay lengths.

This next example flips a coin twice, just to show we can. Each time the Cube wraps around, we restart the lap either high or low, and turn either red or green:

```
public float x=-7, y=0;

void Update() {
  x+=0.2f;
  if(x>7) {
    x=-7;

    // heads=high, tails=low:
    float coinFlip = Random.Range(0.0f, 1.0f);
    if(coinFlip>0.5f) y=3;
    else y=-3;

    // heads=red, tails=green:
    coinFlip = Random.Range(0.0f, 1.0f);
    if(coinFlip>0.5f)
      GetComponent<Renderer>().material.color = new Color(1,0,0);
    else
      GetComponent<Renderer>().material.color = new Color(0,1,0);
   }
   transform.position = new Vector3(x,y,0);
}
```

Again, the way randomness works, this might get high+red a few times in a row, or get green the first four times. But if you wait enough laps it will even out and you'll see high/green, low/green, high/red and low/red.

I snuck in a tricky part. Notice how the first roll uses the declare-and-assign shortcut: `float coinFlip=Random` .... I reuse it for the next flip with just `coinFlip=Random` .... I had to remember not to double-declare `coinFlip`.

Moving on, a coin flip is really just a 50% chance. If we want a 10% chance we can change it to `if(coinFlip<0.1f)`. That's a tiny programming change, but it let's us do different stuff.

This changes the 60/140 delay so give an 85% chance for a short delay. The only code change is 0.5 becomes 0.85. But when you watch it, it seems like the Cube is popping at an exact interval, then "whoa" – it surprises you by staying in place extra-long:

```
// next delay: 85% short, 15% very long:
delayFlip=Random.Range(0.0f, 1.0f);
if(delayFlip<0.85f) delayCount=60; // 85% chance
else delayCount=240;
```

If we have more than two choices, we can make a table using a cascading if. Here's a table I made up for how long the delay should be:

```
// delay: medium 60%, short 30%, long 10%
// Table:
//  0 to 0.6 medium (90 ticks)
//  0.6 to 0.9 short (30 ticks)
//  0.9 to 1.0 long (240 ticks)
```

Here's the cascading, range-slicing `if` to make the table. We roll a 0 to 1 percent, then see which one it is:

```
// next delay: medium, short or long:
delayRoll=Random.Range(0.0f, 1.0f);
if(delayRoll<0.6f) delayCount=90; // medium
else if(delayRoll<0.9f) delayCount=30; // fast
else delayCount=240;
```

Notice how there's only one roll.

Here's the same idea, but with random treasure (the usual way games do it – mostly cheap stuff, and only a 2% chance for the top prize):

```
float roll=Random.Range(0.0f, 1.0f);
string treaure;
if(roll<0.5f) treasure="copper pennies";
else if(roll<0.85f) treasure="silver silverware";
else if(roll<0.98f) treasure="gold rings";
else treasure="magic sword";
```

We can mix and match these ideas. In this next example, I want something more complicated for where to put `y` when we wrap around: 70% of the time, y should be 0, otherwise flip a coin for high or low.

```
public float x=-7;
public float y=0;

void Update() {
  x+=0.2f;
  if(x>7) {
    // wrap-around:
    x=-7;

    float inCenterRoll=Random.Range(0.0f, 1.0f);
    if(inCenterRoll<0.7f) y=0.0f; // y is usually centered
    else {
      float hiLoFlip=Random.Range(0.0f, 1.0f);
      if(hiLowFlip<0.5f) y=Random.Range(0.5f, 4.0f); // high
      else y=Random.Range(-3.0f, -0.5f); // low
    }
  }

  transform.position = new Vector3(x,y,0);
}
```

This has got 3 levels of random. Even after it rolls "not the center" then gets the coin flip for "below," it still rolls for exactly how much below. This type of random looks different than a single anywhere roll. Over time it's obvious that it favors the center, but avoids the zone near the center.

Chapter 11

Functions, part I

This section is an introduction to things called **functions**. You may have heard them called by different names – methods or procedures – they're all the same thing. They work about the same in all languages, and C# uses pretty much the same rules as everyone else.

The basic idea is you can write program lines somewhere else and give them a name. Then anyone can use that name to run those lines. It's merely an organizational trick – we won't be able to do anything we couldn't before – but it's a really useful trick for making larger programs.

`Start()` and `Update()` are functions. The difference between them and the ones we'll write is Unity won't run ours automatically. Our functions will just sit there until Start or Update need to use them.

`print("abc")` is running a function. For now, ours will be even simpler, for example `printHello();` will always print `hello`.

Just so you know, the stuff in this chapter is going to seem a little limited. It's only the first part of the rules for functions. But we'll still be able to do helpful things.

11.1 Intro Function examples

Here's a program with a `printHello` function. It does nothing, but it's legal:

```
public class testC : MonoBehaviour {
  void Start() {}

  void Update() {}

  void printHello() {
    print("hello I say");
  }
```

```
}
```

Look how nicely `printHello` fits in there with it's friends Start and Update. For fun, go to the code editor and play with the -/+ box (left side) to hide/show it. The editor knows this is named `printHello` and it owns the stuff in the curly-braces.

It never runs because we never tell it to. The system has a special rule to auto-run Start and Update. But the general rule is functions just sit there, waiting for their names to be called.

If you remember from way, way back, I wrote that `void Stork()` instead of Start wasn't an error, but wouldn't run:

```
// another legal program that does nothing
public class storkTest : MonoBehaviour {
  void Stork() {
    print("starting program");
    print("ending program");
  }
}
```

The computer thinks you wrote a program with no Start or Update, which is legal, and one function named `Stork`. Stork is a perfectly good function – it just doesn't have a special auto-run name.

To run a function, use the name. To avoid confusing it with a variable, add a paren-pair at the end. This uses printHello, twice:

```
public class testC : MonoBehaviour {

  void Start() {
    print("What does the alien say:");
    printHello();
    print("Again, again!:");
    printHello();
}

  void printHello() {
    print("hello I say");
  }
}
```

The system still runs only Start. But now Start runs `printHello`. The output is:

```
What does the alien say:
```

```
hello I say
Again, again!:
hello I say
```

Of course this is useless so far. The only important thing is how Start jumps to the function and runs the line, then Start keeps going, jumping back to the function a second time. And the whole time we count as running Start line-by-line.

Here's another with two functions. I'm putting one of mine in front of Start, just to show it can be done. The computer doesn't care about the order, since it looks them up by name:

```
string w;

void addZsToBack() {
  w=w+"zzz";
}

void Start() {
  w="frog";
  addZsToBack();
  w+="#";
  addZsToBack();
  print(w); // frogzzz#zzz
}

void addFrontD() { // this is never run (since Start never calls it.)
  w="D"+w;
}
```

The basic run is the same as the printHello example. Start runs, and calls a function twice.

An interesting thing is we never use the second `addFrontD` function. That's fine and not an error or even a problem. The other interesting thing is how `addZsToBack` uses an assignment statement and a global variable, which is also fine.

11.2 Function rules

As usual, it's nice to summarize the rules and notes, in one place. There's nothing really new here:

- Our new functions go "next to" `Start` and `Update` – just inside the big outer `testC : Monodevelope` { } curly-braces; but outside of all other functions.

- The order of functions never matters. You could move `Start` to after `Update` and the program would work the same. You can call any function no matter if it was written before or after you. You can use cut&paste to rearrange functions later on.

- The function name is any legal identifier (the same rules for variable names.) `do_stUff34` is a legal, bad, function name. Like variable names, they're case-sensitive (`printhello();` would be an error, on account of the lower-case H.) They don't need to start with a capital letter. Unity just thought it looked nicer for Start and Update.

- For now the heading is always `void`, the name, then empty `()` parens. Eventually we'll put different things in there, but not this chapter. For now the function call is also always empty `()` parens.

- The body of a function goes inside of { and }. It can have any statements and any number of them: assignment, print, `if`'s, declaring local variables.

Functions normally just sit there, waiting to be run when we need them. Here's the obligatory terrible analogy:

Imagine a cookbook. There's a section on buttering a pan, sifting flour, browning onions – those are like functions. Now suppose you're making lasagna. You go to that recipe, follow it, and you're done. But, the lasagna recipe says "butter a 9 inch pan (page 126,)" which you do. Then later "brown 3 large onions (page 428,)" and you do that. Then near the bottom, it's back to "butter a caserole dish (page 126.)"

Each time it does that, you save your spot, flip to the page it says, follow those instructions, then back to where you were in the lasagna recipe. We never looked at the flour sifting part of the cookbook, since it never told us to.

If you think about it, cookbooks are pretty old and those guys worked out a good system. Computer programs would be crazy not to copy it.

Again, just to have them, here are the written-out rules for running functions:

- The technical term for running a function is **calling** it. You call a function by using the name, followed by `()`. In computer talk, `printHello();` is a function call.

- Function calls count as statements.

- Function calls always return to just after where you called them. A call jumps to the function, runs lines from the top, then jumps back when it hits the end.

- It's fine to have extra unused functions.

- Currently we can't mix a function call as part of a longer line. For example `q=printHello()+"abc";` is junk. `printHello();` all by itself is the only way to use them, for now.

11.3 More function examples

I promise functions will be useful, but this is more practice with the rules.

Let's start with two functions and a Start which doesn't use either. This prints `x`:

```
void A() { print("this is A"); }

void B() { print("I am B"); }

void Start() { print("x"); }
```

If we changed `Start` to be like below, the program would print `"I am B"` three times. The first two will be on different lines because the program doesn't care about where you put returns and spaces (the last one is also on a different line, but you knew that):

```
void Start() {
  B(); B(); // put two function calls on same line, just for fun
  B();
}
```

We can mix it up however we like, or put them inside `if`s:

```
void Start() {
  int x=0;
  B();
  if(x>0) {
    A();
    B();
  }
  x=5;
  if(x>0) { A(); A(); }
}
```

Notice how the function calls in the first `if` are indented like regular statements (since they are.) They should look pretty natural in there. As usual, we only do them if `x` was more than zero (it isn't, so we never run them.)

Then the second `if` used the "combine short stuff on 1 line, but don't forget the {}`'s`" style. I like how the two function calls look just like regular statements in there.

This whole thing runs B, skips the first `if`, then runs A twice.

Anyone is allowed to call a function – we can call one from `Start` and also from `Update`. Here's a simple example where `Start` calls a function, and `Update` uses a counter to call it a few times:

```
int x=0;

void A() { print("this is A"); }

void B() { print("I am B"); }

void Start() {
  B();
  print("end of start");
}

void Update() {
  if(x<6) A(); // first 6 times
  if(x<3) B(); // only first 3 times
  x++;
}
```

`Start` calls `B` and is done. Then Unity-magic runs `Update` over and over, same as it always does. As usual, `x` will be 0,1,2 .... . Update calls `A` then `B` for a while, then just `A`, then nothing. Output will be:

```
I am B
end of start
this is A
I am B
this is A
I am B
this is A
I am B
this is A
this is A
this is A
```

Functions are allowed to do anything `Start` or `Update` could do. This function steps `cows` from 0 to 100 then wraps. The result is that Update Moo's once every 100 frames:

```
public int cows=0;

void doOneStepCowWrap() {
  cow=cows+1;
  if(cows>100) cows=0;
}

void Update() {
  doOneStepCowWrap();
  if( cows == 0 ) print("moo");
}
```

This is looking more useful. Pretend you are *so sick* of looking at add-and-wrap-around lines. In Update we only have to see `doOneStepCowWrap`, which we can assume does one step of the cow wrap-around. This is also our first function with an `if`. Pretty exciting stuff.

A common real use of functions is to hide ugly stuff. I'm not a fan of the long GetComponent color-changing line. My plan is to use globals `r`, `g` and `b`. To set color I'll merely change those globals and call a function to run the line I hate typing:

```
float r, g, b; // global color variables

void applyColor() {
  GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

Now we never have to see `GetComponent` again. It's like we made a new command named `applyColor();`. A program fragment using it:

```
void Start() {
  r=0.4f; g=1; b=1; // light aqua
  applyColor();
}

void Update() {
  ...
  if(laps==5) { r=1; g=0; b=0; applyColor(); } // red
}
```

This is clunky, and we'll make it much better in a later chapter. But I think even this is an improvement, especially if we change our color a lot.

We can do something similar with the position command. Assuming we're using global `x` and `y`, we can make it so `applyPos();` runs the real line for us:

```
int x=-7, y=0; // the position variables we always use

void applyPos() {
  // assumes global x and y set to where we should be
  transform.position = new Vector3(x, y, 0);
}
```

Now we can move with `x=0; y=2; applyPos();`.

11.3.1 Longer resetToLeft example

In this section, I want to have an example of a function we just naturally make as we're writing a program. I want to start with a semi-interesting program that shoots a ball with a curve, using `xSpd` and `ySpd` tricks.

Here's the program. So far, it's just tricks with movement code, and no functions yet (well, I'm using `applyPos`, from above, since I like it):

```
float x=-7, y=0;
float xSpd=0, ySpd=0;

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f; // ball speeds up

  if(x>7) {
    x=-7; y=0;
    xSpd=0;
    ySpd=Random.Range(-0.3f, 0.3f);
  }
  applyPos();
}
```

I haven't done exactly that before, but it's just old stuff combined. `xSpd` is increasing, starting from zero, so it looks like it's falling from left to right. And `ySpd` just makes it move up or down. Combining them gives a curve like tossing a watermelon off a roof, except going from left-to-right.

Now onto the function part. I want to pull the reset code from out of Update and into a function. This code has the same lines as above, just moved around:

```
void resetToLeft() { // these 4 lines used to be in update
  x=-7;
  y=0;
  xSpd=0;
  ySpd = Random.Range(-0.3f, 0.3f);
}

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f;

  if(x>7) resetToLeft(); // this used to be 4 lines in {}'s
}
```

It turns out that, depending on the y speed, the ball could fly off the top, or the right side, or the bottom. I'd like it to check for all 3, changing into a different color. The code for that is simple, and uses our new functions. The good part is in Update:

```
float x, y;
float xSpd, ySpd;
```

```
float r,g,b; // set these then call applyColor

void Start() {
  resetToLeft(); // may as well use these functions to set it up
  applyPos();
}

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f;

  if(x>7) { // fall off right = blue
    r=0.4f; g=0.4f; b=1; applyColor();
    resetToLeft();
  }
  else if(y>3) { // fall off top = yellow
    r=1; g=1; b=0; applyColor();
    resetToLeft();
  }
  else if(y<-3) { // fall off bottom = red
    r=1; g=0.3f; b=0.3f; applyColor();
    resetToLeft();
  }
  applyPos();
}
```

We could have written that without functions. Every function call would be the lines inside of it, pasted in. It would be longer and harder to read. That's the only problem functions solve – avoiding cut&paste.

Functions also help keep things consistent. Suppose we needed to change where `x` starts on a reset. `resetToLeft` is a single spot to change that. Without that function we'd need to hunt down all 4 places and change them.

11.4 functions with more math

Functions are allowed to have all sorts of program lines in them, even complex ifs. Here's a function that prints a silly random name, that Start uses twice:

```
void Start() {
  print("Hal starts at:");
  randomTown();
  print("and walks to:");
  randomTown();
}
```

```
void randomTown() {
  // makes things like "Clarkberg" or "New Tomatown"

  string mid="";
  string ending="";

  // use a look-up table to pick "town" "ville" or "berg":
  float endingRoll = Random.Range(0.0f, 1.0f); // using the % and table trick:
  if(endingRoll<0.33f) ending="town";
  else if(endingRoll<0.66f) ending="ville";
  else ending="berg";

  // flip a coin for "Clark" or "Toma":
  float midFlip=Random.Range(0.0f, 1.0f);
  if(midFlip<0.5f) mid="Clark";
  else mid="Toma";

  string townName = mid+ending;

  // for fun, 10% chance to add "New" in front:
  float newCheck=Random.Range(0.0f, 1.0f);
  if(newCheck<0.1f) townName="New "+townName;

  print( townName );
}
```

`randomTown` has got all the stuff that's usually in Update or Start: declaring local variables, ifs, cascading ifs, growing variables. Functions can use it all. It would be weird if they couldn't.

To finish up, it might print this:

```
Hal starts at:
Clarkberg
and walks to:
New Tomaville
```

11.5 Common Errors

As usual, just so we see them, here are some common errors and their messages:

- Forgetting the parens in the function call. Ex: `randomTown;`. The computer thinks you're trying to use the function name like it was a regular variable. It gives the general-purpose error: *Only assignment, call, increment, decrement, and new object expressions can be used as a statement.*

It doesn't say anyhing about missing parens (but at least it tells you the line.)

- Trying to "call" a variable like a function. Ex: `n();`. That gives a pretty cool error: *Expression denotes a 'variable', where a 'method group' was expected.*
  Method is another word for function, so this isn't too bad. It's saying the `()`'s make it expect a function.

11.6 Style

This is mostly a summary of things I've mentioned before:

- Function names that do things usually have a verb in them, like `applyColor`, `resetToLeft` or `setSize`.
- It's fine to use 1-line function to "rename" a command. For example, `void turnYellow() { GetComponent<Renderer>().material.color = new Color(1,1,0);}`.

  Sometimes we call that a wrapper or a front-end, since all it does it pass you along to a real command doing the actual work.
- One type of function is something we think will be of general use. `applyColor` is like this. We might paste it into every new script we start (assuming we always use the global `float r, g, b;` trick. We'll find a better way soon.)
- Another idea for a function is something we'd only use in one program, in several places. `resetToLeft` is an example of this. No other program would use it, but it shortens the one it's in.
- Another type is a function used only once, like `doCowWrap`. All it does is move some code out of Update, which is fine.
- Figuring out what would be good to turn into a function is difficult, and takes a lot of practice. You have to waste a lot of time writing bad ones. You can look up advice about "cohesion" and so forth, but sadly, they only make sense after you've written a lot of bad functions.
  So don't feel like you have to try too hard to make them. If you suddenly think "hey, this could be a function," go ahead and write one.

11.7 Chains of functions

The rule "call the function, run it, come back where you came from" also applies to chains of functions. A function is allowed to call another function. The computer can remember as many return locations as it needs to.

Here's a mechanical example showing a function chain. In this, `A` calls `B` which calls `C`. Printing "start" and "end" in each function is sometimes added for real, to try to track down funny behavior. Here it just makes a nicer example:

```
void Start() {
  C();
  B();
  A();
}

void A() {
  print("starting A");
  B();
  print("ending A");
|

void B() {
  print("B1");
  C();
  print("B2");
}

void C() { print("C"); }
```

The first call, `C()` is nothing special. It runs `C` and comes back to `Start`.

This next one, `B()`, prints `"B1"`, then jumps to `C` which prints `"C"`, back to `B` which prints `"B2"`. Then finally back to Start. It prints this:

```
B1
C
B2
```

The last function call, to A, has a bigger chain: `Start` calls A, prints and calls B, which prints and calls C, which prints and pops back to `B` then `A` then back to Start. Output is:

```
starting A
B1
C
B2
ending A
```

You may notice that it's just `starting A` and `ending A` wrapped around what B does. Which it obviously has to be.

A way to think about the function-chain rule is, suppose you call function `Dog` that does something you want. Maybe `Dog` also calls `E` and `F`. Maybe `E`

calls a bunch of other functions. You don't have to care. It doesn't matter. All that matters is that `Dog` did what you wanted and came back. What happened inside of `Dog` are just unimportant details.

Suppose we often want to recenter a ball and turn it blue. We could write this function:

```
void centerBlue() {
  r=0; g=0; b=1; applyColor();
  x=0; y=0; applyPosition();
}
```

It didn't need to call applyColor and position – it could have included those lines inside of itself. But it doesn't matter that it used them, either. Calling `centerBlue` does what it should, and that's all that matters.

11.7.1 Multiple local scopes

This section is just explaining that local variables are fine even when one function calls another. The way the computer handles that is sort of interesting, and might help you understand functions better. But this isn't a new rule or anything you need to know for a while.

Here's some code to start with. `Start` calls a function named `abc`, nothing special so far:

```
int x=5;

void Start() {
  int n = 7;
  abc();
  print("start n = "+n); // prints 7
}
```

It seems pretty obvious that last line has to print 7. We didn't change `n`, and it's our personal, local variable, so no one else can even see it. `x` could change, but it's a global, so we expect that.

Suppose `abc` also declares local `n`. That shouldn't matter to `Start`, and it doesn't. But it seems funny that we now have local Start-`n` and local abc `n`, and both at the same time when Start calls `abc`.

Here's the rest of the example, and a picture of how the computer handles it. Not only does `abc` have a local `n`, but it calls another function `def` which has a third local `n`:

```
void abc() {
  string n;
  n="cow";
```

```
  def();
  n += "y";
  print("abc n is "+n); // cowy
}

void def() {
  int n=20;
  n += 5;
  print("def n is "+n); // 25
}

Output (from the original Start):
def n is 25
abc n is cowy
start n = 7
```

Here's a picture of when we get to `def`. It really means that Start and `abc` are waiting, with their own saved local `n`'s:

```
Start |   abc    |  def
----- |  ------  |  ---
n: 7  |  n: cow  | n: 25
```

These are formally called **Stack frames**. Each function (even `Start` and `Update`) gets one. They hold all the local variables for each function. When you call a function, a new stack frame is made for it, local vars are set up, and it's put on top of yours. When a function ends, its frame pops off the stack, leaving yours the way it was.

In the picture, `Start` is waiting for `abc`, which is waiting for `def`, which is running. Everyone's local variables are remembered, and they never conflict with each other.

Again, the important thing: local variables work the way you think they should. Re-using the same name won't mess you up, not even with function calls.

This next thing is really far from things you need to know, but kind of interesting: stack frames are why creating and destroying local variables takes zero time. Before the program runs, the compiler figures out the stack frame for each function, with the exact arrangement of each local variable.

When you call a function, the computer gives you the precomputed amount of space for all local variables. The exact location of each one is already built into the function.

And again, that's not important to know. I just don't want you to be worried about local variable creation and destruction slowing down the program. If they were bad, we wouldn't have invented them.

Chapter 12

Functions, part II - parameters

12.1 Parameters

If we had to write the built-in `print` function using the rules we have now, it wouldn't be very useful – just `print();`. We couldn't tell it what to print. `print("abc");` is an example of giving an input to a function. We can write our own functions that work the same way.

To do this, we're going to need several new rules: what the inputs can be, how the function sends them, how a function definition says the inputs it wants, and how it uses them.

The basic idea is that everyone, even functions, likes to read their data from variables. So if a function needs an integer input, we'll give it a specially marked integer variable. Then we'll make up rules to get our input into that variable. There are a lot of rules, but they're all just for this basic idea.

To get started, here's a 1-input function that turns integers into words. If you call `showAsWord(1)` this will print `"one"`:

```
void showAsWord(int n) {
  if(n<=0) print("zero");
  else if(n==1) print("one");
  else if(n==2) print("two");
  else print("many"); // a better version would have more numbers
}
```

The new thing in the heading is the `(int n)` inside the parentheses. It says that `showAsWord` must have an integer input. It also says the input will be in a variable named `n`. The computer does that automatically. When you call

`showAsWord(4);` the computer declares `n` and puts a 4 into it. Then it runs the function.

Now, to see how cool this is, cover up the heading. Just look from `if(n<=0)` on down. Those four lines are an ordinary cascading `if`. If we saw this anywhere else, we'd figure that `n` was declared somewhere and somehow got a value, but the exact way doesn't matter.

That's the key idea of inputs to functions. The magic part is where we declare the variable from the heading and fill in the caller's value. But after that, there's no more magic – `n` is just a regular variable.

Just to be sure, here's the function being used, nothing special:

```
void Start() {
  showAsWord(1); // "one"
  showAsWord(0); // "zero"
  showAsWord(2); // "two"
  showAsWord(7); // "many"
}
```

Notice how there's no `n` in this part. We never need one. When we call `showAsWord(1);`, the computer creates local `n` inside the `showAsWord` function, set to 1. And as we know, local variables are invisible to everyone else and never interfere with anyone else either. `n` is for the function's personal use only, which is perfect.

Here's a similar example, written in steps this time. I want a function that tells me whether water is `"ice"`, `"water"` or `"steam"` at a certain temperature. The first part is easy – I'll pretend we have variable `temp` and write a cascading `if`:

```
if(temp<=32) print("ice");
else if(temp<212) print("water");
else print("steam");
```

That's not a function – just regular code – but I figured I need do know how to actually do it before making it into a function.

Functions need names – I'll pick `printWaterState` for this. I need to decide ahead of time whether the input is an int or float. Temperatures could have decimals, so float. In the code above I already picked `temp` for the variable name. So the heading looks like `void printWaterState(float temp)`. The complete function is:

```
void printWaterState( float temp ) {
  if(temp<=32) print("ice");
  else if(temp<212) print("water");
  else print("steam");
}
```

The same magic happens – when someone calls `printWaterState(60.0f);` the 60 is automatically copied into our `temp` variable.

To be complete, some function calls:

```
void Start() {
  printWaterState(150.6f); // "water"
  printWaterState(7); // "ice" (auto convert int to float works as normal)
}
```

Notice again, there's no `temp` variable here. That's good. We run the function by just giving it any old float.

Here's a 1-input function to resize ourself. Whatever float we give it, we'll be that big:

```
// sample uses:
resize(0.2f); // tiny
resize(5.0f); // 5x5x5: over half the screen

void resize(float newSize) {
  transform.localScale = new Vector3(newSize, newSize, newSize);
}
```

If you remember, size is really an x, y and z. This function sets them all to the one number we give it. It's written for a program where we never want a funny x-only stretch. It's a nice shortcut for setting x, y and z to the same value.

It feels like a real computer command. `resize(0.2f);` says what it does, and does what it says. It hides that ugly localScale stuff from us. It's doing what functions are for.

Here's the same idea, but with colors. A fun fact: if you set red, green and blue all to the same value, you get a "grey scale" color – black, white, or any shade of gray in-between. This shortcut color function will do that for us:

```
void setGreyScale(float bright) {
  GetComponent<Renderer>().material.color =
         new Color(bright, bright, bright);
}
```

`setGreyScale(0.25f);` would give a grey that barely stood out from a black background; `setGreyScale(0.9f);` would turn us almost white.

Like any other functions, functions with inputs can also do math. Suppose we have a lot of artist-programmers who prefer the standard 0-255 `ints` for color values. We can write a function that turns us red, using 0-255 for how red we should be:

```
void setRed(int amt) {
  float r = amt/255.0f; // 255.0f must be a float, to keep the fraction
  GetComponent<Renderer>().material.color = new Color(r, 0, 0);
}
```

Now `setRed(80);` turns us a darkish red, and `setRed(240);` is almost maxed-out red (if you've used image programs, those numbers would seem completely natural.)

Here's a longer example with a `string` input. I want the function to move into a corner of the screen, using `"UL"`, for upper-right, and `"LL"`, `"UR"`, and `"LR"` for the other corners. Here's the heading:

```
void moveToCorner( string cornerName ) {
```

My plan is to figure out the left&right and top&bottom numbers. I'll only need 2 of them, but it's easier to compute them all. Then I'll check the input and move us to whichever corner. This has got local variables, calculations and so on. By now we know functions can have all that:

```
void moveToCorner( string cornerName ) {
  // figure L/R and U/D numbers ahead of time:
  float left=-5.0f, right=5.0f;
  float down=-3.0f, up=3.5f;

  float xx, yy; // set these to the position we want

  if(cornerName=="UL") { yy=up; xx=left; }
  else if(cornerName=="LL") { yy=down; xx=left; }
  else if(cornerName=="UR") { yy=up; xx=right; }
  else if(cornerName=="LR") { yy=down; xx=right; }
  else { xx=0; yy=0; print("bad cornerName"); }

  transform.position = new Vector3( xx, yy, 0);
}
```

For fun, I decided that bad inputs, like `"topEast"`, put us in the middle and print an error message. That's not great, but what else can you do?

To finish it, some function calls:

```
moveToCorner("LL"); // puts us in lower left
moveToCorner("ur"); // Oops! puts us at 00 -- string == is case-sensitive
moveToCorner("UR"); // puts us at (5, 3.5)
```

The second call is why we don't like to use strings as instructions – too easy to spell them wrong.

Now some bonus function calls to show a new rule:

```
string cWord = "LR";
moveToCorner(cWord); // put us in lower-right corner
moveToCorner("L"+"R"); // also in lower-right corner

string LorR="R";
moveToCorner("U"+LorR); // "UR"
```

We're allowed to use math inside of a function call – `+`, variable look-ups, any math we can think of. In fact, we've been doing it all along. `print("n is "+n);` computes `"n is 6"` and then calls print.

12.2 Rules

Most of these are just formal ways of saying stuff you already know, but some of it might be new. Plus it's just nice to see it written up:

- In a function call, the input goes inside the parens, the same as `print("a");`. Our functions from before, like `applyPos()`, also follow this rule – the inputs (none) are inside the parens.
- The input when you call a function is called an **argument**. The input the function asks for is called a **parameter**. In `showAsWord(1);`, 1 is the argument for the parameter `int n`.

  These words only matter because the computer uses them, a lot, so they're good to know. A bad argument means the call used a wrong input. If a local shadows a parameter, that's inside the function. They don't really relate to normal english usage. They often get shortened to `arg` and **parm** (parm isn't even the first four letters of parameter, but oh well.)
- The argument (the thing in the function call parens) can be anything that counts as the correct type. `showAsWord(1+b*6)` is fine (if `b` is an `int`.) It will fix any wrong types that `=` would fix, so 3 is fine for a `float` parameter.
- Function calls have to have the exact number and type of arguments. These are all errors: `showAsWord();`, `showAsWord(1.3f);`, `showAsWord("1");`.

  Another way of seeing this, you can only run the function if you give it usable inputs.
- To make a function have a parameter, declare a variable in the heading, like `void A(string w)`. Can use any type, and any name.
- After the function starts, the parameter counts as an ordinary variable. You can add to it or change it like anything else.

Again, just a summation of the rules from the examples, but I tried to use the official words. Plenty of rules are like this – pretty intimidating until you understand them.

12.3 More rule-showing examples

The name of the parameter really is just any name you pick. My original `showAsWord` used `n` as the parameter name, but anything else would work. This looks worse, but works fine – all that matters is the code is reading the input variable:

```
void showAsWord(int xyz) {
  if(xyz<=0) print("zero");
  else if(xyz==1) print("one");
  ...
```

The style rules for parameter names are the same as for any variables: try to pick one that says what it does, or just something boring like `n`, `w`, or `x` if you can't think of a good name.

Once the function is running, the parameters count as regular variables. You're allowed to change or copy them if you like. In this example, I decided to "fix" the input before using it:

```
void showAsWord(int n) {
  if(n<0) n=0; // <-- can change input n
  if(n==0) print("zero");
  else if(n==1) print("one");
  ...
}
```

If I call `showAsWord(-4);`, it starts with `n=-4;`. But then the first line changes `n` to 0, which is legal, and seems fine.

After they get the inputs, those variables don't have any special "input goodness." In this example, I picked a long variable name for the heading – hoping it makes it easier for people to see what the function does. But I want to use a shorter name inside, so I copy it to `x`:

```
void showAsWord(int numToPrintAsAWord) {
  int x = numToPrintAsAWord; // copy to a shorter-name var
  if(x<=0) print("zero");
  else if(x==1) print("one");
  ...
}
```

You're even allowed to ignore or destroy the inputs if you want. In this example, I want to test how the function works on 29. This temporarily ruins the function, but it's legal, and is fine for testing:

```
void showAsWord(int n) {
```

```
  n=29; // <- testing line
  if(n==0) ...
  ...
}
```

This one simply ignores the input, so is useless, but it's also legal. And also the kind of thing people write for testing:

```
void showAsWord(int n) {
  print("NUMBER");
}
```

Sometimes we call that a *Stub*. The idea is: if you want to finish a function later, as least make the heading correct. That way everyone can write their parts using it.

The parameter name really is a local variable, only inside that function. In this example, Start can also have an `n`:

```
void showAsWord( int n ) { ... }

void Start() {
  int n=47; // <- a different n
  showAsWord(1); // "one"
  print("n="+n); // our n is still 47
}
```

Start gets to have an `n` with no surprises. As usual, the `n` inside the function is completely different. It's just an unimportant coincidence the names happen to be the same.

12.4 Fun errors

I mentioned some of these before, but here are almost all the function errors and error messages. Same idea as before – if we cause them on purpose and read them, they won't surprise us when we see them for real:

- Input when we shouldn't have one: `applyPos("carrot");` gives error: *No overload for method 'applyPos' takes 1 arguments.* This is just telling us that `applyPos` (from last chapter) doesn't take any inputs. Method is another word for function and remember argument is the technical term for the input when you call it.
- Missing an input is the same error: `showAsWord()` says: *No overload for method 'showAsWord' takes 0 arguments.*

- The wrong `type` of input: `showAsWord("hat")` gives *The best overloaded method match for 'testA.showAsWord(int)' has some invalid arguments.* It's just showing you the function, and how it wants an `int`. A funny thing is, other errors like this would say "cannot convert 'hat' from string to int." This error is more like "here's how it should look – yours doesn't match."

- Adding the type in the call: `numAsWord(int 4);` is an error. It says *unexpected symbol 'int'.* It's easy to get the function call and definition confused. The definition has to give the type, the call just has to match it. Another way to remember is you wouldn't write: `n = int 4;`.

- Declaring the parameter also as a local variable:

  ```
  void A(int n) {
    int n=4;
    print("input is " + n);
  }
  ```

  This gives you: *A local variable named 'n' cannot be declared in this scope because it would give a different meaning to 'n', which is already used in a 'parent or current' scope to denote something else.* You can't have two local variables both named `n`.

  The error looks so odd because parameters aren't exactly local variables.

12.5 Multiple Inputs

Functions can have 2 or more inputs. There are only a few extra rules for this, which aren't complicated. Here's an example of a 3-input mad-lib function and some sample calls:

```
void sayStory(string animal, string animalMood, int howMany) {
  string w;
  if(howMany==1) w = "I saw a " + animal + ". It was "+animalMood+".";
  else w = "I saw " + howMany + " " + animal+"s. They were "+animalMood+".";
  print(w);
}

void Start() {
  sayStory("cow", "lazy", 1);
  // I saw a cow. It was lazy.
  sayStory("whale", "happy", 17);
  // I saw 17 whales. They were happy.
}
```

The two new rules are: put commas between them (both places: the function call, and the definition.) Rule two: when the function is called, match args to parms exactly in order.

If you remember, that clunky `applyColor();` used global variables. We can finally make the good version, where we directly give it the color values:

```
void colorChange(float r, float g, float b) {
  GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

To make orange we can use `colorChange(1, 0.5f, 0);`. This is a good solid use of the "make new commands" idea of functions – one short all-in-one command to fully set your color.

Here's one for resizing that does a little extra thinking for us. A basic resize ends with `=new Vector3(wide, tall, 1);`. The last slot, `z` is almost always 1 (it's deepness, which we can't really see,) but we still have to enter it.

We can write a function that "knows" our rules, filling in that 1 automatically:

```
void resize(float wide, float tall) {
  transform.localScale = new Vector3( wide, tall, 1);
}
```

We'd use it like `resize(2, 0.5f);` to make us wide and short. We don't have to type the longer localScale line and we don't have to think about the thickness. That's a good deal.

Just to show `float` and `string` inputs together, here's an odd resize. It's a little fakey: I want to send the overall size, then a letter whether it's a square (`"S"`,) a tall rectangle (`"T"`) or a wide rectangle (`"W"`):

```
void shapeResize(string shape, float sz) {
  float wide=sz, tall=sz; // so far, may change if not square

  if(shape=="W") { tall/=2; } // wide - chop height in half
  else if(shape=="T") { wide/=2; } // tall - chop width in half
  // else square, so what we have is fine

  transform.localScale = new Vector3( wide, tall, 1);
}

// sample calls:
shapeResize( "W", 5); // giant sideways bar
shapeResize( "T", 1); // smallish "standing" block
shapeResize( "S", 0.2f); // tiny square
```

Like I wrote, this is a bit silly – I just wanted to show we could use two different types of inputs.

12.5.1 More errors

Arguments really are always matched to parameters in the exact order. That means we can now have some new, fun input out-of-order errors. `sayStory` takes inputs in order *(string, string, int)*. Putting the `int` somewhere else is an error:

```
sayStory(5, "dog", "barky"); // ERROR. Wrong order
```

The computer tries to match 5 with `string animal`. It can't, so it's an error.

In our minds, we accidentally put 5 in front, instead of at the end. But the computer won't even think about different orders. As soon as it sees that the 5 doesn't match, it stops checking and gives two errors: *The best overloaded method . . . has some invalid arguments* and then more detail: *Argument #1 cannot convert int expression to type string.*

You have to give the exact number of inputs. Too many or too few is an error. Both of these give a *No overload for method takes __ arguments* error:

```
// sayStory("cow", "lazy"); // ERROR. missing one
// sayStory("cow", "lazy", 5, 7); // ERROR. one extra
```

You might think the second one should just skip the extra 7. We decided it's an error so you have a chance to clear it up. Maybe you meant to write 5+7?

Of course, the computer only checks the types. It has no way to make sure the first input is an animal. If you flip the order of two `string`s, it won't cause an error, but will probably be wrong:

```
  sayStory("sleepy", "chicken", 3); // Ooops:
  // I saw 3 sleepys. They were chicken.
```

You aren't allowed to use the multiple variable shortcut for parameters. You have to list out each type/name pair. This is an error:

```
void setPos(float x, y) { // error, missing second float
  transform.position = new Vector3(x,y,0);
}
```

The computer can't tell whether `y` is supposed to be a float, or if you forgot to put the type in front. My compiler tells me *identifier expected*, which isn't very helpful, but at least it tells me the line.

The correct version is `void setPos(float x, float y)`.

Chapter 13

Casting and `char`

This section is about two things that aren't super important. It's here as a little break. One is the official way to say "turn `int n` into a `string`." The other is a new type for just one letter, which we won' t need until we look inside of `strings`.

I won't use either of these for a while – you could skip to the next chapter on more functions, if you wanted.

13.1 casting

We know that in `3+2.2f` the computer has to convert the 3 into `3.0f`. Likewise `"b"+4` requires the 4 be converted into a string. The official word for that is **casting**. As a verb: "4 is cast into a string".

The fun thing is, we don't have to just let the computer cast because it needs to. That's called an **implicit** cast. There are commands that explicitly tell the computer to convert one type into another.

The rule for an *explicit* cast is to put a type in parentheses before the value, like `(int)` or `(float)`. It will try to change the next thing into that type. Some examples (these are explained more in their sections):

```
print( (int)3.7f ); // 3 -- turns 3.7 into an int
float f = (float)5/4; // 1.25 -- turns 5 into float 5.0f
```

The parens around the type aren't math parens – they're special required cast parens. As usual, you can add more math parens if you need to: `((float)(5/4))` has one set of required cast parens, and two sets of math parens.

You can attempt to cast from any type to any other. Some work, some don't, some work in a funny way. The exact rules aren't that important to know – feel free to skim. If you sort of get the general idea, that's fine.

13.1.1 `(int)` from float

Using the `(int)` cast on a float drops the fraction. For example `(int)5.9f` is 5. Some examples:

- `(int)2.9f` is 2.
- `(int)0.3f` is 0. This is the same rule, I just wanted to show it even drops the fraction after a zero.
- `float r = (int)2.9f;` makes `r` be 2.0f. Like all math, it goes one step at a time and won't look ahead. So first we turn 2.9 into 2. Then we make it `2.0f` so it can go into `r`.
- `int n = (int)3.7f;` is legal, and makes `n` be 3. The key is, it doesn't just drop the fraction, it turns it into an integer.
- `int n=(int)9.0f;` is legal (but very silly.) It makes `n` be 9. Technically, it drops the point-zero fraction. You can think of it as perfectly turning `9.0f` into an `int`.
- `int n = (int)f;` is legal. Casting works on variables.
  Like other math, it doesn't actually change `f`. If `f` was 7.35, `n` would become 7, but `f` would still be 7.35.

You can use it inside longer expressions. The rule is that a cast happens before math does, but I usually use extra parens just in case. Here are some somewhat silly examples of casting in longer equations:

- `(int)9.7f/2 );` is 4. Casting goes first, so this becomes `9/2`. Those are both `int`s, so we get 4.
- `9/(int)2.9f` is also 4. The `(int)2.9f` goes first, turning it into `9/2`.
- `(int)(4.8f/1.5f)` is 3. The parens make `4.8f/1.5f` go first, giving 3-point-something. Then the `(int)` chops it down to 3.
- `(int)4.8f/1.5f` is a funny 2.66666. The 4.8 gets int-ed, to make `4/1.5f`. But then mixed-type rules turn it into `4.0f/1.5f` and does grown-up math.
- This is kind of a long one, and just for fun. Suppose we have `float a=5.2f, b=2.8f`. Using `(int)a/(int)b` gives 2 (counts as 5/2.) Using `(int)(a/b)` gives 1 (counts as 5.2f/2.8f, which rounds down to 1.) `a/(int)b` gives 2.6 (counts as 5.2/2.)
  This is really just saying we can int-ize just `a`, or just `b`, or both, or the result.

An old trick to round to the nearest is `(int)(f+0.5f))`. It adds 0.5 first, then chops the fraction, so 4.6 become 5.1 chopped to 5; but 4.3 goes to 4.8, then gets chopped down to 4 anyway. It's very clever.

Using the same idea, `(int)(num+0.99999f)` will round up.

A cute way to round to 2 decimals is to multiply by 100, cast to an `int`, then divide by 100. It's like we slide things over, chop, then slide them back. This example shows it step-by-step, then all in one line:

```
float f = 10.0f / 6.0f; // 1.666666  starting number

// step-by-step:
float f2 = f*100; // 166.6666
f2 = (int)f2; // 166
f2 = f2/100.0f; // 1.66

// all in one line:
float f2 = ((int)(f*100))/100.0f; // 1.66
```

13.1.2 `(float)` from int

A float cast looks like `(float)7`. It adds point-zero to the end. Technically it converts `int` 7 to `float` 7.0f.

This is what we were doing automatically before, so we never really need to use it. But it can still look nice and is good for explaining things:

- `(float)3;` is just a terrible way to write 3.0f.
- We already know the trick how `1.0f*a/b` will trick ints into doing grown-up float division. `(float)n` is another way to do that: `(float)a/b`.

  Technically, `1.0f*a` causes an implicit cast, whereas `(float)a` is an explicit cast. But they both do the same thing. Most people think the `(float)` version looks better.
- For fun, you can write out every implicit cast as an explicit one:

  ```
  float f = 9/1.2f + 4/3;
  // computer turns it into:
  float f = (float)9/1.2f + (float)(4/3);
  ```

  There's no reason to do this. It's just a way to explain the automatic "turn ints into floats when you need to" rules.

You can also cast `double`'s to `float`'s (if you skipped that section or hated it, skip this one too).

Suppose we need `x` as a double for extra accuracy. Using it for position or color or size is an error, since they take floats. The computer could cut off the extra 6 decimal places, but it wants to be sure that's what you want. You have to cast it to `float`:

```
double x; // why not a float? Pretend there's some reason
  ...
transform.position = new Vector3( (float)x, 0, 0);
```

13.1.3 `string` casts, conversions

There is a way to turn `string`s into numbers, but it's a little funny.

Casts with strings are disabled. In other words `(string)4` and `(int)"37"` should be legal, but aren't, just because. The error messages all sound like "I know you're trying to cast, but the one you're trying isn't allowed."

These are all proper uses of a cast, but all give errors:

```
int n= (int)"4"; // cannot convert string to int
string w= (string)7; // cannot convert int to string
float f= (float)"3.6"; // cannot convert string to float

int n= (int)"abc"; // cannot convert string to int
```

The last one shows the 1-step-at-a-time rule. It doesn't complain `"abc"` isn't even a number, since it never gets that far.

Instead of a cast, there are functions. This turns a string into an int:

```
int n = System.Convert.ToInt32("17"); // n is 17
```

The name of the function is `ToInt32`. It's in the namespace `Convert`, which is inside the `System` namespace. We haven't seen the rule yet about a function having an output – the part where `n=` is legal, but we will, next chapter.

We can already implicitly cast ints to strings with `""+n`, and that works fine. But there's also a function for it:

```
string w = System.Convert.ToString(n); // same as w=""+n;
```

Note how it's also in the `Convert` namespace. It also works for float to string.

13.2 `char` type

Back in chapter 3, you may have noticed that strings are much more complicated than ints and floats. The basic type that really goes with those two is **character**,

which is one "keyboard symbol." Strings are really lists of characters. For example, strings "cats" and "1?->" are each 4 characters.

Characters are primitive, compared to strings. They're good to see, since they're a common, standard type. We're also going to need them later, when we look inside of strings, and they also have some really fun casting rules.

But, again, these are really just details. At some point, you should know what characters are, but the rest of this is just for fun.

13.2.1 char examples

The official name is char (pronounced "care", like the first syllable of character.) A character literal has single quotes around it. On a standard keyboard it's just under the double-quotes (not the slanted one on the upper-left. That's called a "tick," and isn't used for anything.)

Here's some character use:

```
char c1 = 'y', c2 = '['; // anything on the keyboard is a character
char c3 = ' '; // even a space

if( c1 == 'g' ) {} // can use regular compares
if( c1 != 'u' ) {}
```

Characters have to be exactly one letter. They can never be 2 or more, or empty. This is the rule that keeps them simple. A character is one box, holding one letter. It's as basic as an int:

```
char ch = 'ab'; // ERROR -- Too many characters in character literal
ch = ''; // (2 single quotes, no space) ERROR -- Empty character literal
ch=' '; // 1 space. This is fine.
```

As you might guess, the **type** rules won't let you mix strings and chars. These next examples are all things the computer could do, but instead it gives horrible errors about type mismatches:

```
char ch = "a"; // ERROR -- can't assign string to character
string w; w='a'; // ERROR -- can't assign character to string

if(ch == "x") {} // ERROR -- can't compare string to a character
if(w=='a') {} // ERROR -- can't compare character to a string
if(w==ch) {} // same error
```

The one way it will mix them is the same way it mixes ints, floats and strings. + will auto-convert a char into a string:

```
string w="horse"; w += 'y'; // horsey
char ch='a'; string w = ""+ch; //  "a"
```

Sometimes you want a character which you can't type, like a return, or a double-quote inside of a string. A standard way to make those characters is to use an **escape sequence**. For example, `print( "***\n***" );` will print two lines of three stars.

The slash says to begin an escape sequence, and `n` is the escape code for newLine. Together \n is one character. The internet does a great job of listing the tables, with examples.

13.2.2 Casting `chars`

The most important thing about this section is that you *really* don't need to know it. It's just for fun.

Computer can't really store letters. That's probably obvious. They store letters as numbers, using a chart. `char ch='a';` puts 97 into `ch`. The computer will gladly print it as 97. Likewise, if you give the computer a 98, it can use the chart to see that's `'b'`.

The chart is the ASCII chart, which is part of the larger Unicode chart (you can find a copy on-line). Some interesting values from it: A-Z are 65-90, a-z are 97-122, and the keys `'0'-'9'` are 48-57. Basically, we took the numbers 0-255 and arranged every letter, upper-case letter, digit, and punctuation symbol. There's no perfect spot for things, so we put them where-ever. Some early versions didn't even have all of the letters in a row.

Examples of char casting:

```
int n='a'; print(n); // 97
n='3'; print(n); // 51
```

The computer is happy with this. `'a'` actually is 97. The computer is glad to put in into an `int`. The `'3'` is a little sneakier. We put in between single-quotes, which means it's a char, which means we use the chart and it's the number 51.

This even works with math:

```
print( 'a'+2 ); // 99 (97+2)
print( 'd'*2 ); // 200 (d is 100; 3 steps from 'a')
print( '2'+12 ); // 62 (since '2' is 50)
```

An explicit cast can go the other way, forcing the computer to look up the number as a letter:

```
print( (char)100 ); // d
print( (char)65 ); // A
print( (char)49 ); // 1 <- this is really '1'
```

One use for this trick is making special characters which can't be typed. 169 is the copyright c-in-a-circle symbol. `w="Cat crammer"+(char)169;` will add that symbol to the end of your invention.

Some fun tricks we can do are checking whether a character is a lower-case letter. We're really comparing numbers, but it looks nicer writing them as letters:

```
if(ch>='a' && ch<='z') print("lower case");

if(ch>='A' && ch<='Z') print("upper case");
if(ch>='0' && ch<='9') print("ch is a digit");
```

We can turn `'0'` through `'9'` into a real 0-9 by subtracting the code for `'0'`:

```
int num;
if(ch>='0' && ch<='9')
  num=ch-'0'; // ex: '3'-'0' is 51-48 is 3
else
  num=-1; // not a digit
```

This update loop will print the alphabet. We start at `'A'`, add 1 each update, and repeat after we hit `'Z'`:

```
int x='A'; // starts x at the number code for A (which is 65)

void Update() {
  print( x + " " + (char)x ); // ex: 65 A
  x++;
  if(x>'Z') x='A';
}
```

This abuses implicit casts from char to int three different places.

Chapter 14

Functions, part III - return

14.1 Introduction

You may have noticed that several of the old functions computed something and then just printed the answer: the ice/water/steam function, or finding the word for a number, or even the random New Clarkville town generator. Printing the answer is just terrible. We want them to *be* the answer. The way 2+3 is 5. we want `waterState(25)` to just be `ice`.

Our goal is to be able to do things like this:

```
string w=waterState(20);
print("The lake is "+w); // The lake is ice

string townName=getRandomTown();
print(townName+" was overrun by rats"+); // Tomaberg was overrun by rats
```

14.2 return values

Our old functions did things. They went all by themselves on a line, like `applyColor(1.0f, 0.5, 0);`. applyColor doesn't have an answer or a value, and we wouldn't want it to.

Our new functions will be part of longer lines, and will work like math:

```
int n = 2+3; // 2+3 turns into 5
string w = waterState(25); // waterState(25) turns into "ice"
```

We already have the rule that functions come back to where you called them. Now that rule means we can pause midway through a line. Above, `string w=` is waiting for `waterState(25)` to come back with the result.

The new rule for this is the `return` statement. It quits the function and returns the answer. Here's the improved `waterState` function, using a return:

```
string waterState(float tempInDegs) {
  string w;
  if(tempInDegs<=32) w="ice";
  else if(tempInDegs<212) w="water";
  else w="steam";

  return w;
}
```

The print statement at the end is gone, Instead we have `return w;`. The only thing it does is send back the answer.

Let's to a quick run-through. `s=waterState(31);` marks where we came from. We're midway through the line, with `s=` waiting. We copy 31, run waterState, return `"ice"` and pop back. Our program now counts as `s="ice";` and keeps running from there.

There's one more change. The very first thing in the function, before the name, is the type of the return value. It replaces the `void`. In this case, it clearly has to be `string` since the 3 possible answers are strings.

Those are the only changes: put the return type in front, and use a `return` statement at the end.

The turn-int-into-a-word function can be rewritten. It's not very exciting, except for how it now works properly. `string w=intToString(2)` now gets `"two"`:

```
string intToString(int n) { // <- string return value
  string numWord="unknown";
  if(n==0) numWord="zero"; // this is code from the if chapter,
  if(n==1) numWord="one"; // nothing special about it
  if(n==2) numWord="two";
  // ... until 9
  return numWord;
}
```

The 90/80/70/60/50 letter grade `if` is pretty much the same. The answer is one letter, so a function can return it:

```
string letterGrade(int n) {
  string gr;
  if(n>=90) gr="A";
  else if(n>=80) gr="B"; // copied straight from the...
  else if(n>=70) gr="C"; // ...chapter on cascading ifs
  else if(n>=60) gr="D";
  else gr="F";

  return gr;
}
```

`string grade1 = letterGrade(72);`. is now allowed. `grade1` is a B.

We think of these as "pure" functions – basically like math. They get all input from parameters, and don't change anything in the program. They just give the answer back and let the main program use it however it likes. These are considered the best type of function.

A fun thing is to write out some real math functions. Some of these are built-in but they're fun to write anyway.

Finding the largest of two numbers is usually called `max`. This function is the same `if/else` from a few chapters ago, with a `return`:

```
float max(float a, float b) { // <- answer will be a float
  float ans;
  if(a>b) ans=a;
  else ans=b;
  return ans; // <- no printing. return the answer
}
```

If we have `float q=max(3,8);`, then `q` will be 8.

This is our first `float` return-type. It makes sense, since the answer is one of the inputs, which are floats. It's also our first returning function with two inputs. A `return` can only give one answer, but a function can always have all of the inputs it needs.

Absolute value (`abs`) is a fun, short function:

```
float abs(float a) {
  float ans=a;
  if(a<0) ans=a*-1;  // if it was negative, make it positive
  return ans;
}
```

`abs(6);` is 6 and `abs(-12)` is 12. This function barely does anything – it's one `if` that we could write ourselves. But calling `abs(n)` is shorter and easier to read, so this is a good function.

14.3 Using value-returning functions

In our minds, value-returning functions work like math. Since they don't do anything, we can run them whenever we want, saving the answer for later. This prints comments about 2 grades:

```
string gw1=letterGrade(g1), gw2=letterGrade(g2);

if(gw1=="F") print("Arrg! I can't read any more!");
```

```
else {
  if(gw2=="F") print("Arrg! So close.");
  else print("Yay! Passing!");
}
```

We might not even use `gw2`, but it seems easier to compute them both at the top, the same as we like to do with all of our math.

This uses `max` to figure out if anyone is old enough to rent a car:

```
int oldest = max(amyAge, bradAge);
if(oldest>=18) print("Can rent a car");
```

We're allowed to use these functions inside of larger math. These two add waterState to another string:

```
print("Look, it's some " + waterState(10)); // Look, it's some ice

string w2 = waterState(270) +" world"; // steam world

print("H2O at "+ t1 + " degrees is " + waterState(t1) );
```

There aren't any special rules here. `waterState(270)` always becomes `"steam"`, no matter where it is. `4+waterState(0)+9` is `4ice9`, and so on.

These use our new math functions inside a larger equation:

```
n = max(3,6) + abs(-2); // 8 (6+2)
n = 2*max(-12,3)+100; // 106 (2*3+100)
```

Using the same function several times works just fine. The computer calls them in order, replacing values as it goes:

```
n = max(3,7) + max(1,-6) + max(2,2); // 10 (7 plus 1 plus 2)
n = abs(3) * abs(-5); // 15 (3 times 5)

n = max(10,2) * abs(-3) + abs(5); // 35 (10*3 + 5)
```

Here's a picture of the steps in the last one:

```
n = max(10,2) * abs(-3) + abs(5);
// makes three function calls:
// n =  10    * abs(-3) + abs(5);
// n =  10    *    3   +  abs(5);
// n =  10    *    3   +    5;
```

To really check this out, we can add a `print` inside of `max`. That's a terrible idea for real, but good for testing:

```
float max( float a, float b ) {
  int ans;
  if(a>b) ans=a;  else ans=b;
  print("In max. a="+a+" b="+b+" answer="+ans); // testing!
  return ans;
}
```

Now we can run a line with two `max` function calls and watch them run:

```
print( max(2,6) + max(1,1) );
// In max. a=2 b=6 answer=6
// In max. a=1 b=1 answer=1
// 7  <-- the print we wrote
```

A fun and completely legitimate trick is using functions inside of an `if`. Some examples:

```
if( max(a,b)>10) ) print("both must be 10 or less");

if( waterState(temp) == "ice" ) print("can walk on it");

if( waterState(t1)=="ice" || waterState(t2)=="ice")
  print("at least one is ice");
```

The neat thing is, it's still not a new rule. Same as before, functions are called, run, and count as their answer. Then the line continues.

14.4 More math functions

Here are some more examples of math-like functions.

`sign` returns -1, 0 or +1. The idea is, it really says negative, positive or zero, but -1/0/+1 is the best way to say that. As usual, the inside is completely boring, except for the `return`:

```
int sign(float a) {
  int ans;
  if(a<0) ans=-1;
  else if(a>0) ans=+1;
  else ans=0;
  return ans;
}
```

For examples `sign(6)` is 1, `sign(-0.43f)` is -1.

Previously we saw a pair of `if`'s to make sure a number is between 1 and 10, or some other range. We can put that in a function:

```
float clamp(int n, int min, int max) {
  float ans=n; // so far
  if(n<min) ans=min;
  else if(n>max) ans=max;
  return ans;
}
```

A common use would be `cats=clamp(cats,1,10);`. If cats was outside of 1-10, that line fixes it. If cats was 7, the answer is 7 and it changes cats to 7, which seems odd, but it works.

It feels so much like a real math command that people use the word clamp to mean "adjust into a range"

We often need to get a percent from one number to another. For example 10% of the way from 4 to 7. In math that's a linear interpolation, abbreviated lerp. The math is easy, but we often write it as a function:

```
float lerp(float start, float end, float pct) {
  // NOTE: pct should 0 to 1. 0.3 is 30%
  float gap=end-start;
  float distFromStart = gap*pct; // ex: 10% of the way
  float ans = start+distFromStart;
  return ans;
}
```

A sample use, our screen from the Cube examples goes from -7 to 7. To put something 30% of the way across, we can use `float xx=lerp(-7, 7, 0.3f);`.

Squaring a number is super-easy, just one line, and a new trick:

```
float square(float n) { return n*n; }
```

Notice how we did math after the `return`. That's completely legal and works the normal way. We usually think it looks better putting the answer into a variable, unless the whole thing is one equation like here.

To see why this might be useful, consider squaring x+y. `square(x+y)` looks nicer than `(x+y)*(x+y)`.

We aren't limited to real math. Suppose your magical manna points are your Intelligence stat or Wisdom stat, which-ever is higher, plus half of the other one, but at least 5. We can write that:

```
int mannaPoints(int intell, int wis) {
  int ans;
  if(intell>wis) ans=intell+wis/2;
  else ans=wis+intell/2;
  if(ans<5) ans=5;
  return ans;
}
```

Even though this is totally made up, it works the same as any other math function. `int myManna = mannaPoints(10,5);` gives 12. `mannaPoints(1,2);` is 5.

In the old days if you wanted to know how a game worked, you looked through the code for things like this. Some places called the mannaPoints function, you searched for it, and voila – now you know the formula the game uses for manna.

14.4.1 Extra return rules

The `return` statement jumps out of the function even if it's not at the end. The function quits right then – any lines after the `return` are skipped. That should make perfect sense – once you have the answer, the function is done.

You're allowed to have as many `return` statements as you need, and they can have any kind of constant or equation after them. Here's `intTostring` rewritten using the "return as soon as you know it" method:

```
string intToString(int n) {
  if(n==0) return "zero";
  if(n==1) return "one";
  if(n==2) return "two";
  return "three";
}
```

That `return "three"` at the end looks mighty suspicious. There's no `else` in front of it, so it looks like the answer is always `"three"`. But the function works by abusing the `return` rules. If `n` is 0, the function quits right then with answer `"zero"`, and never gets past line one. It only gets to `return "three"` if the `if`s were all false.

Here's `max` rewritten to have return-abuse:

```
float max(float a, float b) {
  if(a>b) return a;
  else return b;
}
```

It doesn't even have a final `return` by itself, since the if/else always does it.

You can use the return-early trick for testing. Here we adding an extra line to temporarily make `intToString` always say NUMBER_WORD:

```
// abusing return for testing:
string intToString(int n) {
  return "NUMBER_WORD"; // temporary for testing
  // this is all skipped:
  if(n==0) return "zero";
```

```
  if(n==1) return "one";
  if(n==2) return "two";
  return "three";
}
```

Using the trick for mannaPoints shows why we have to be careful with it. Our answer may not be completely done:

```
int mannaPoints(int intell, int wis) {
  int ans;
  if(intell>wis) return intell+wis/2;
  else return wis+intell/2;

  // oh, no! Those returns skipped the "at least 5" rule
  if(ans<5) ans=5;
  return ans;
}
```

14.4.2 Optional empty return

If you have a function with no return type (it has `void` in front,) you're allowed to use just `return;` (with nothing after it) to jump back. You can put it at the end, for fun, and can also use return-from-middle.

Here's just a decorative end `return`. It's a little like writing The End:

```
void reset() {
  x=-7; y=0; speed=0.05f;
  return; // not needed, but looks pretty
}
```

Here's the early-return trick on a wrap-around function. It uses the `return` to quit and not wrap if it doesn't need to:

```
void wrapAround() {
  // assuming x is a global
  if(x<7) return; // not too big, so just quit
  x=-7;
}
```

You can even use return-from-middle in `Start` or `Update` – they're functions with `void` return values. Here's working move&wrap code with return-from-middle:

```
public float x=-7;

void Update() {
  x=x+0.1f;
```

```
  transform.position = new Vector3(x,0,0);

  if(x<7) return;
  x=-7;
}
```

Normally, `Update` quits when it gets to the end. Quitting early does the same thing, only faster. Either way, the "auto-run over and over" rule still happens.

A use of this trick is to crudely comment out code. This causes Update to do nothing:

```
void Update() {
  return; // for testing. update quits here
  x=x+0.1f;
  ...
}
```

Another legitimate use might when something is killed. We can do death stuff to it, then `return` to make sure nothing else happens:

```
void Update() {
  ...
  if(health<=0) {
    // set color to black
    // do other death stuff
    // now prevent the rest of Update from changing color or position:
    return;
  }
  ...
}
```

Obviously return-early can cause lots of confusion if used too much. Your update might have a line adding to `x`, but `x` is not changing. After an hour you spot the `return` up above. The compiler gives a warning when a naked return cuts off the rest of a function, but not for one in an `if`.

14.4.3 `return` can use math

Sometimes it looks nicer to put equations after a `return`. Like anything else, the computer works them out first – `return 1+3;` is the same as `return 4;`.

Some examples. This `abs` use two returns, one of them using math:

```
float abs(float a) {
  if(a<0) return -1*a; // could also use -a
  return a;
}
```

This redo of `lerp` does the math in the return line. As usual, it does the math first, then the `return`:

```
float lerp(float start, float end, float pct) {
  float distBetween = end-start;
  return start + distBetween*pct;
}
```

This mad-lib sentence maker builds a giant string in the `return`:

```
string walk1(string animal, string place) {
  string action="walk";
  if(animal=="fish" || animal=="dolphin") action="swim";
  return "My "+animal+" and I went for a "+action+" in the "+place;
}
```

14.4.4 Errors

If you say you're going to return something of a certain type, you have to. And it you say you won't return anything (by using `void`,) you can't. That seems pretty obvious – a function breaking that rule won't even make any sense. But there are some funny cases and seeing the error messages is helpful:

It's easy to write a function that usually returns something, but for some values it just reaches the end without ever hitting a `return`. For example this gives compiler error: *not all code paths return a value.* because of 10 through 100:

```
string badFunction(int num) {
  if(num>100) return "big";
  if(num<10) return "small";
  // return ""; // this would fix the error
}
```

It's a compile error – you can't even run the program. Maybe you just forgot. Or maybe you know the input can never be 10-100. The fix is adding a dummy return at the end. Sometimes I'll use something like `return "BAD VAL";`.

This one's a little tricker. We can see it always `return`'s something, but the computer can't tell. It incorrectly gives the *not all code paths return a value* error:

```
string sayNum2(int num) {
  if(num<0 || num>1) return "bad";
  if(num==0) return "zero";
  if(num==1) return "one";
}
```

Adding `return ""; // can't get here` as the last line will make the compiler happy. Or, rewrite it as a cascading if/else.

When using `return;` to quit early from a no-answer function, it's easy to accidentally write `return 0;`. Zero seems like nothing, but it's not nothing – it's a number that means nothing (who knew computer science was so philosophical?):

```
void moveAndWrap(float n) {
  n+=0.1f;
  if(n<7) return 0; // ERROR, use "return;"
  n=-7;
}
```

Returning the wrong type is clearly an error. But the computer will do any conversions that an `=` would do. `return 3;` is fine when you need a float.

Functions starting with `void` don't have answers. Using them inside of math gives an error. This function prints a string, but it has no official answer, so trying to assign from it won't work:

```
void sayMoo() { print("mooo"); }

  string w = sayMoo(); // ERROR
```

It says: *Cannot implicitly convert type void to int.* That's not quite right, but it lets you know where the problem is.

Oddly, it's *not* an error to ignore a return value. This is legal, but useless:

```
max(3,15); // not an error. Answer is 15, but we don't use it for anything
```

This might be useful for testing. You might have a bunch of printing lines inside and just want to run it to check what they say.

Chapter 15

Constants and enums

This section is another little vacation about cutesy language features. It's fine to skim or skip. The main reason to know them is when you see them in other people's code.

You can use these tricks, but they're sort of like `+=`. They don't do anything you couldn't do before.

15.1 Constants

A trick we'd like to borrow from math is using letters for constants. For example, we might want to make `cpi` a shortcut for 2.54 (centimeters per inch.)

We could just write it like this:

```
float cpi = 2.54f; // DON'T CHANGE! centimeters per inch
```

But the program might be easier to read, and a little safer, if we had a rule to lock `cpi` to that value, and make it an official constant. We do. The rule isn't a huge improvement, but it's short and easy to remember.

When you declare a variable, putting `const` in front locks it to that value. You have to use the declare-and-`=` trick when you do this. Examples:

```
const float cpi=2.54f;
const int SecsPerMin=60;
const float WaterFreeze = 32.0f;
```

You don't have to use them only for "real" things. You're allowed to use the rule for anything:

```
const string HomeTown="Townville";
const float XMin=-7, XMax=7; // both are locked
const int abc=12;
```

I can't think of any possible reason anyone would want to have `abc` as a locked-in shortcut for 12, but it's legal.

You use them like regular variables, except you can't change them (trying to is an error):

```
if(temp>WaterFreeze) ...
float x = Random.Range(XMin, XMax);
print("You wake up in "+Hometown+".");

secsPerMin=45; // ERROR
// The left-hand side of an assignment must be a variable, a property or an indexer
```

The advantages are minor. They look a little different in the pop-up (mine have an orange F next to them) and we can't accidentally change them. The main advantage is making the program a little easier to read.

You can declare `const`'s locally. This code checks whether we're off the sides, with an extra 0.02 unit tolerance:

```
void fixRange() {
  const float EDGE_GAP=0.02f; // how far from edge counts as out-of-bounds
   if(x<XMin+EDGE_GAP {} // do off left edge stuff
   if(x>XMax-EDGE_GAP) {} // do off right edge stuff
}
```

Using all caps and underscores is an older style for writing constants.

Constants have one other small, tiny advantage – they run just a tad faster. The compiler replaces them with the numbers. When your program runs, there's no variable look-up needed.

15.2 enumerated types

Sometimes we use `int`'s to stand for an option, in a clunky sort of way. For example, we might make a variable holding different types of cars, set up like this:

```
// Car Types: 0=sedan, 1=compact, 2=convertible and 3=van
int car1type; // use the car table
car1type=3; // car1 is a van
  ...
if(car1type==1) // is car1 a compact car?
```

Using numbers to stand for things is a solid computer idea, but that last line is icky.

An **enumerated type** is very simple way to make this easier to read. Here's a rewrite. The first line describes a car-type to the computer:

```
enum CarT {sedan, compact, convertible, van};
```

That says we can declare `CarT` variables, which are really ints going from 0 to 3. The computer automatically numbers the words. `sedan` is 0, `compact` is 1, `convertible` is 2 and `van` is 3.

`enum` is a keyword. It's only used in this one place.

Here's a rewrite using our new `CarT`:

```
CarT car1type; // computer knows this is a 0-3 int
car1type = CarT.van; // really just 3
  ...
if(car1type==CarT.compact) ... // still comparing to 1
```

This is *exactly* the same as the previous version. It declares `car1type` as an int, assigns 3 to it, and compares it to 1. But it looks so much nicer.

If you type out the second line, `CarT`-dot even gives you a pop-up with sedan through van. It's reading them from the first `enum` line we wrote.

To show it in use, this function tells you how many doors your car has. This is the "old" version, using ints and the hand-made table:

```
int numDoors(int carType) {
  // carType should be a 0-3 number based on the carType table
  if(carType == 1 || carType==2) return 2; // compact/convert
  else if(carType==3) return 3; // van: 2 side, plus back
  else if(carType==0) return 4; // sedans are basic 4-doors
  return -1; // can't happen, but just in case
}
```

This version is exactly the same, using the exact same numbers, but using the `CarT` enum:

```
int numDoors(CarT carType) {
  if(carType == CarT.compact || carType==CarT.convertible) return 2;
  else if(carType==CarT.van) return 3; // 2 side, plus back
  else if(carType==CarT.sedan) return 4;
  return -1; // can't happen, but just in case
}
```

Notice how it takes a `CarT` input. That's a much nicer hint about how it works, even if it's still just an int.

Another, small, enumerated type example that doesn't really do anything:

```
enum bird {crow, swan, duck};
```

```
bird b1 = bird.duck; // this is really a 2
bird b2 = bird.crow; // really just 0

void Start() {
  // 50% chance b2 is really a swan:
  if(Random.Range(0, 1.0f)<0.5f) b2=bird.swan;
```

Here's a real enumerated example from Unity. You can spin a phone four different ways. 1-4, plus 0 for unknown. Unity uses an `enum` to label them:

```
enum DeviceOrientation {Unknown, Portrait, PortraitUpsideDown,
    LandscapeLeft, LandscapeRight};
```

That tell us `DeviceOrientation` is another word for int, and it has 5 possible values, with those names.

Unity creates one global of that type, `Input.deviceOrientation`, and auto-sets it as you tilt the tablet. Notice it's in the `Input` namespace.

Sample code using it:

```
// copy into var with short name:
DeviceOrientation d = Input.deviceOrientation; // really just 0-4
// portrait mode is the long way up/down:
if(d==DeviceOrientation.Portrait || d==DeviceOrientation.PortraitUpsideDown)
  print("please hold it sideways.");
```

This is really checking whether `d` is 1 or 2. But it's easier to understand with those words.

One more example, from standard `C#`. You can open a file for reading only, writing only, or both. A 0-2 int says what you want. In the old days, you wrote `open("cow.txt", 0);`. The second input was how to open it, and everyone knows 0=read only.

C# added a 0-2 enum:

```
enum FileAccess {Read, Write, ReadWrite};
```

Now we can use the nicer-looking `open("cow.txt", FileAccess.Read);`. It's extra nice because if we didn't know how the command worked, we'd see the second input is a `FileAccess` variable. Typing FileAccess-dot shows the options.

15.2.1 value returning ?: ifs

What makes `if`s useful is you can put any commands, and all you want, into each part, and you don't even have to have an `else`. But there's one common way to use an `if` that could use a short-cut:

```
string desc;
if(size>100) desc="big"; else desc="small";
```

In our minds, the `if` is really picking between `"big"` and `small"`. We're thinking of it more like this:

```
// totally illegal, but a cool idea:
string desc = if(size>100) "big"; else "small";
```

We decided to make a special type of `if` that works like that. The true and false parts are naked values, of the same type. It returns one of them. Legal code for what we were trying to do:

```
string desc = size>100?"big":"small";
```

Everything after the `=` is the special `if`. The rules are:

- It has three parts with `?` and `:` separating them. The second symbol is a colon (not a semicolon.)
- The first thing is any true/false test, the same as an `if`. The next two are possible answers: *test*`?`*answerIfTrue*`:`*answerIfFalse*
- The two answers must be the same type, but can be any type. In other words, two `int`s, or two `string`s, but not one `int` and one `string`. Ex:'s:

  ```
  int n = (x<0)?-1:+1; // n will be +1 or -1
  float g = (w=="cow")?1.5f:2.9f; // cows are 1.5, everything else is 2.9

  print( n<10?"yak":7 ); // error -- "yak" and 7 aren't same type
  ```

  You can use the int-instead-of-float shortcut, like in `float f = n<10?4:3.5f`.
- You have to give both answers. If you don't want to give one of them, you can usually put 0 or `""` to make it work. Ex:

  ```
  string w = (x>=100)?"mega": ; // ERROR what goes into w if x<100?
  string w = (x>=100)?"mega":""; // legal
  ```
- You don't need parens around the test, but sometimes it looks nice.
- It counts as math. You can put it inside a longer equation (this sounds tricky, but is a really nice feature). Ex:

  ```
  int n = 3 + (a>10)?5:1; // n is 3 plus either 5 or 1
  print("I have " + (fr==0?"no":"some") + " frogs); // I have no/some frogs
  ```

Putting it inside string math is semi-common. Two more examples of that:

```
print(num + " " + (num==1?"child":"children") ); // 1 child / 2 children

print( "I see " + (num==1?"a":"some") + animal + (num==1?"":"s") );
// I see a frog / I see some frogs
```

`num==1?"":"s"` is sneaky. We only want to print an extra `"s"` for plural, but we have to give some answer otherwise, so we use `""`.

Some general examples:

```
score += round==1?10:25; // 10 for round 1, else 25

// leap year has one extra day:
int daysInYear = year%4==0?366:365;
```

Some people use this trick for "n can't be less than 0." This looks funny but it works:

```
num = num<0?0:num; // can't be less than 0

num = num>10?10:num; // can't be more than 10
```

It can also be nested, but when they start getting that long, it might be better to rewrite them as real `if`s. This figures out "ice", "water" or "steam" based on temperature, using the same logic as a cascading `if`:

```
string  H2O = degs<=32?"ice":(degs<212?"water":"steam");

//same thing, written another way:
H2O = degs<212?(degs<=32?"ice":"water):"steam";
```

15.3 More enumerated types

The computer will sometimes allow you to mix enums and ints. They're the same thing, but not every combination is allowed.

`car1type=3;` is an error. Even though 3 is Van, it wants you to write out `CarT.van.`

You almost never want < and > to compare enums, but you can. It compares the int values:

```
if( car1type <= CarT.compact ) // legal. compares as ints
```

Likewise you almost never want to add, but you can. `car1Type++;` is legal. It goes to the next car in the list. Past Van it will go to 4, which isn't a car.

That's odd, and makes no sense, but legal.

The numbering normally starts at zero. We usually don't care about the exact numbers, so 0 is fine. But, if you want, you can pick the numbers using = after an item. Items without = go up by one. Examples:

```
enum Size {tiny, small, medium=10, hefty, big=20, huge, collosal};
//          0      1       10       11      20     21      22
```

This is a pretty obscure rule, and there's rarely a reason for it. But I like how it shows how `enum`s really are just `int`'s.

Chapter 16

Overloading and default parms

16.1 Introduction

This is another relaxing chapter, with two fun rules that let you reuse the same function name: **default parameters** and **overloading**.

They can make it look like the same function takes all sorts of different inputs, so you can use these tricks to write an `A` so that `A(3);`, `A("goat");` and `A(3,6);` are all legal.

But they're just shortcuts and tricks. They don't change the way functions really work.

16.2 Default parameters

A **default parameter** is a shortcut that lets the computer auto-fill parameters in some function calls. Here's an example of my old `setPos` function using them. The only new thing (which I haven't explained yet) are the extra two `=`'s in the heading:

```
void setPos(float x=0, float y=0) {
  transform.position = new Vector3(x, y, 0);
}
```

The first thing to know is if we call it the normal way, like `setPos(5,2);` it runs the normal way. Those two `=`'s will do nothing, like they weren't even there.

The new rule is that if the function has an *equals-something* after a parameter, we can leave it out of the function call. The computer will fill in that value. `setPos(4);` matches 4 to `x` and fills in `0` for `y`. It's a shortcut for `setPos(4, 0);`.

`setPos();` fills in `0` for `x` and `0` for `y`. It's a shortcut for `setPos(0, 0);`

The other important thing to understand is that it makes no difference in how the function works. `setPos(3);` and `setPos(3, 0);` are 100% the same thing to the computer. Both ways, `y` is 0 and the computer doesn't care how it got that way.

Here's another silly example with strings, showing how some parameters can have defaults, while others don't:

```
string adventure1(string name, string ani="dog", string place="park") {
  string w = name+" and a "+ani+" played in the "+place;
  return w;
}
```

This says if you leave out the last input, `place`, it fills it in with `"park"`. If you leave out the last two inputs, it also fills in `animal` with `"dog"`. But you have to always give one input, `name`, since it has no default. An example of every way to call it:

```
// calls:
string a = adventure1("Al", "ox", "bog"); // Al and a ox played in the bog
a = adventure1("Bev", "fish"); // Bev and a fish played in the park
a = adventure1("Carl"); // Carl and a dog played in the park
a = adventure(); // ERROR
```

There are two ways we usually use them. One is to add a very, very optional extra input. For example:

```
void setPos(float x, float y, float z=0) {
  transform.position = new Vector3(x, y, z);
}
```

In our minds, this is a 2-input function. x and y are all we change most of the time. But in the oddball case when we need to, we can throw in a value for z, like `setPos(-7,4,1)`.

The other way to see it is as a shortcut for a common value. Suppose my cube game often has `y` near the top, at `y=4`. We might write `void setPos(float x, float y=4)`. In our minds, this is still a 2-input function, but sometimes we'll write `setPos(-7);` as a shortcut for "the usual height". Even if we never do that, we still have a nice built-in reminder that the normal y is 4.

Built-in functions use lots of default parameters. The Unity spin command has a default 4th input which no one ever uses:

```
void Rotate(float xAng, float yAng, float zAng, Space relativeTo=Space.Self)
```

The fourth input has a default parameter, which means we can ignore it. `Rotate(0,90,0)` is legal. This is a case where it's telling us that if we know about rotation spaces, we can pick one. Otherwise the one it gives us is just fine.

But the fourth input is fun to look at. It's an enumerated type. We can look it up. It's `enum Space {Self, World};`. In the 4th input, `Space relativeTo=Space.Self`, the type is `Space`, which is really an int that can be 0 or 1. 0 stands for Self and 1 stands for World. Altogether it means we normally rotate in Self space, but can optionally use World space. For example `Rotate(0,90,0,Space.World)`.

If you try this, the real command has the word transform in front: `transform.Rotate(0,90,0);`.

Errors

As usual, inputs are matched left to right exactly in order – it won't skip any to try to make things work. Suppose we have this:

```
void A(string w="cow", int x=0) { ... }
```

You don't have to give this any inputs, but if you do, the first one has to be a string, for `w`. `A(3);` tries to put 3 into `w` and can't. It gives you the odd error *No overload for method 'A' takes '1' arguments*

Because of the left-to-right, no-skipping rule, default parameters have to start from the right. This next function is illegal. It tells you *Optional parameter cannot precede required parameters*:

```
void A(string w="cow", int y) { ... } // ERROR
```

The problem is that you have to give it `y`, since it has no default, which means you also have to give it `w`. There's no way to make it ever fill in `cow` automatically.

16.3 overloading

You're allowed to reuse a function name if the parameters are different. This is called **overloading**. That's the entire rule. The rest of this is just examples and how we usually use that trick.

A different number of inputs lets you reuse the name. For example, we can make a 2 and a 3 input `max`:

```
float max(float a, float b) { if(a>b) return a; else return b; }
```

```
float max(float a, float b, float c) {
  if(a>b && a>c) return a;
  if(b>c) return b;
  return c;
}
```

They're still two completely different functions. But if we type `max(` the pop-up shows us both, making it look like there are two versions of `max`.

Different types also let us overload a name. For example we can write another version of `max` with 2 inputs, but both are `int`'s:

```
int max(int a, int b) { if(a>b) return a; else return b; }
```

This lets us write `int n=max(5, 2);` without a cannot-convert error. It runs the int version. But it's still nothing special – we have three completely different functions which happen to share a name.

Here's a silly overloading example that shows the rules:

```
void silly(float x) { print( "A" ); }
void silly(string w) { print( "B" ); }

silly(1.2f); // A
silly(""); // B
silly(4); // A (4 can turn into a float, but not a string)
```

In Unity our old Random.Range is overloaded. An `int` version gives whole numbers, like rolling a die:

```
// int version, since inputs are ints:
int n = Random.Range(1,4); // random 1, 2 or 3
// it rolls 1 less than the maximum, just because

// old float version. Could roll 1.653 or 3.983
float f = Random.Range(1.0f, 4.0f);
```

The same as before, these are 2 completely different functions. It just feels like 2 versions of `Random.Range`. They could have named the second one `RangeInt`, but most people think it's nicer to type `Random.Range(` and look through the options.

The full rules for overloading are complicated. For example `Random.Range(2, 5.0f);` has one int and one float. It runs the float version since it's closer. And if you write `float f=Random.Range(1,4);` it runs the int version. There are more rules than that, but if you need them it probably means you've gone a little too nuts on overloading.

16.4 Style

Overloading and default parameters are often combined to make it look like we have one very flexible function. When you type a function name and see the little “1 out of 4” indicator, you can arrow down to see the overloads or versions with and without default parameters.

Here’s an example of where it feels like we have 4 options for setColor, but it’s actually 3 different functions, one of them using a default parm:

```
void setCol(float r, float g, float b) {
  GetComponent<Renderer>().material.color = new Color(r,g,b);
}

void setCol(float greyScale) {
  setCol(greyScale, greyScale, greyScale); // call 1st version
}

void setCol(string colName, float bright=1.0f) {
  float r=0, g=0, b=0;
  if(colName=="red") r=1;
  else if(colName=="yellow") { r=1; g=1; }
  else if(colName=="purple") { r=1; b=1; }
  ...
  r*=bright; g*=bright; b*=bright;
  setCol(r,g,b); // calling 1st version
}
```

The pop-up for setCol will show us these:

```
setCol(float r, float g, float b)
setCol(float greyScale)
setCol(string colName); // <- bright filled in with 1
setCol(string colName, float bright);
```

Before we had overloading, we gave functions slightly different names like `max2f`, `max3f` for 2 and 3 float max functions, and `max2i` for the int version. That was fine. You can still do that. But today almost every computer language has overloading.

Using overloads has zero effect on the final program, since the compiler takes care of them. It checks the inputs and figures out which one to use, baking that into the compiled code. In fact, it secretly gives them all different names.

Chapter 17

booleans

So far, our types are `int`, `string` and `float`. This whole section is adding a new one called **Boolean** for true/false variables (boolean is the math word for "true/false math.") They're really very simple, and we could have seen them back in the `if` chapters.

I waited this long since we didn't really need them so far. Plus, now that we know lots of other things, we can go back and see more tricks using `bool`s. For example, functions returning `bool`s, or using them to untangle some messy nested `if`s.

17.1 Bool rules

The official name for the type is `bool`. Like `int`, it's all lower-case. Since it's a **type**, you can declare variables with it: `bool a;`.

`bool` literals are `true` and `false`. Those are built-in computer words, like numbers, meaning you can say `a=true;`. Those are all there are, which is sort of a new thing. An `int` can be any of a billion different numbers, `string`s can be any word in the dictionary, plus lots more. `bools` can only ever be `true` or `false`. Exs:

```
bool a;  a = true;
bool b = false;
a = b; // now a is false
```

It takes a little to get used to `true` and `false`. They aren't strings and they aren't variables. If it helps, `true` is stored as a 1 and `false` as a 0. But they officially count as type `bool`. For fun, some more examples, mostly errors:

```
string w; w="true"; // legal. Not a bool, just another string
w = true; // ERROR -- cannot assign a bool to a string
bool b; b="true"; // ERROR -- "true" is a string
```

```
int true; // ERROR -- "unexpected symbol."
// can't use a built-in C# word for an identifier

int n; n=false; // ERROR. false cannot be converted to an int
// even though false is really 0, it's still a bool

bool f; f=1; // ERROR. 1 cannot be converted to a bool
// even though true is 1, 1 counts as an int
```

The "math" you can do to a `bool` should be familiar: you can use `&&` and `||` in the exact way we did before. Can also use **not**(`!`) (explanation point) to flip true/false(this is also the same as before.) Some examples that do nothing useful except show rules:

```
bool a = true || false; // true
bool b; b = true && false; // false
bool c; c = !false; // c is true

a = b || c; // true (false||true is true)
a = b && c; // false (false&&true is false)
a = !b; // true (!false is true)
```

These look funny for two reasons: 1) we never used these outside of an `if` before, and 2) we always got true/false using compares (ex: `x==0`, `y>0`.) But, the computer doesn't care. Where-ever math is, the computer solves it. If something is `false` because we computed it, or because we just typed in `false`, that's all the same to the computer.

The answers to comparisons are `bools`, which means you can assign them. These set `bool` variables based on compares (they still don't do anything useful):

```
bool isPositive; isPositive = n>0;
bool is1to10= n>=1 && n<=10;
```

Those are easy if you cover up the first parts. We've seen `n>0` and `n>=1 && n<=10` plenty of times. We know how to convert them into true/false. If it helps, draw a little `if( )` in pencil around them.

If `n` is 12, `isPositive=n>0;` sets it to true.

These next two use `==` compares, which looks extra weird mixed with assignment `=`. The trick is the same: figure the true/false answer, then assign:

```
bool isCat; isCat= w=="cat";
bool isLucky= n==14 || n==77 || n=123;
```

If `w` is `"duck"`, then `isCat` will be false. If `n` is 77, `isLucky` will be true.

17.1.1 bool variables in `if` statements

Before, I said the rule for the test in an **if** was it had to be something true or false. The actual rule is it has to be a **bool**. Things like `n>0` or `x==1 || x==6` are officially `boolean expressions` and give a `bool` result.

But `if`'s don't especially like fancy expressions. Like everything else, you can give them an actual `bool` value (`true` or `false`,) or a `bool` variable, or a `bool` expression (what we've been doing.)

The simplest examples are using literals. These are useless, but they are legal:

```
if(true) print( "I always get printed" );

if(false) print( "I never get printed" );
```

These are confusing because they're so simple. The first one is like `if(1==1)`, but even simpler. The second is like `if(1>2)`, but even simpler.

A more useful trick is using `bool` variables inside `if`s. Some examples using a `bool` named `isReady`:

```
bool isReady; // pretend someone sets this

if( isReady ) print("ready!"); // most common use

if( isReady ) print("ready!"); // it also works with an else
else print("waiting...");

if( isReady == true ) print("B"); // longer version

// these are the same:
if( isReady==false ) print("waiting...");
if( !isReady ) print("waiting...");
```

The first one `if(isReady)` is the most common way to use bools and ifs. The `if` runs when `isReady` is true. The if-else below is nothing special. I just wanted to show that any kind of `if` can use a **bool** variable in the test.

The line with "B", `if(isReady==true)` is the same thing, and is also legal. It just works out that the `==true` isn't needed. Most people leave it out. For example, `if((n>0)==true)` is legal, but not needed. Use `==true` whenever you think it looks nicer.

The last two lines are the same – they run when `isReady` is false. `if(isReady==false)` should be obvious. `if(!isReady)` is a little more common. Most people read it as "if not isReady". Sometimes `if(b==false)` looks nicer, and sometimes `if(!b)` does. I use both about equally.

`bool`s with `&&` and `||` are nothing special. Assume we have `bool a,b,c;`. Here are some legal examples using and/or combinations:

```
if( a && b ) print( "A and B are both true" );
if( a==true && b==true ) print( "same as above" );

if( a && !b ) print( "A is true and B is false" );
if( a && b==false) print( "same as above" );
if( a==true && b==false ) print( "same as above" );

if( a || b ) print( "A, or B or both are true" );

if( a || !b ) print( "A true, or false B, or both" );
if( a==true || b==false ) print( "same as above" );

if( a || b&&c ) print( "tricky, but legal" );
if( a==true || (b==true && c==true) ) print( "same as above" );
```

We can also mix `bool` variables and regular compares:

```
if( isReady && n==0 ) print( "please select an action" );
if( !isReady || n<0) print( "waiting" );
```

17.2 Common bool tricks

17.2.1 Pre-computing if tests

One use for `bool`s is to compute and save an if-test, or part of one, then use it later. Sometimes that saves you from computing the same thing twice, and sometimes it makes the `if` easier to read.

Here's an example of an `if` spread out over a few lines:

```
// original 1-line version:
if( n>=100 ) print( "n is big" );

// same thing, in 2 steps:
bool big= n>=100;
if(big) print( "n is big" );
```

This is the same idea as using a temporary variable for math. The second version pre-computes `n>=100` and saves it. We couldn't save a true/false before, but now we can. Then `if(big)` uses the saved value. A longer example of the same thing:

```
bool oneDigit= n>=-9 && n<=9;
if(oneDigit) print( "n is a 1-digit number" );
```

The trick works with parts of if-tests. In this example, there are a lot of ways to win, and I use a temporary `bool` to save each one:

```
bool hasDoubles = n==33 || n==44 || n==55 || n==77;
bool zone1 = n>=150 && n<=170;
bool bigEnough = n>=1500;

if(hasDouble || zone1 || bigEnough) print( "winner" );
```

If we tried to write this as one big `if`, we'd have to spread it across lines anyway. This way we get to use helpful variable names, making it easier to read.

Precomputing tests can simplify my "which xy quadrant" example from back in the **if** chapter. Here's a nicer rewrite using `bool` variables (remember, positive x & y is quadrant 1, and so on):

```
bool xIsPos = x>=0, yIsPos = y>=0; // are x/y positive?

int quad=-1; // no quad, just in case
if(yIsPos) { // top part:
  if(xIsPos) quad=1;
  else quad=2;
}
else { // bottom part:
  if(xIsPos) quad=4;
  else quad=3;
}
```

This used `xIsPos` twice, so we saved some typing there. It only used `yIsPos` once, but I think it still looks nice, computing both together on top.

Here's a very long example using `bool`s to precompute possible answers. We're making a game that rolls three dice, and you get points for things like 3-of-a-kind, a pair, a straight (like 234 or 345,) or a wrap-around straight (561 or 612). The first part rolls the dice and sorts them. Sorting three numbers has nothing to do with `bool`s, but is fun:

```
int a=Random.Range(1,7), b=Random.Range(1,7), c=Random.Range(1,7);
// uses the int-only version. rolls 1,2,3,4,5 or 6

// Sort them: very sneaky way:
if(a>b) { int temp=a; a=b; b=temp; }
if(b>c) { int temp=b; b=c; c=temp; }
if(a>b) { int temp=a; a=b; b=temp; } // same as 1st if
```

Sorting like this is very, very sneaky, but it's a neat example of `if`s. If you want, try it with combinations of `a,b,c`= 321, 213, 312 ... . But all you have to know for the rest is that this really does sort them.

Now for the interesting part. I'll use a `bool` to hold each "fact." I don't want to worry right now about how many points a pair or straight is worth. For this part, I just want to compute & save each fact:

```
bool triples= a==b && b==c;
bool pair= triples==false && (a==b || b==c);
bool straight= b==a+1 && c==b+1;
bool wrapStraight= a==1 && b==5 && c==6 || a==1 && b==2 && c==6;
```

I think that looks really nice: the left side with `triples`, `pair` ... makes it easy to see we're computing "do we have any of these special rolls?" And having each one on a line by itself, after the name, makes it easy to see which math is computing which thing (imagine instead all that math was in a very long `if`. Ugg.)

The math is also somewhat interesting, because they're sorted. Some comments: `triples` doesn't need to compare `a` and `c`. `pair`, can also skip comparing `a` and `c` (we can't have 323.) `pair` also checks `triples==false` so we don't double-count 333 as a triple and pair.

`straight` seems obvious, but I still had to check it three times to be sure it said "`b` is one more than `a`, `c` is one more than `b`." For `wrapStraight`, I just listed both possibilities (561 or 612.) Cheesy, but not too long and easy to read.

To make this next part really interesting, pretend there are three versions of the game where different rolls give different points (I just made up some rules and what amounts you win.) This checks the game version, then uses the `bool`s to figure results:

```
int winChips=0; // how many chips we win from this roll

if(version==1) { // big payout on triples, everything else is nothing:
  if(triples) winChips=50;
}
else if(version==2) { // only triples and pairs pay out:
  if(triples) winChips=30;
  if(pair) winChips=8;
}
else if(version==3) { // trying for straights, get a little for pairs/triples:
  if(straight) winChips=25;
  if(wrapStraight) winChips=20; // not quite as good as a real straight
  if(triples || pair) winChips=5;
}
```

Because we computed `triples` and so on ahead of time, and gave them nice names, the lines in this part are easy to read. `if(triples) winChips=30;` says exactly what it does.

For real, I only do this for long, ugly `if`s. This dice game is an example of when I would, but probably nothing simpler.

17.2.2 bool return values

Now that `bool` is a type, functions can return them. Their main use is for putting a function inside of an `if`. Suppose there's some long compare that we use a lot – why wouldn't we want to write a function for it?

A good one is checking whether two numbers are close together. We already know how to do that with `if`'s. We only need to wrap a function around them. I'll name it `isCloseEnough`:

```
bool isCloseEnough(float a, float b) { // <- bool return
  float diff = b-a;
  if(diff<-0.01f || diff>0.01f) return false;
  else return true;
}
```

As a simple use, `b=isCloseEnough(5,9);` makes `b` false. But the best uses are inside of an `if`:

```
if( isCloseEnough(x1, x2) )
  print("objects are lined up!");
```

The inside of the `if` jumps to the function, runs it, and returns true if they were close together. It's a normal function call. Then the `if` continues. There aren't any new rules here, but it's certainly a new trick.

Notice how it ends with two close-parens. I left some space, but most people smush them together.

Two more sample uses:

```
if( isCloseEnough(x,0) )
  print("x is centered");

if( isCloseEnough(x,0) && isCloseEnough(y,0) )
  print("almost at (0,0)");
```

A fun trick is to add `print("comparing "+a+" and "+b);` as the first line in `isCloseEnough`. In the last example, you'd see it print maybe "comparing 1 and 0" then "comparing 2 and 0."

You could also use `if(isCloseEnough(x,0)==true)`, but don't need to. If you want to check whether two things aren't close, can use `if( isCloseEnough(n1,n2) == false)` or `if(!isCloseEnough(n1,n2) )`.

`bool` functions aren't limited to standard math. A really nice use of a `bool`-returning function is to simplify some of our personal tests.

Suppose we're using 0-6 for day of the week, where Sunday is 0. Here are some functions that "know" the days, and use them:

```
bool isWeekend(int dayNum) { return dayNum==0 || dayNum==6; }
bool isWeekday(int dayNum) { return dayNum>=1 && dayNum<=5; }
bool isPartyday(int dayNum) { return dayNum>=5; }
```

Even though these are short, you can see how nice this trick is by comparing with and without a function:

```
if(d1==0 || d1==6) { dogHours=2; } // what??
if(isWeekend(d1)) { dogHours=2; } // ah ... checking for weekend
```

In the second one, it's really obvious that you're checking for the weekend. The first is also easier to write the wrong way – maybe you forget whether Sunday is 0 or 1.

You can assign a `bool` function result to a variable. It might be helpful if you reuse that value a lot:

```
bool xIs5 = isCloseEnough(x, 5.0f); // we use this twice

if( xIs5 && y<5 ) { ... }
if( xIs5==false && y>10) { ... }
```

17.2.3 Bools for function settings

A `bool` parameter can be used to add an option to a function. For example, my very old `resetToLeft` function might have an option to give `y` a random value:

```
void resetToLeft(bool randomY) {
  x=-7; xSpd=0; ySpd=0;  // the usual stuff

  if(randomY) y=Random.Range(-3.0f, 3.0f);
  else y=0;

  transform.position = new Vector3(x,y,0);
}
```

`resetToLeft(false);` has `y` centered, at 0. `resetToLeft(true);` tells it to roll a random value for `y`. A harder one: `resetToLeft(laps>=10);` randomizes `y` starting at the 10th lap. As usual, it works out the math first. `laps>=10` is either true of false.

We often use this with default parameters. If we rarely randomize we'd write `void resetToLeft(bool randomY=false)`. That lets us use `resetToLeft();` most places, with a true stuck in for the few times we need to randomize.

17.2.4 flags

Like any other variable, `bool`s can be created to remember things. They can't count, but they are good for remembering "did that happen yet?" things. We usually call that a **flag**.

In this example, I want to take the wrap-around moving Cube code, and make it "bend" halfway across by rolling a random y-speed. My first attempt has a bug: when we get halfway across, it rerolls `ySpd`, but does it every step, making us twitch as we move:

```
// standard Cube wrap-around code, with midway bend (works wrong)
float x=-7, y=0;
float ySpd=0;

void Update() {
  x+=0.1f; y+=ySpd;

  if(x>0) { // midway bend:
    ySpd=Random.Range(-0.06f, 0.06f);
  }

  if(x>7) { // standard wrap-around:
    x=-7; y=0; ySpd=0;
  }
  transform.position = new Vector3(x,y,0);
}
```

The problem is, it rolls a random speed for `y` when it passes midway, at 0, But during the next move, it doesn't remember that it's already done it, so does it again, and again.

We can use an extra `bool` to remember we've already done it. I'll name it `bent`. False means we haven't bent yet, true means we have:

```
// Working Cube wrap-around code, with midway bend
float x=-7, y=0;
float ySpd=0;

public bool bent = false; // <- new
// public, in case we want to watch in the Inspector

void Update() {
  x+=0.1f; y+=ySpd;

  if(x>0 && bent==false) { // <- new
    ySpd=Random.Range(-0.06f, 0.06f);
    bent=true; // <- new
```

```
  }

  if(x>7) {
    x=-7; y=0; ySpd=0;
    bent=false; // <- new
  }
  transform.position = new Vector3(x,y,0);
}
```

The first `if` says "if we're past the middle and haven't bent yet." Then inside, `bent=true;` makes sure we don't do it again. When we restart a lap, we reset `bent` back to false, allowing a bend for this new lap.

When someone says "use a flag," this is all they're saying – declare an extra `bool` variable. Setting it true or false is called setting or resetting the flag. The term comes from curbside mailbox flags (the ones you set to up when you put outgoing mail into them).

17.2.5 Incremental bool calculations

Sometimes it's nice to compute a `bool` a bit little at a time. Suppose we want to check whether `n1` through `n4` are all positive:

```
bool allArePositive=true; // so far
// each number can reject:
if(n1<=0) allArePositive=false;
if(n2<=0) allArePositive=false;
if(n3<=0) allArePositive=false;
if(n4<=0) allArePositive=false;
```

Notice how it starts `true`, and any one failure makes it `false`. The other pattern than works, for certain problems, it starting false, and if even one person is happy, it become true.

Of course, we could have written it in one big or:

```
if(n1<=0 && n2<=0 && n3<=0 && n4<=0) allArePositive=false;
```

Here's one where we have to do a little more math for each check. We want to see a movie, but any one thing can prevent that:

```
bool stuckAtHome=false; // so far

if(homework>0) stuckAtHome=true;
if(isWeekday(dd)) stuckAtHome=true;
int daysSinceLastMovie=dayNow-lastMovieDay;
if(daysSinceLastMovie<4) stuckAtHome=true;

if(!stuckAtHome) print("Here's $20. Have fun at the movies" );
else print( "where are you going, young lady?" );
```

Notice it follows the same pattern. We start false, and each test might set it to true.

This wouldn't fit very well into an `if`, but we could have used a trick from the previous section, making a bool for each, then a big `if` checking them all:

```
// another way to do it:
bool stuckHomework = homework>0;
bool stuckWakeUpEarly = isWeekday(dd);
int daysSinceLastMovie=dayNow-lastMovieDay;
bool stuckSawRecently = daysSinceLastMovie<4;

if(!stuckHomework && !stuckWakeUpEarly && !stuckSawRecently)
  print("Here's $20. Have fun at the movies");
else
  print("where are you going, young lady?");
```

17.3 Testing tricks

Sometimes you want to temporarily enable or disable an `if`. Putting a `true` or `false` is a cheap way to do that. For example, you're testing a mini-game and want to disable a 10 second time-out:

```
//if(totalSecs>10.0f) { // original line
if(false) { // testing temp line. Give me all the time I need
  // time-out code. Never runs for testing
}
```

You can also use `if(true)` as a placeholder when you're writing the program. Suppose you have plans for being out-of-ammo (but aren't even close to writing it yet.) You might do this in the firing code:

```
if(true) { // add ammo check here, later
  // make bullet
  // aim bullet, etc ...
}
else {} // add out-of-ammo stuff
```

All that does is give you a code-reminder, and maybe makes it a little easier to add the real ammo-test later. I only use this trick when I'm sure I'll be adding something, and I know this is where it should go, and I want to mark the spot.

Pulling a test into a bool can help with debugging. Suppose we have `if(x*y<z/q)` (I just made up some random, complicated-looking equation) and it's working funny. We'd like to simplify it for testing. First we rewrite it like this:

```
bool check1 = x*y<z/q;
if(check1) { doComplicatedThing }
```

No change, so far. But now we can change it by adding a new line. This temporarily tests `x*y<10`:

```
bool check1 = x*y<z/q;
check1 = x<10; //  TESTING, temporarily change the condition
if(check1) { doComplicatedThing }
```

When you change a program for testing, it can often be difficult to remember everything you did and to change it all back. Any little thing like this can help.

Chapter 18

Function examples

18.1 Introduction

This chapter is all function examples and tricks, plus some new Unity functions that let us do neat things.

One style for writing short functions is to change the input into the answer. It saves you having to make an extra variable, and often looks fine. Here's the `clamp` function written in that style (recall if forces the first input to be between the other two.) `n` is the input and the output (after we fix it):

```
float clamp(float n, float min, float max) {
  if(n>max) n=max;
  else if(n<min) n=min;
  return n;
}
```

On longer functions it can be too confusing. You can forget that `n` isn't the original input any more.

Another style is putting all the math on the return statement. This is popular for short functions:

```
// recall lerp(4, 7, 0.3) is 30% of the way from 4 to 7:
float lerp(float begin, float end, float pct) {
  return begin+(begin-end)*pct;
}

// using the fancy ?: value-returning if for max:
float max(int a, int b) { return a>b?a:b; }
```

"Front-end" functions often use that trick. This uses the real clamp to do the work:

```
float clamp1to10(float n) { return clamp(n,1,10); }
```

Putting it on one line like that makes it easy to see that all this function does is run the real `clamp` with 1 and 10 already fllled in.

This uses the old `bool closeEnough` function to check for a number close to zero:

```
bool almostZero(float n) { return closeEnough(n,0); }
// ex:
if( almostZero(milkLeft)) print("buy milk, ma.");
```

The same trick works on made-up non-mathy functions. Here's a story-telling function, then a shortcut with the name filled in:

```
// two-input story function:
string adventure1(string name, string animal) {
  string ans = "One day "+name+" took the "+animal+" for a walk on the beach.";
  return ans;
}
```

We run that a lot with the name `"you"`, so we make a shortcut:

```
// short-cut using adventure1:
string youAdventure1(string ani) { return adventure1("you", ani); }
```

`youAdventure("cow");` is you and a cow on the beach.

We often mix the tricks. This `closeEnough` has the equation on one line, and uses function `abs` (absolute value) to do the work:

```
bool closeEnough(float a, float b) {
  return abs(b-a)<0.01f;
}
```

In the old days, I would have computed `abs(b-a)` into a variable, then used an if/else it check it. `return abs(b-a)<0.01f;` does the same thing, but shorter.

You're allowed to nest a function call in a function call. This uses 2-input `max` to find the max of 3 numbers:

```
float max3(float a, float b, float c) {
  return max( max(a,b), c);
}
```

It runs "inside out." The inside `max` finds the largest out of `a` or `b`, then the outside `max` compares that to `c`.

18.2 Change me type functions

A lot of value returning functions look like they change you, but, of course, they can't. For example:

```
int cows=14;
clamp(cows,0,10);
print(cows); // 14 ??  why wasn't this clamped to 10?
```

The confusion is, in our minds, `clamp(cows,0,10)` forces `cows` between 0 and 10. But what it really does is compute that number and return it. Like every other function, we have to "catch" the result:

```
int cows=14;
int c2=clamp(cows,0,10);
print( cows + " " + c2 ); // 14 10
```

To force cows between 1 and 10, use `cows=clamp(cows,0,10);`. Likewise, to make `n` positive, use `n=abs(n);`.

18.3 Dice functions

Functions using random feel different, so I'm putting them in their own section. None of these are all that special. Just examples of how, if you use `Random.Range` a lot in a certain way, you can write a function for it.

Rolling a 50% chance isn't that long, but if we use it a lot, we could turn it into a function. A funny thing is it takes no inputs. `heads()` is true half of the time:

```
bool heads() { return Random.Range(0.0f, 1.0f)>0.5f; }
```

Now we can say `if(heads())`.

Rolling a percent, like a 20% chance, is also short but awkward. Here's a common function to make that easier. It uses 1 to 100, so you can write `chance(20)` for a 20% chance:

```
bool chance(float pct100) { return Random.Range(0.0f, 100.0)<=pct100; }
```

`if(chance(20))` is true 1 time in 5. If we have a game with lots of percent-based odds, this will be useful.

Sometimes we want get a random plus/minus value, for example -0.1 to 0.1, or -2 to 2. We can make a shortcut function for that:

```
float randPM(float plusMinus) {
  return Random.Range(-plusMinus, plusMinus);
}
```

If we want about 50, we can use `50+randPM(2);` to get 48 to 52. To have red to be about 0.4 use `0.4f+randPM(0.1f);`.

If we have a board game, we might roll two 6-sided dice a lot. The integer version of 1-6 is `Random.Range(1,7)` (it won't roll the highest, just because.) We can write a function to roll that twice and add:

```
float roll2d6() {
  return Random.Range(1,7) + Random.Range(1,7);
}
```

Now we can use `int move = roll2d6();`. If you need an 8 or more to beat the orc, use `if(roll2d6()>=8)`.

18.4 More style; types of functions

We often break up a main program into slop-tastic chunks like this:

```
void Update() {
  doAllMoves();
  updateColorChange();
  if(difficulty>1) updateMoveMonsters();
  updateCheckWinLose();
}

void doAllMoves() { ... }
```

It's pretty much understood that `doAllMoves` will be a sad excuse for a function. It reads and sets a bunch of globals. The others will be the same way. All they do is move code from out of Update. But that's helpful. It makes things easier to find. We can use the editor collapse-code-block feature to hide or show only the functions we need to see now.

We love functions that only read from their inputs. We can see everything they use right there on the line calling them. But sometimes, rarely, we make a function that has a setting. Here's a standard set color with a special option to not be too dark:

```
void setColor(float r, float g, float b, bool notTooDark=false) {
  if(notTooDark) { // none can be less than 0.3:
    if(r<0.3f) r=0.3f; if(g<0.3f) g=0.3f; if(b<0.3f) b=0.3f;
  }
  GetComponent<Renderer>().material.color=new Color(r,g,b);
}
```

`setColor(r1,g1,b1);` runs it normally (since the 4th uses a Default setting). Otherwise run it with `setColor(r1,g1,b1,true);`.

Another way to do that is using a global for the setting:

```
// change this to change how setColor works
bool colorCantGetDark=false;

void setColor(float r, float g, float b) {
  if(colorCantGetDark) {
    if(r<0.3f) r=0.3f; if(g<0.3f) g=0.3f; if(b<0.3f) b=0.3f;
  }
  GetComponent<Renderer>().material.color=new Color(r,g,b);
}
```

This can be nice. Suppose we can't be too dark for the next 10 seconds. Set `colorCantGetDark=true;`, then `false` after 10 seconds. That's way easier than needing it as a 4th input.

We rarely do things this way. It's easy to forget about the setting – it's more hidden than if it was an input. And if we need this function in a new program we also have to copy the global variable. But Unity has at least one thing that works like this (there's a global setting for raycasts).

Some functions mostly do something, but return a minor value, usually saying if it worked. For example, this function's job is to move and wrap-around. It returns true when it wraps:

```
bool moveAndWrap(float xMove) {
  // true means we had to wrap-around
  x+=xMove;
  if(x>7) { x=-7; return true;}
  if(x<-7) { x=7; return true; }
  return false; // didn't wrap around
}
```

We can use it by itself: `moveAndWrap(0.1f);`. We ignore the return value. Or we can catch the answer and act on it:

```
bool didALap = moveAndWrap(0.1f);
if(didALap) {
  lapCounter++;
  // do more lap stuff
}
```

The first line is as if we're calling `moveAndWrap(0.1f)` to do things for us. But then we're asking it "so, how did it go? Was there a lap?"

We often us it directly inside the `if`:

```
if(moveAndWrap(0.1f)) { // do lap stuff
```

Inside of the `if`, the program actually moves us. Functions inside of `if`'s really, really shouldn't also change things. But in this case it's fine. We understand it's a "do some stuff then tell me how it went" situation.

When we have a function that mostly computes, but also does something, we call those *side effects*. For example, this checks whether we're off the edge, but also adjusts us:

```
bool isOffEdge() {
  // assume x is a global for our position
  if(x>7) { x=7; return true;}
  if(x<-7) { x=-7; return true; }
  return false;
}
```

This is confusing, since it feels like an answer-only function:

```
if(isOffEdge()) { // arrg. This also may have moved us
  // check how far off we are:
  // opps! We're in-bounds now, since isOffEdge fixed us
  ...
}
```

But if we renamed it `bool forceIntoBounds` it would feel like a do-something function that happened to have a minor return value.

18.5 More Unity Movement examples

We know enough now to use some new Unity built-ins and make some more complex, fun movement. I'm going to barely skim some of the set-up – I'm assuming at this point you either played with Unity enough to figure it out, or you're happy just reading these examples.

We need 3 new things. The Rotate function spins us. This spins us 2 degrees each frame, like a top:

```
void Update() {
  transform.Rotate(0,2,0);
}
```

Rotate has 3 inputs. The other 2 rotate us like a summersault, or rolling sideways. We won't use them.

The Translate function moves us on our personal forward. Running this with a tilted Cube will go whichever way it's facing. The speed of 0.1 is about

the same speed we were using before – takes 140 updates to cross from -7 to 7 at that speed:

```
void Update() {
  transform.Translate(0,0,0.1f);
}
```

We won't use the other 2 input slots. They move us sideways and up.

Combined, the two new functions can move us in a circle, spinning and moving forward:

```
void Update() {
  transform.Rotate(0,2,0);
  transform.Translate(0,0,0.1f);
}
```

The last new rule is asking the cube where it is. `transform.position.x`. gives us our current x position, which is probably -7 to 7. As a test, this snaps us to the center whenever `x` goes past 3:

```
void Update() {
  transform.Rotate(0,1,0); // slow spin for bigger circles
  transform.Translate(0,0,0.2f); // faster movement

  // checking our position:
  if(transform.position.x>4)
    transform.position = new Vector3(0,0,0); // snap to center
}
```

To see this we'll need to be in top view. We can put the camera 10 units over the center, at (0,10,0), Then tilted looking down with rotation (90,0,0). I put a little ball as a child, just in front of it, so I could see which way it was aimed.

Now we're ready to write some code. Whatever we do, we want to stay in bounds. We'll write a function to check for it:

```
bool outOfBounds() {
  float x=transform.position.x;
  float z=transform.position.z;
  if(x<-7 || x>7 || z<-5 || z>5) return true;
  return false;
}
```

If we slow down our spinning, we can rotate in circle big enough that we'll go out of bounds. The code below will spin us 180 degrees when that happens:

```
void Start() {
  // We start with a random spin:
  float ySpinAny=Random.Range(0,360);
  transform.Rotate(0, ySpinAny, 0);
}

void Update() {
  // spin slowly and move forward (same code as before):
  transform.Rotate(0,1,0);
  transform.Translate(0,0,0.1f);

  if(outOfBounds()) // turn around when out-of-bounds:
    transform.Rotate(0,180,0);
}
```

This is nice, but gives us boring, predictable curves after watching it for a while.

We can change it up with the usual tricks. Our turn speed can be a variable and we can add a little randomness to the 180 degree flip:

```
float spin=1; // our turn speed

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
}

void Update() {
  transform.Rotate(0,spin,0);
  transform.Translate(0,0,0.1f);

  // 4% chance to change the spin:
  if(Random.Range(0.0f, 1.0f)<0.04f)
    spin=Random.Range(-1.0f, 1.0f);

  if(outOfBounds()) {
    // spin about 180 degrees, 150 to 210 degrees:
    float turnAround = Random.Range(180-30, 180+30);
    transform.Rotate(0,turnAround,0);
  }
}
```

This moves in semi-interesting curves, with fun bounces.

This has the same rare stuck-out-of-bounds bug as the old back-and-forth code. If it somehow gets too far out it will just spin in place. We can make an improved outOfBounds function to force it in. The name is changed so we remember it also can move us:

```
bool keepInBounds() {
  float x=transform.position.x;
  float y=transform.position.y;
  float z=transform.position.z;
  bool wasOut=false;
  if(x<-7) { x=-7; wasOut=true; }
  if(x>7) { x=7; wasOut=true; }
  if(z<-5) { z=-5; wasOut=true; }
  if(z>5) { z=5; wasOut=true; }

  // if it was out of bounds, move us to the fixed position:
  if(wasOut) transform.position=new Vector3(x,y,z);

  return wasOut;
}
```

I think that's a neat use of a bool variable.

We can try other things. If we move sideways and forward, we're going diagonally. If we use a counter we can move left 10 times, right 10 times, left 10 times in a zig-zag:

```
float zigSpd=+0.05f; // sideways movement. Will flips +/-
public int zigLen=14; // how long each zig lasts
int zigTimer; // counts up to zigLen as we move

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}

void Update() {
  // forward and left or right:
  transform.Translate(zigSpd,0,0.1f); // forward and L or R

  zigTimer--;
  if(zigTimer<0) {
    zigSpd*=-1; // flip left/right
    zigTimer=zigLen; // restart the count
  }

  if(keepInBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

This works – it moves in short, straight lines making a zig-zag – but I just don't like the way it looks. My second try at zig-zagging is to always move forward, and change our facing. We'll snap 30 degrees left and right as we move:

```
public int zigLen=35; // how long each zig lasts
public float zigDegs=30.0f; // degrees of each zig
int zigTimer; // counts up to zigLen


void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}

void Update() {
  transform.Translate(0,0,0.1f); // only move forward

  zigTimer--;
  if(zigTimer<0) {
    zigTimer=zigLen; // reset the count
    transform.Rotate(0,zigDegs,0); // do a zig
    zigDegs*=-1; // zig the other way, next time.
  }

  if(keepInBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

18.6 Reading keys

We finally know enough rules to decipher a built-in keyboard reading function. This uses the A and D keys to move back-and-forth:

```
float x=0;

void Update() {
  if( Input.GetKey(KeyCode.A) ){
    x-=0.2f;
    if(x<-7) x=-7; // stay in bounds
  }
  if( Input.GetKey(KeyCode.D) ){
    x+=0.2f;
```

```
    if(x>7) x=7; // stay in bounds
  }

  transform.position =  new Vector3(x,0,0);
}
```

You can probably figure out that `Input.GetKey(KeyCode.A)` is true when A is pressed. Here are some notes, mostly obvious, about it:

- The name of the function is `GetKey`. It's in the namespace `Input`. So, altogether: `Input.GetKey`.
- It returns a `bool` – true means that key is down. Like any `bool` function, you can use it inside of an `if`.
- The input is which key you're checking for. `KeyCode` is just an enumerated type, specially made to give names to all the keys. For example, `KeyCode.DownArrow`.
- There's also an overload which takes a `string` as input. For example, `Input.GetKey("a")`.
- In general there are three ways to check keys: just pressed, currently being held, or just let go. In Unity, these are `GetKeyDown`, `GetKey` and `GetKeyUp`. Most sorts of keys or virtual buttons have those three things.

Here's a different version, which also moves back&forth, but using `GetKeyDown`. Tapping A or D starts you moving left or right. Tapping S stops you. I don't like this as much, but it's fun to program:

```
float x=0;
float spd=0;
// set to 0, +0.2 or -0.2 by keypresses, stays the way it was set

void Update() {
  if( Input.GetKeyDown(KeyCode.A) ) spd=-0.2f;
  if( Input.GetKeyDown(KeyCode.D) ) spd=0.2f;
  if( Input.GetKeyDown(KeyCode.S) ) spd=0;

   x+=spd;
   if(x>7) { x=7; spd=0; }
   if(x<-7) { x=-7; spd=0; }

  transform.position =  new Vector3(x,0,0);
}
```

We can tweak that so holding a key gradually speeds you up:

```
float x=0;
float spd=0; // changed by holding down a key
public float accel=0.005; // how much speed changes in one tick

void Update() {
  if( Input.GetKey(KeyCode.A) ) spd -= accel;
  else if( Input.GetKey(KeyCode.D) ) spd += accel;
  else {
    // if not holding a key, slow down, then stop:
    spd *= 0.98f; // slow down (0.98 is trial and error)
    if( spd<0.02f && spd>-0.02f) spd=0; // near 0 should stop now
  }

   x+=spd;
   if(x>7) { x=7; spd=0; } // kill our speed when we hit an edge
   if(x<-7) { x=-7; spd=0; }

  transform.position =  new Vector3(x,0,0);
}
```

The first two `ifs` don't limit the speed, but we'll hit an edge before we get too fast. The part where it slows down is an old movement trick. We want to make the speed go towards zero, whether it's positive or negative. Multiplying by 0.98 makes 1 and -1 both go towards zero. It's only 2% smaller, but it adds up fast. The last slowdown line says "when speed gets really small, just be 0."

For something completely different, we can have A and D rotate, while W moves forward:

```
void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

  if( Input.GetKey(KeyCode.W) ) transform.Translate(0,0,0.1f);
}
```

We could change this version so the W key is acceleration instead of movement:

```
float spd=0; // forwards speed. always positive

void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

 // speed up if W held, slow down if not:
  if( Input.GetKey(KeyCode.W) ) {
```

```
    spd+=0.01f;
    if(spd>0.14f) spd=0.14f; // maximum speed of 0.14
  }
  else { // slow down if W not held:
    spd-=0.005f;
    if(spd<0) spd=0;
  }
  transform.Translate(0,0,spd);
}
```

Moving onto another completely different movement, we want the Cube to crawls around the sides of the screen. A moves clockwise and D moves counter clockwise.

The basic plan is using a 0-3 `int` to remember which side we're on. I'll use an enumerated type, written specially for this:

```
enum Wall {top, right, bottom, left}; // Wall vars can be 0,1,2,3
Wall wall = Wall.bottom; // sample use
```

`Wall wall;` looks funny but it's common. The type is `Wall` and the variable is lower-case `wall` since I couldn't think of a better name.

`wall` will act as a state-variable. Each wall handles movement on itself, passing you to the next wall when you hit an edge:

```
float x=0, z=-5; // middle of bottom wall

void Update() {
  // get direction of move:
  int mv=0; // -1 0 or 1. Direction based on bottom wall
  if(Input.GetKey(KeyCode.A)) mv=-1; // A=clockwise
  else if(Input.GetKey(KeyCode.D)) mv=1; // D=counter-clockwise

  // handle move for each wall:
  if(mv!=0) {
    float mvAmt=0.1f*mv;

    if(wall==Wall.bottom) {
      x+=mvAmt; // note: + for D, - for A
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.top) {
      x-=mvAmt; // top moves moves backwards from bottom
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.right) {
```

```
      z+=mvAmt;
      if(z>5) { z=5; wall=Wall.top; }
      else if(z<-5) { z=-5; wall=Wall.bottom; }
    }
    else if(wall==Wall.left) {
      z-=mvAmt; // left wall: forwards=down
      if(z>5) { z=5; wall=Wall.top; }
      else if(z<-5) { z=-5; wall=Wall.bottom; }
    }
  }
  transform.position = new Vector3(x,0,z);
}
```

The enumerated type really was helpful here. While writing this, I messed up some of the numbers and directions. Seeing lines like `if(wall==Wall.right)` made it easy to see I was in the correct section.

18.7 Making a real function library

So far, I've been suggesting that all of your useful functions should be pasted into each new script. That will work, and we did it in the old days, but there's a nicer way. Almost all languages and environments let you spread your program over several files. Common, similar functions are usually placed in a file by themselves.

The steps to putting functions by themselves are:

- Make a new C# script the usual way (`Project -> CreateC#script`; double-click to open in the editor.) The name doesn't matter.
- Delete everything past the first two `using` lines.
  All that stuff is Unity set-up for things that go on a gameObject. we're making something different.
- Pick what you want to call the namespace. I'm calling mine `rand`. This does not have to match the file name.
- Set up the namespace like below. You might notice it's a simplified version of what we had before:

  ```
  public class rand {

  }
  ```

- Write your pure functions inside of it, with `public static` in front of each one (ex below).
  By pure, I mean they take inputs, and return values. They can't use any globals (which should be obvious, since how would they know what globals your other scripts have) and they also can't change position, color or size.

- Don't put it onto a gameObject (the system won't even let you.) Just having it be created is enough. The system will find it.
- Now any other script can run them by putting `rand`-dot in front.

A sample file with some random functions:

```
using UnityEngine;
using System.Collections;

class rand {
  public static int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }

  public static bool heads() { return Random.Range(0.0f, 0.1f)>0.5f; }

  public static bool chance(int pct100) {
    return Random.Range(0.0f,100.0f)<pct100;
  }
}
```

Anyone can run `if(rand.head()` for a coin flip.

Unity uses this trick in a few places. `Random` holds several randomizing functions. `Mathf` has lots: `Mathf.Approximately(a,b)` is the close-enough function. `Mathf.Abs`, `Mathf.Clamp` and `Mathf.Max` are also there.

Chapter 19

Structs

This section is about making own own new variable types. It's not as exciting as it sounds, since all we can do is group together `int`s, `float`s and `string`s. But it's a nice trick, and it's the basis for classes and Object Oriented Programming.

For example, it takes three `float`s to make a color (red, green, blue.) A struct lets `Color` be an official type, which the computer knows is made of three `float`s.

Even better, the trick works for stuff we just make-up. Suppose we're writing a program about cows – each has a name, age and weight. It would be nice to build that into the program. Then we could declare `Cow c1;` and let the computer automatically make those variables for the parts.

The common name (and the one C# uses) for these is **struct**. There are some built-ins, and we can make our own. I think the best way to learn them is to make the `Cow` example, then a few more, then give the formal rules.

19.1 Cow struct example

Here's the top part of a program making a sample `struct` named `Cow`:

```
class TestA : MonoBehaviour {

  struct Cow {
    public string name;
    public int age;
    public float pounds;
  }
```

First, let's take a look at the overall form. The last five lines, `struct Cow` { }, count as one thing. The name `Cow` is in front. Then a big set of curly-braces mark what belongs to `Cow`. The inside looks like three variable declarations. They aren't, but it's the same rules.

Now onto what it does. This makes a new **type** named `Cow`, and says how to build it. It doesn't declare any variables, or really do anything. It's more like a recipe or a blueprint. It lets the computer know that if someone declares a `Cow` variable, it's legal, and says how to make one.

The names inside are called **fields**. It's the same use of the word as those web-based forms that say "required field." They describe the parts of a `Cow`.

Now we can declare some `Cows`:

```
int n; // old, boring declare just for reference

Cow c1;
Cow c2;
```

`Cow c1;` declares `c1` as a `Cow`. Just like you're probably thinking, the computer says "A Cow? How do you declare a Cow?" But then it looks near the top and sees `struct Cow`. That tells the computer it *can* declare a `Cow`, and the middle part tells it what the parts are – a string and int and a float.

`Cow c2;` declares another `Cow`. Nothing new there – I just want two `Cows` for later in the example.

Here's a picture of all 3 variables in the program:

```
n ___

c1               c2
-------------    ------------
| name:      |   | name:      |
| age:       |   | age:       |
| pounds:    |   | pounds:    |
-------------    ------------
```

Again, `n` is nothing special – it's just there for comparison. The interesting thing is that `c1` is one variable, with those three parts inside of it. Then `c2` is another variable, with the same three parts. Inside of `c1`, `name` is a string. There's another string. `name`, inside of `c2`.

Here's a program using `c1` and `c2` (I'm including everything here, so only `Start` is new):

```
class TestA : MonoBehaviour {

  struct Cow {
    public string name;
    public int age;
    public float pounds;
  }

  int n;
```

```
  Cow c1;
  Cow c2;

  void Start() {
    c1.name = "Bessy";
    c2.name = "Miss Cowy-cow";

    c1.age=6;
    c2.pounds=1200;

    print( c1.name + " is " + c1.age + " years old" );  // Bessy is 6 years old
  }
}
```

This is showing us the last thing we need – how to use the parts of a `struct`. `c1.name="Bessy";` shows the rule: put the variable name, then a dot, then the **field**. `c1.name` is the `name` slot in the first cow.

After you type "`c1.`" (c1, then a dot,) autocomplete will even show you the fields: `name`, `age` and `pounds`. It knows about them from the `struct Cow` we made at the top.

The first two lines are showing us that each `Cow` has it's own `name`. `c1.name` and `c2.name` are different. In other words, the picture is correct. The next two lines are using the `age` and `pounds` fields, just to show we can. `c2.pounds=1200;` goes to `c2`, then inside its `pounds` field, and sets that to 1200.

The `print` shows that the "varName-dot-field" rule is like a variable – you can assign to `c1.name` and also read from it.

After `Start` runs, here's the new picture:

```
n ___

c1                      c2
--------------    ------------
| name: Bessy |   | name: Miss Cowy-cow
| age: 6      |   | age:
| pounds:     |   | pounds: 1200
--------------    ------------
```

The picture is really pretty boring. `Start` assigned four things, and the picture shows all four of them. The `pounds` in the lower-right is `c2.pounds`, the last assignment put 1200 in it, and there it is.

`c1.pounds` and `c2.age` are empty, since the code didn't put anything in them. If I tried to print them, I'd get the usual "has not been initialized" error.

L-values

In the assignment chapter, I made a big deal about how the thing on the left had to be a variable, all by itself. You could never have a formula – `x+1 = 7;` is an error. But here, the left side isn't quite a variable. `c1.age` says to find variable `c1`, and then look inside for the `age` part. It's like a little formula, so `c1.age=6;` looks like it breaks the rule.

The real rule for assignment statements is that the left side needs to count as a variable. It needs to be a box that can be changed. The technical name is **L-value**. It stands for "value that can be on the left-hand of an assignment statement." I know **L-value** sounds really fake, but put "lvalue" in a Search Engine – that's really what we call it.

Before this, only actual variables were **L-values**. Using a struct field is a new thing that also counts as a variable. In other words, in every way, `c1.age` works like an `int` variable. You can use `c1.age++;` or `c1.age *=2;` or `c1.name = "a" + c1.name + "b";`.

It takes a while to get used to seeing mini-formulas that count as variables, but they eventually feel natural. Later, we'll see other things like this – they aren't simple variables, but they count that way.

19.1.1 More examples

This is a traditional very basic **struct** example, with just two fields of the same type. It makes a FullName **struct** containing a first and last name. This mini-program creates it, declares some `FullName` variables and uses them a little:

```
// creates a new struct: "FullName":
struct FullName {
  public string first, last;
}
```

As with Cow, this doesn't create any variables. It defines `FullName` as a new struct type.

The fields are both the same type – two strings. You can re-use any types you need for the fields. This also uses the comma-declare shortcut. The same as always it's just a shortcut and isn't any different than declaring them on two lines. A trick in choosing field names is to remember that everyone knows they're part of a FullName. `first` and `last` are fine since we know it's fullname-first and fullname-last.

The start of the code using them:

```
void Start() {
  FullName mom1, mom2; // both are fullnames
```

```
  mom1.first = "Olga";
  mom1.last = "Pog";

  string m = mom1.first + " " + mom1.last;
  print( "Mother1's name: "+ m );  // Olga Pog
```

Structs can be declared using the comma shortcut. They can also be local or global, like `mom1` and `mom2` are local to Start here. You can use struct variables in pretty much all the ways and places you'd use ints or strings.

The two lines assigning to `mom1` are pretty typical. In our minds we're assigning "Olga Pog" as her name, and it takes two steps. It common to assign to every field, all together like this.

In the next line I wanted to show the fields act like strings in all ways: `mom.first + " " + mom.last` is just three strings added together.

I'll reuse some other old tricks for `mom2`:

```
  mom2.first= "blue" + "elf" + 451;
  mom2.last = mom1.last;
  mom2.last += "ie";

  print( "Mother 2 : "+ mom2.first + " " + mom2.last ); // blueelf451 Pogie
}
```

The second line, `mom2.last = mom1.last;` is just copying a string to a string, even if it took those funny dots to get to them. In the next line, the `+=` shortcut works as normal, tacking `ie` to the end.

Here's a short code snippet using a `Cow` and a `FullName` together. Nothing really special, just to show it can be done:

```
Cow a1;
a1.name = "Lou-Lou";
a1.age=7;

FullName p1;
// person has same name as the cow:
p1.first = a1.name; // no problem -- just string = string
p1.last = "Smith";
print( p1.first + " " + p1.last ); // Lou-Lou Smith

// a1.last = "Cowstein"; // ERROR no last in a Cow
// p1.age = 85; // ERROR no age in a FullName
```

`p1.first = a1.name;` is one of those things that can confuse you later. First off, it's simple: `p1.first` is a string, and so is `a1.name;`, so it's no different than `w1=w2;`.

The possibly confusing part, if you start thinking about it, is how we're copying from the inside a `Cow` into the inside of a `FullName`. It sort of looks like the types are different. The trick is, every step of the way doesn't have to match – only the end. You can copy a string field from a Cow into a string field of a FullName.

Here's the picture, after it runs:

```
 a1                      p1
 ----------------    ------------
 | name: Lou-Lou |  | first: Lou-Lou
 | age: 7        |  | last: Smith
 | pounds:       |  ------------
 --------------
```

The two errors at the end are showing how each `struct` type has it's own personal fields – the ones listed in its `struct { }`, and nothing else. `Cow`s don't have last names, because we didn't write `public string last;` inside `Cow`.

This next example is about making a semi-useful `struct` to improve my old move-and-wrap code, and using some Unity-magic to see it.

To make something increase and wrap-around, I need 4 variables: the value, the speed, the maximum and the minimum (the last two are for the wrap-around part.) In my mind, those four variables make one variable-mover. So I'll make a `struct` for it:

```
struct Mover {
  public float val; // the main thing
  public float spd; // amount to change val, each update
  public float min, max; // the limits for val
}
```

I want to use them in a program that changes the red and green parts of my color. Each will use it's own `Mover` variable:

```
Mover red, green;
```

That creates the 4 `float`s I need for red, and the 4 for green. Here's a picture:

```
  red          green
 ---------    ---------
| val     |  | val     |
| spd     |  | spd     |
| min     |  | min     |
| max     |  | max     |
 ---------    ---------
```

We need to add some special Unity tricks to see them in the Inspector. Adding `public` in front of a global normally makes Unity show it, but `structs` need two extra things. You have to also write `public` before the struct definition, and `[System.Serializable]` on the line before. That second thing is a real C# thing, but it's pretty far-out and Unity is abusing it anyway, so I'll just say it's magic.

The final result looks like this:

```
[System.Serializable]
public struct Mover {
  public float val;
  public float spd;
  public float min, max;
}

public Mover red, green; // now they're visible in the inspector
```

The really cool part is, if you put just that in a script (before `Start`), on your Cube, the Inspector will show `red` and `green`. If you pop-open the little triangles, you can see that each one has `val`, `spd`, `min` and `max`. It knows about `struct Mover` and makes a picture for us!

Here's some code using them. It's the same color change code from way back, except using the struct:

```
void Update() {
  red.val += red.spd;
  if(red.val > red.max) red.val=red.min;

  green.val += green.spd;
  if(green.val > green.max) green.val=green.min;

  GetComponent<Renderer>().material.color = new Color( red.val, green.val, 0);
}
```

I like how a simple `struct` helps organize this. The program really just has 8 variables, and before we would have hand-declared all 8, like `float redMax;`. Using the `struct` automatically makes red and green have the same 4 subparts; and typing red-dot and using the pop-up is an easy way to write the program.

The variables start as all 0, so running this won't do anything good. Start could give them values, like `red.spd=0.02f;`. But it's easier entering values through the Inspector. Probably 0 and 1 for min and max. Or 0.5 to 1, and so on. A different speed for red and green will make a longer pattern. With a little work, they can cycle through a bunch of different shades before repeating.

19.2 Common errors

Seeing some common struct errors is a good excuse to talk about the rules, and explain more about how they work:

- `Cow.name = "steve";` is an error, because `Cow` isn't a variable. In the sample program, there are two cows: `c1` and `c2`, so there are only two `names` – `c1.name` and `c2.name`. `Cow` is a **type**, so `Cow.name="Steve";` is like `int = 6;`. The thing in front of the dot has to be an actual declared variable.
- `age=6;` won't change the age of any cows, and is probably an error. There has to be a cow in front – `c1.age` or `c2.age`. Even if you have only one cow variable, `age=6;` won't assume you mean that cow.
- `c1 = "Elly";` is an error. Struct variables always have to use a dot and say which field you want (there are a few exceptions.) Even though `name` is the only place it could go, you still have to write `c1.name="Elly";` The computer will never "guess" a **field** for you.
  Also, as a double-check, the types don't even match: `c1` is a `cow` and `"Elly"` is a string.
- `FullName p; p="Mark";` is an error. I thought that since both parts of a `FullName` are strings, maybe the computer would change them both to "Mark". But it still won't guess fields for me. I have use `p.first = "Mark";`. Or I could use the old same-assign shortcut: `p.first=p.last="Mark";`.
- `print(c1);` isn't a red-dot error, but won't work. You have to print out each part yourself: `print(c1.name+", "+c1.age+ ...)`.
  Most commands with **struct**s are that way. For example, you can't use `f1+f2` on two `FullName`s. You have to add `f1.first+f2.first` yourself.
- `int age;` is *not* an error! The field names only live inside that struct. If you declare variable `age` like that, and use `age=7;` the computer knows you're talking about the regular `age` variable. If you use `c1.age=7;`, it's definitely the `age` field of `c1`.

  Here's a picture of `cow c1, c2; float age;`:

  ```
  c1                       c2
   ------------    ----------
  | name:      | | name:
  | age:       | | age:
  | pounds:    | | pounds:
   ------------   -----------

  age:
  ```

  The three ages are `c1.age`, `c2.age` and `age`. The regular `age` could even be a float or a string. That `age` and the Cows' `age`'s are totally unrelated.

19.3 rules

We've seen most of the rules in examples, but it's still nice to have them all in one place. For most, just skim them to be sure you've gotten them right. But there should be a few new things in here:

- A new **struct** type is defined using `struct` *identifier* { *list-of-fields* }. It goes in the global area. `struct` is short for *structure*, as in a small building.
- The name of a struct is any legal identifier: `a`, `d_2`, `bunchOfRocks` .... We try to use style rules so you don't confuse struct names with variable names. One rule is to capitalize the first letter. Another is to add underscore-t (for example `cow_t`).
- Each field looks like a variable declaration with the word `public` in front: any type, a space, and an identifier. Ex: `public float f;`. They aren't really variable declarations, but it seemed right to borrow the syntax.
  If you forget the word `public`, it isn't a "red dot" error, but it means you can't use that variable (this is a somewhat silly, tricky rule. If you make a struct, but you get errors when you try to use a part, check you put `public` in front of that field).
- Can use the multi-declare shortcut for fields: `struct FullName` { `public string first, last;`}
- Fields can be different types, the same type, or any combination. There's nothing special about a struct where all fields happen to be the same type, or all different.
- It's legal to have only one field, or even zero. That's usually pointless, but sometimes people have a reason to do it. Ex:
  `struct Number` { `public int val;` }. If you do that, you still have to use the fields: `Number x; x.val=5;`
- The order you create fields doesn't matter. In other words, the definition of `Cow` could have `age` first, and it would work the same. Since you look them up using words, the computer finds the correct `field` no matter where it is.
- The field names can be reused outside of that `struct`. Here's a legal example where `Book` uses `title` as a field name, and so does `Song`, and we also declare it as a global:

  ```
  struct Book { public string title;  public int pages; }
  string Song { public string artist, title;  public float length; }
  int title;
  ```

The three `title`s have nothing in common with each other. It's just a coincidence they have the same name.

- There isn't any short-cut for using field names by themselves. Even if you only have 1 `Cow` variable, `age=6;` won't work on it. You always have to have variable-dot-fieldName.

- Structs have a special `=` shortcut to copy one entire struct to another. For two cows, `c1=c2;` means to use `=` on each part. It's a shortcut for `c1.name=c2.name; c1.age=c2.age; c1.pounds=c2.pounds;`.

  There's a new error here as well: every part must have been assigned. `FullName f1, f2; f1.first="A"; f2=f1;` is an error. You have to at least write `f1.last="";` before you can copy it.

- Different structs always count as different, even if they have the exact same fields. For example, if you made `struct Sheep` with the exact same contents as `struct Cow`, and had `Cow c1; Sheep s1;`, then `s1=c1;` would be an error.

  This is on purpose. If you made one `struct` for sheep and another for cow, it's because you want the computer to tell you when you mix them up.

  Of course, `c1.name = s1.name;` is always legal – it's just a string to a string.

19.3.1 namespace vs. struct dots

We already know C# uses the dot for namespaces. In that chapter I warned you we were going to double-use the dot for something else. This is the second use. An example of both uses:

```
float n = Mathf.PI; // namespace
int a = c1.age; // inside struct
```

They both look inside the thing before the dot, but besides that, they're very different. `Mathf` isn't a variable, and there's one `Mathf.PI`. It's really a built-in global.

On the other hand, there's one `age` for each `Cow` we declare; and `Cow.age` is an error.

There really should have been a different symbol for each, and it is confusing at first, but you get used to it. When you see `Jark.groat` you just have to check whether `Jark` is a namespace, in which case `groat` is a global there; or whether `Jark` is a declared variable, in which case `groat` is a field.

Examples:

```
Cow c1; c1.age=Random.Range(0,8);
     field^          ^namespace
int n = Mathf.Max(c1.age, c2.age);
    namespace^      ^field  ^field
```

19.4 Builtin structs: Color, Vector3

We know Unity3D's position and size are a combined x,y&z. Now that we've seen structs, it should be obvious that's how Unity is doing that. `Vector3` is just three floats, named x, y and z. It's already in the computer, so you can't redefine it, but here's what it looks like:

```
struct Vector3 {
  public float x, y, z;
}
```

The name is from math: vector is a math term for list of numbers, and the 3 at the end is a reminder it has 3 parts.

The old position line, `transform.position = new Vector3(x,0,0);`, was using a shortcut. `new Vector3(x,0,0)` is a way to create an instant struct. Creating a normal Vector3 variable can be nicer. Here's the old and the new way:

```
void Start() {
  transform.position = new Vector3(0,0,0); // old way

  // new way:
  Vector3 upperLeft;
  upperLeft.x=-6.5f; // just pick some numbers for x,y,z
  upperLeft.z=4.0f;
  upperLeft.y=0;

  transform.position = upperLeft;
}
```

The last line is assigning a struct to a struct – it copies the x,y,z from `upperLeft` into position (the old line also did that, but I didn't make a big deal about it.)

We'll see how this improves movement code, later.

For fun, write `public Vector3 p;` as a global variable in any script and look in the Inspector. You should see the x, y and z fields going across. It lists them that way instead of with the triangle-toggle, since it's built-in – they tried to make it look nicer.

Scale, a Cube's size, is also a `Vector3`. That might seem odd, since Vector3 is for position, but it makes perfect sense. After all, we use floats for anything that wants a decimal – position, how red to be, how much a cow weighs. Anytime we want an x,y,z together, we should use a Vector3.

This make us be wide and skinny (I'm leaving out the old way):

```
void Start() {
  Vector3 sz;
  sz.x=3; // wide
  sz.y=sz.z=0.4f; // standard double-assign trick
  transform.localScale = sz;
}
```

For extra fun, this uses the same Vector3 to set position, then size:

```
void Start() {
  Vector3 v; v.x=6; v.y=-2; v.z=0; // just some position
  transform.position = v;

  v.x=0.4f; v.y=2; v.z=1; // redo v to use as scale
  transform.localScale = v;
}
```

19.4.1 Color

We've been setting colors using r, g and b – another struct. It looks like this (Unity already has it defined):

```
struct Color {
  public float r, g, b, a;
}
```

The very short field names are fine, since they're inside of a color – `r` clearly stands for red. The last one, `a`, is "alpha" for transparency. It won't do anything (setting up the ability to be transparent takes a few steps).

It turns out Unity has a `Vector4`, which is also four floats. Vector4 and Color are identical structs. But's that legal, and fine, like Cow and Sheep. We prefer using a struct named `Color` for colors, with floats named r, g and b.

If you declare `public Color c1;` as a global, the Inspector will make a fancy color-picker. That's just great, and why we use fancy editors. But it's merely a very nice way to set the 4 floats in a color stuct.

The code below will turn us orange, using a Color variable. Even though alpha is ignored, we have to set it to something – the rule where you can't assign a struct unless every field is set:

```
void Start() {
  Color cc;
  cc.r=0.9f;
  cc.g=0.6f;
  cc.b=0.1f;
  cc.a=1; // ignored, but must be set to something

  // same old line, but assigning our Color variable:
  GetComponent<Renderer>().material.color = cc;
}
```

19.5 Constructors, struct literals

We've seen how you can create an instant red using `Color(1,0,0);` or an instant xyz position with `Vector3(-7,0,0)`. The basic rule for an instant struct is to put the name with parens around the values, for example (this is almost legal): `Cow("Amy",12,1600)`.

The cow in front should make sense. (0,1,1) could be a color, or a Vector3, or a hundred other structs we wrote which happen to have three numbers. We have to say which one it is.

The technical term for the command is Constructor – it constructs a color from those values. Cute, right? Technically `new Vector3(1,2,0)` constructs a struct literal. Remember *literal* is the term for a constant value of a type.

`C#` requires the word `new` in front. We've already seen this in `transform.position = new Vector3(x,0,0);`. Eventually our Cow will be `new Cow("Amy",12,1600);`

We can use constructors to assign to struct variables. Here we set up a Vector3 and a Color using the shortcut:

```
Vector3 pos = new Vector3(5,0,0); // assigns all 3 at once
Color winCol = new Color(0.5f, 0.2f, 0);

winCol = new Color(0,0,0); // can change later with same shortcut
winCol.g=1; // can assign fields like normal
```

Another rule, and this is common in many languages: empty parens gets you all 0's or empty strings. For example `new Vector3()` is the same as `new Vector3(0,0,0)`, and `new Cow()` is a shortcut for initializing the name to `""` and age and weight to 0's.

`new Color();` gets you four 0's, which is a useless see-through black. But at least it follows the constructor rules.

Another rule, also common, is you have to write the useful constructors yourself. This means none of the structs we write will have any until we learn

to write them. We can use `Cow c = new Cow();` to blank it out, but we'll have to make real cows the long way until we write a Cow constructor.

But this rule also means you can write all kinds. If you type "`new Vector3(`" the drop-down shows three versions. Besides all three or none, you can enter just x and y, leaving z at 0:

```
transform.position = new Vector3(x,y); // sets z to 0
```

For color, they added a different shortcut. If you leave off the final transparency, it sets it to you 1 (1 means solid, not transparent at all). I've been using it the entire time:

```
Color c = new Color(1,0,0,1); // solid red (final 1=not see-through)
c=new Color(1,0,0); // short-cut (sets alpha to 1)
```

There's no shorter shortcut, since why would anyone want to enter just red and blue?

Then, remember, constructors are only shortcuts – we can always declare a variable and hand-set each field.

19.6 Nested structs

The fields in a `struct` can be any type, even another `struct`. We usually call that a nested struct, like those Russian dolls nested inside of each other. It's a useful thing to be able to do. There aren't any special rules for this – structs inside structs work the normal way – but they can look funny. This is just a lot of examples.

I want to change the `Cow` struct so each Cow also has two `Color`'s. So that's a struct in a struct. To simplify, I'm getting rid of age and weight:

```
struct Cow {
  public string name;
  public Color mainCol;
  public Color spotCol;
}
```

Here's some code using it. I set each color a different way. For `mainCol` I copy an entire `Color` variable into it. For `spotCol` I copy in r,g,b,a one at a time:

```
Cow c1;
c1.name = "Bessy"; // same as before

Color blk; blk.r=0; blk.g=0; blk.b=0; blk.a=1;
c1.mainCol = blk; // copy a Color to a Color

c1.spotCol.r=0.4f;
```

```
c1.spotCol.g=0.3f;
c1.spotCol.b=0.1f;
c1.spotCol.a=1;
```

The last part with `c1.spotCol.r=0.4f;` is the most interesting. Using more than one dot isn't a new rule – it's just the way the usual struct rule works out. The trick is to read left-to-right. `c1` is a Cow. It has mainCol or spotCol. `c1.spotCol` is a Color, it has r, g, or b.

As usual, once you get to a box, it counts as a variable of that type. `c1.mainCol.g` is a regular float. You can do float things with it, like `c1.mainCol.g+=0.2f;`.

`c1.r` is an error (plus it makes no sense.) We have to use `c1.mainCol` or `c1.spotCol`, then we can add the dot-r. It's really the old can't-jump-over-a-field error.

Here's another example, putting one of my own **struct**s inside another. A `Plumber` has a FullName and an hourly rate:

```
struct FullName { public string first, last; } // no change

struct Plumber {
  public FullName name;
  public float hourlyRate;
}

FullName customer; // for comparison
Plumber p1;
```

The first thing is that this doesn't hurt or change `FullName`. We can still use it like normal, to declare `FullName customer;`. In other words, now that `FullName` is a type, anyone can use it, including Plumber.

Here's a picture of `p1`:

```
p1
------------------
|       ----------
| name | first:
|      | last:
|       ----------
| hourlyRate:
-------------------
```

`p1` has two parts. That's what the definition of `Plumber` says. The first part, `name`, also has two parts: `p1.name.first` and `p1.name.last`.

Here's some code using `customer` and `p1`:

```
// this is just a regular FullName, to show it works the same:
```

```
customer.first = "Lom";
customer.last = "Lomson";

p1.hourlyRate = 24.50f; // nothing special here

p1.name.first = "Larch";
p1.name.last = "Larchenson";

// redo the name:
p1.name = customer; // plumber's name is now Lom Lomson
```

Same as before, `p1.first` would be an error. You can't skip past `name`.

There's no limit to putting structs in structs in structs. You just use as many dots as you need.

19.7 Structs as function inputs

Since a struct is a type, it can be used as a parameter. Here's an example of a function which turns a FullName into a nice string:

```
string commaName(FullName f) {
  string w=f.last + ", " + f.first; // last, first
  return w;
}
```

`commaName(customer);` would give us `"Larchenson, Larch"`. Since structs can't print themselves, functions turning them into nice-looking strings are common.

This one turns a plumber into a string. It's interesting since a plumber contains a FullName, so we can re-use `commaName` for the plumber's name:

```
string PlumberDesc(Plumber p) {
  string nm= commaName( p.name ); // re-using. p.name is a FullName
  return nm+"; Hourly rate: $" + p.hourlyRate;
  // ex: Clark, Stan; Hourly rate: $37
}
```

If plumber names were printed in a different way we'd write it out. Say we only wanted "Clark, $37/hr:

```
string PlumberShortDesc(Plumber p) {
  return p.name.last + ", $" + p.hourlyRate + "/hr";
}
```

Built-in structs can be inputs. This is a somewhat silly function that tells us how bright a color is (it averages red, green and blue):

```
float getBrightness(Color c) {
  float sum=c.r+c.g+c.b;
  return sum/3.0f; // range is 0-1
}
```

Suppose we had some funny loop that cycled `c1` through a lot of colors. `if(brightness(c1)<0.1f)` could be used to test when it's too dark.

Passing structs to functions is another of the main reasons we make them. It's a lot clearer and shorter for a function heading to say it takes a `Cow` than to say it takes a string, int and float.

19.8 Structs as return values

Functions can also return `structs`. There's no special rule for it – these are just examples:

This one returns a random `Color`. I like it because it follows my standard pattern: declare variable for the answer, fill it, return it:

```
Color randCol() {
  Color ans;
  ans.r = Random.Range(0.0f, 1.0f);
  ans.g = Random.Range(0.0f, 1.0f);
  ans.b = Random.Range(0.0f, 1.0f);
  ans.a=1;
  return ans;
}
```

We can use it with `Color c1 = randCol();`. Just so you know, this method tends to get a lot of ugly browns and greys.

This one does a table look-up (pretend we have Colors 0, 1 and 2, and use an `int` to say which one to use):

```
Color numToCol(int colNum) {
  Color ans;
  if(colNum==0) ans=new Color(1,0,0); // red
  else if(colNum==1) ans=new Color(0,1,0); // green
  else if(colNum==2) ans=new Color(0.5f, 0.5f, 1); // light blue
  else ans=new Color(1, 0, 1); // error purple
  return ans;
}
```

It's just a standard table-making cascading `if`, with a none-of-the-above in the final `else` (purple.) I used the constructor short-cuts to make the colors, but didn't have to (I could have used curly-braces and set rgb by hand.)

A sample use is `GetComponent<Renderer>().material.color=numToCol(1);`.

A lot of programs use 0-255 for color values. For example, (255,128,0) is orange. This function makes a usable Color from 0-255 values. For example, `Color c = colFrom255(255,128,0);` will give us (1,0.5,0), which is the orange value in Unity:

```
// lets us make a Color using 0-255 numbers:
Color colFrom255(int r, int g, int b) {
  Color ans; ans.a=1;
  ans.r = r/255.0f;
  ans.g = g/255.0f;
  ans.b = b/255.0f;
  return ans;
}
```

We could use `GetComponent<Renderer>().material.color = colFrom255(210,0,255);` to turn ourself blueish-purple.

The next, slightly fakey, function returns a `Vector3`. It takes a `y` coordinate and returns a point on the left edge of the screen (it just fills in -7 for `x` and 0 for `z`):

```
Vector3 leftPos(float y) {
  Vector3 ans;
  ans.z=0;  ans.y=y;
  ans.x = -7; // left edge
  return ans;
}
```

We might use it like `transform.position = leftPos(-3);`, to put us on the left near the bottom.

It also shows the "field names only matter in the struct" rule. Input `y` is a regular float. `ans.y = y;` is fine. It copies the input `y` into the `y` field.

Finally, we can do both – use a struct for input and output. This one brightens (or darkens) the input Color. It uses the "modify the input" trick, to save declaring an extra variable:

```
Color brighterCol(Color c, float amt) {
  c.r+=amt; if(c.r>1) c.r=1;
  c.g+=amt; if(c.g>1) c.g=1;
  c.b+=amt; if(c.b>1) c.b=1;
  return c;
}
```

It's also an example of a function that seems like it might change you, but doesn't:

```
brighterCol(c1, 0.1f); // does nothing
Color c2 = brighterCol(c1, 0.1f); // c1 unchanged, c2 is a brighter c1
c1=brighterCol(c1, 0.1f); // makes c1 10% brighter
```

19.9 Uninitialized structs

Structs don't automatically initialize any of their fields. As usual, `cow c2; n=c2.age;` is an uninitialized variable error. That seems about right.

But since they have parts, structs have an extra bonus rule: when you use a whole struct, every part needs to have a value. In other words, `c2=c1` or `doCowStuff(c1)` are errors unless every part is filled in.

The idea is, what if `c1` only has the name filled in? `c2=c1;` technically makes `c2` into an exact copy, with only its named filled in. But it seems weird that a variable can have empty parts after an `=`. So they made it illegal.

For example, this function only reads the age. But you need to fill in the whole cow to run it:

```
float normalCowWt(Cow c) {
  return 20.0f + 40.5f*c.age;
}

Cow c1; c1.age=8;
int n = normalCowWt(c1); // error. Passing uninitialized c1
```

This rule is a cool example of how computer languages always have rough spots and have to choose. C# is trying to be safer, so make some things errors that aren't actually a problem.

It also leads to a fun style rule. C# users are in the habit of always using the empty constructor: `Cow c = new Cow();`. It's a simple way to make sure everything is filled in. You don't need to. If you were going to fill every part anyway it's a complete waste. But it doesn't hurt and feels like a "better safe than sorry" thing.

Chapter 20

Struct and Unity examples

This section is more examples with the `Vector3` and `Color` structs, some built-ins and playing with some Unity features that need structs to use.

20.1 More Vector3, Color code

This first thing is a rewrite of the old movement code, to show off `Vector3`'s. It doesn't run any better, but it might look a little nicer.

Since the Unity system thinks position is a `Vector3`, we should store ours in one. Instead of declaring `x`, `y` and `z` for our movement variables, like we did before, we can declare `Vector3 pos;`.

This is the move&wrap code, rewritten with a Vector3 replacing `float x` we had before:

```
Vector3 pos;

void Start() {
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) pos.x=-7;

  transform.position = pos;
}
```

Notice how we still need to set `transform.position` to really move us. `pos` is still a normal variable, with no meaning until we use it for something.

A neater example of this is rewriting the "each lap is a random y" version. We needed global x and y for that before. Now declaring `Vector3 pos;` gives us both. But otherwise the code is the same:

```
Vector3 pos;

void Start() { // no changes in start
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) { // wrap-around
    pos.x=-7;
    pos.y = Random.Range(-3.0f, 3.0f); // <-- new line
  }
  transform.position = pos; // copies the x and y we changed
}
```

This next example is a little different. I'd like to cycle through three different colors. To get the colors, I'll declare three `Color` variables. That's the main point of this example – we could do this by declaring 9 floats (rgb for each color), but it's nicer to use 3 Colors, now that we have structs.

We could set them in the program, but we may as well just do it through the Inspector. Here's the first part, with a function to set us to the correct color:

```
public Color c0, c1, c2; // set these using Inspector
int curColNum=0; // 0, 1 or 2. Which Color we're on now
int delay=40;

void Start() {
  setToCurCol();
}

void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=c2;
  GetComponent<Renderer>().material.color=cc;
}
```

`setToCurCol` is one of those sloppy global-using functions for just this program. It uses the compute-then-use idea: the cascading `if` gets the correct color into temporary Color variable `cc`, then the last line uses `cc` to set our color.

The old way, we'd be changing and setting lots of `r`, `g`, `b`'s. I think this way feels nicer – we assign a color to a color.

The Update part isn't special. It just moves `curColNum` through 0,1,2,0 ..., with a delay:

```
void Update() {
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }
}
```

This part doesn't even use colors. It just slow spins a simple int through the 0,1,2 color numbers. But that's part of the point – most of the work is just moving numbers around.

I want to grow the example a little, to show that `Color` variables are still just variables, and everything is under our control. I'll change the last color to be random, so it shows `c0`, `c1` then a random color. I also want it so pressing the space-bar picks new random `c0` and `c1` colors (we'll see it spin through those exact new colors, until we press space again).

This is the whole thing redone, including my random color function from before:

```
public Color c0, c1; // 2 colors in the sequence (may change)
int curColNum=0; // which Color we're on now: 0, 1 or 2
int delay=40;

void Start() {
  setToCurCol();
}

float rand01() { return Random.Range(0,1.0f); }

Color randCol() {
  Color cc; cc.r=rand01(); cc.g=rand01(); cc.b=rand01(); cc.a=1;
  return cc;
}
```

`rand01()` is a typical "helper" function. All it does is make `randCol` shorter and easier to write.

The rest:

```
void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=randCol(); // <- last color is random
  GetComponent<Renderer>().material.color=cc;
}

void Update() { // no change in this part:
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }

  // pressing space rerolls the first two colors:
  if(Input.GetKeyDown(KeyCode.Space)) {
    c0=randCol();
    c1=randCol();
  }
}
```

This should cycle through 2 familiar colors then a random one, over and over. Pressing space appears to give random colors, but then we'll see same, same, random after a few repeats. It works since Color variables are still just regular variables, which we can change and play with in the usual ways.

20.2 More built-in functions

Most built-in structs have built-in functions to do useful things with them. This section uses two fun ones made for `Vector3`'s: `Distance` and `MoveTowards`.

They're normal functions, but have one bit of weirdness. They're both in the Vector3 namespace – you find them with `Vector3.Distance` and `Vector3.MoveTowards`. C# likes to double-use struct names this way. Vector3 is a struct, and it's also a namespace holding regular functions.

It's nice that you can find built-in Vector3 functions by typing `Vector3`-dot then reading what it says.

20.2.1 Distance

`Distance` takes two `Vector3` points and tells you how far apart they are:

```
public Vector3 v1, v2;
```

```
  float d = Vector3.Distance(v1, v2);
  if(d<2) print("too close");
```

It's a pure math function. It reads the input points and tells you a number.

Now that we know out position is a Vector3, we can check how far we're away from something:

```
public Vector3 target;

void Update() {
  // pretend code moves us

  if( Vector3.Distance(transform.position, target) < 0.2f )
    print("we're at the target location");
}
```

20.2.2 MoveTowards

`MoveTowards` is named for what people usually do with it – move themselves towards some target. It's useful because it will even move the correct distance diagonally.

The official heading looks like this:

```
Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta)
```

Remember, the parameter names are just made up: `current` and `target` are good descriptive names, but don't let `maxDistanceDelta` scare you. It's a float for how far we want to move.

`Vector3.MoveTowards(p1, p2, 0.1f);` says to pretend you're at `p1` and you moved 0.1 towards `p2`. Where would you be? It's another pure math function – it doesn't actually move or change anything – merely returns the spot.

The most common use is: `pos = MoveTowards(pos, target, 0.1f);`. You may recognize that form, it's the same idea as `n=abs(n);`. It changes `pos` because we specially added an `=` to make it change `pos`.

MoveTowards has one extra thing it does: it won't overshoot. `MoveTowards(p1,p2,10);` will move 10 units from `p1` to `p2`, only if they're 10 or more apart. Otherwise it stops at `p2`.

Altogether this code will creep `pos` towards `target` and then stop:

```
public Vector3 pos, target;

void Update() {
  pos = Vector3.MoveTowards(pos, target, 0.1f);
}
```

When it gets to target, the program keeps running. MoveTowards runs every Update, changing `pos` to the new value. But the new value is always the same. It's a neat trick.

20.2.3 More movement examples

Here's a short program using MoveTowards and Distance. It moves the Cube from a random spot, to the center of the screen, over and over. The first line does the movement. The `if` is for resetting. I'm assuming the Camera is in a front-view:

```
public Vector3 pos, targ; // targ starts at 000

void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get close enough:
  float dist=Vector3.Distance(pos, targ);
  if(dist<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);
  }
  transform.position=pos;
}
```

That should move directly to the center, pop away, and repeat. Sometimes it might pop next to the center and almost immediately pop away again, but that's just how random works.

The program works for any target position. We could manually put `targ` in a corner. But it would be cool to have the program change the target as it runs. This adds our old lap-counter to randomize the target every 5 laps:

```
public Vector3 pos, targ;
int laps=0;

void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get there:
  if(Vector3.Distance(pos, targ)<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);

    // change target every 5 trips:
    laps++;
    if(laps>=5) {
```

```
      laps=0;
      targ.x = Random.Range(-7.0f, 7.0f);
      targ.y = Random.Range(-5.0f, 5.0f);
    }
  }
  transform.position=pos;
}
```

The moral here is that we can find some cool functions we didn't know about, like diagonal movement towards a target. But using it in a program is usually the same old tricks.

This version speeds up towards the target. We change the 0.1 movement into a variable. It starts at 0 after each teleport, and increases as we move:

```
public Vector3 pos, targ; // targ starts at 000
float mvSpeed=0;

void Update() {
  pos = Vector3.MoveTowards(pos, targ, mvSpeed);
  mvSpeed += 0.01f; // check: will reach 0.1 in 10 Updates

  float dist=Vector3.Distance(pos, targ);
  if(dist<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);
    mvSpeed=0; // after a pop, needs to speed up from 0
  }
  transform.position=pos;
}
```

20.2.4 Color namespace

`Color` is also re-used as a namespace, holding preset global color variables:

```
public Color c1, c2;

void Start() {
  c1 = Color.red; // <- a pre-made global
  c2 = Color.green;
}
```

The trick is that we're double-using Color. It means 2 different things. As usual `Color c1, c2;` declares variables. But `Color.red` isn't declaring anything. It's how we find a pre-set global variable. It's a shortcut for `new Color(1,0,0)`.

Color also has a function hidden in it. Lerp for colors lets you find a color in-between 2 others. This gives a color halfway between c1 and c2:

```
public Color c1, c2;
public Color cHalf;

void Start() {
  // cHalf is an average of c1 and c2:
  cHalf = Color.Lerp(c1, c2, 0.5f);
}
```

When you see `Color.Lerp` you just have to know we're not declaring a variable. It's a normal function named `Lerp` in the Color namespace.

20.3 Transform struct

After using `transform.position` and `transform.localScale`, you may have realized that transform is a struct. When you type `transform`-dot and get a pop-up, that's the normal struct menu showing the possible fields. The Transform struct looks pretty much like this:

```
public struct Transform {
  public Vector3 position;
  public Vector3 localScale;
  public string name; // <-new
  ... // <- more stuff we don't care about
}
```

Unity automatically declares a global `transform`, of type `Transform`. That's what we've been using this whole time to move around.

`transform` is basically a regular variable. The only thing magic is that the system is always checking it and changing the display based on the values.

We can write a fun program changing the name. It's pretty much the same thing as `c1.name="Bessy"`, except that the system reads the new name and displays it in the object panel:

```
public string name="I stay the same";

void Start() {
  print("Old name: " + transform.name);
  transform.name = "abc" + Random.Range(1,1000);
  // look in Inspector/Hierarchy - name has changed
}
```

I declared `string name;` to show off the local-only rule for field names. Transforms have `name`, but anyone else can have one, too, plus we can declare

our own `name` variable. The system will never confuse them.

`transform.position.x` should now make sense. It's a struct in a struct. `transform.position` is a Vector3, which means you use dot-x, dot-y or dot-z. Typing `transform.x` is an error for the normal reason – you can't jump past a field. You have to say dot-position or dot-localScale.

20.3.1 Special errors; pull-out trick

Arrrrg! `transform.position.x=0;` is an error! The rules say it should be fine, but Unity needed some extra tricks to run correctly and messed it up. You get the error *Cannot modify a value type return value of 'UnityEngine.Transform.position'. Consider storing the value in a temporary variable.* Arrg!.

But that's fine. The work-around is also a good way to program. Instead of reaching inside of something, it's often good to "pull it out", make changes, then copy it back. Here's how to change the position:

```
  Vector3 p = transform.position; // copy the whole position
  p.x+=0.1f; // playing with a Vector3 is fine
  if(p.y>5) p.y=5; // here, too
  transform.position = p; // copy it back
```

This next program does the same thing with the scale. It shrinks the x-scale down to 0.5 while leaving the rest the same. It's not super exciting, but you can reset the scale as it runs, and it will go down again:

```
void Update() {
  Vector3 p = transform.localScale; // copy of the scale
  if(p.x>0.5f) {
    p.x-=0.002f;
    transform.localScale = p; // copy it back
  }
}
```

20.4 Rigidbodies

Most game engines have something they call *physics*. It means you can set up an object to automatically act like a real ball. You can push it once, and it rolls, falls, bounces, and eventually stops. It does all of that automatically, with no script needed.

The reason we care is that we can jump in with programming to change it: the speed is a `Vector3`, and there's a standard programming way to check when we hit something.

Some fun background: the whole name is rigidbody physics. Real objects bend and flex a little. Some bend a lot. Assuming they're all completely solid and stiff – rigid – makes the math easier.

20.4.1 Simple physics set-up

To start, let's get a Cube that falls and bounces around. We'll need a Cube with no script (can remove the script, or make a fresh cube) and a front view Camera (the one showing x and y movement), since gravity makes you fall on `y`. We'll also eventually need a floor and walls (or it won't stay on the screen).

To let the Unity system know it should auto-move the Cube, select it, find `Component` on the top bar, and select `Component->Physics->Rigidbody`. The Cube's Inspector should now have a mini-panel named Rigidbody. Just in case, make sure the ball is somewhere the camera can see it, maybe a little near the top, like (0,4,0), and Play. It should fall, hit the floor, and stop.

To see it bounce more, we can tilt the Cube so it doesn't hit flat. Giving it a z-rotation of 10-30 degrees should make it fall and bounce sideways (just type 20 into the z part of Rotation in the Inspector panel). It should rock just a little, then stop.

The system has a way to set an object's slipperiness and bounciness. As you can see, the default setting is like a bean-bag – not bouncy at all. To give ourselves more to work with, we can make it very bouncy. It takes two steps. We have to make "Bouncy," then apply it to the Cube.

Down in the Assets panel, select `Create->PhysicMaterial`. That should create something with a green picture of a Bounce, named `NewPhysicMaterial`, asking you to rename it. If you like, name it Bouncy (but the name won't matter).

Select the new PhysicMaterial and look at its Inspector. The Bounciness setting is a 0-1 percent for how much it will bounce back. Set `Bounciness` to around 0.9 or so. Then set the `BounceCombine` dropdown to Maximum.

We can use that to make anything be bouncy. Drag that PhysicMaterial onto the Cube. Play should now have the cube really bounce and roll around for a while. It will fall off the edge unless you put some walls there (back in the section about setting up a nicer Scene). Just in case, the actual location of the PhysicMaterial is in the Cube, under BoxCollider (pop it open) in the Material slot (starts with `[none(Physic Material)]`). It will have your new PhysicMaterial if it worked.

That's a lot. I'll sum up, but this stuff is also easy enough to look up, if you know the terms:

- Add a Rigidbody component to a Cube, with no scripts. Play should have it fall.
- Add at least a floor, if you don't have one (a wide, deep, short Cube), so we can see it bounce. Rotate the Cube slightly, so the bottom is tilted and it bounces to one side.
- Create a PhysicMaterial. Set bounciness=0.9; BounceCombine=Maximum. Drag the PhysicMaterial onto the Cube. It should now bounce more.

Plus, if you have trouble with this, there are lots of Unity places to read about these, and what some of the other settings do.

20.4.2 Playing with velocity

The physics system stores our current speed in a `Vector3`. We can't see it in the Inspector, but we can in code. It's `GetComponent<Rigidbody>().velocity;`. This code would fling us sideways:

```
void Start() {
  Vector3 spd = new Vector3(10,0,0); // speed is 10 going right
  GetComponent<RigidBody>().velocity=vel;
}
```

It feels funny since we can set speed once, then let it move by itself. The speed is in units per second, which is why 10 is a good number.

The other odd thing is how the system is constantly changing it. Each Update it applies gravity – it subtracts a tiny bit from `y` the same as we previously did by hand. When the cube hits something, `velocity` is flipped. That's how it makes the bounces.

Normal real-world gravity is about 10 meters a second, each second. If `velocity.y` is 10, it will go up, slow down, and start to fall after 1 second. This will launch us in a high arc, coming back down after about 2 seconds (during which time it will have gone 4 units to the right):

```
void Start() {
  Vector3 vel;
  vel.x=2; vel.y=10; vel.z=0; // strong push up, a little to the right
  GetComponent<Rigidbody>().velocity = vel;
}
```

We can make that a little nicer by having the space bar give us a random extra pop. Pressing space replaces the old `velocity`. We magically change direction:

```
void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    beginRandomPopUp();
  }
}

void beginRandomPopUp() {
  Vector3 vel;
  vel.x=Random.Range(-5.0f, 5.0f); // random left/right
  vel.y=Random.Range(5.0f, 15.0f); // up 5-15
  vel.z=0; // z is towards/away. Don't let it change
  GetComponent<Rigidbody>().velocity = vel;
}
```

A fun effect is to give us a fake bouncy floor. In this next code, dropping below y=-3, gives the random upward pop (make sure your real floor is below this, so it can fall that far. My floor is down at -5):

```
void Update() {
  if(transform.position.y<-3) {
    beginRandomPopUp();
  }
}
```

This gives completely unrealistic bounces since it ignores your current speed. You could be falling slowly and get a big leftward bounce, or the opposite. A variation is to give an upwards push, using `+=` to add a little each update. This next code will gradually slow us, then push us up, giving an effect like bobbing in water. I'm starting the push when it gets below 0, since it takes a while to reverse direction:

```
void Update() {
  if(transform.position.y<0) {
    Vector3 vel = GetComponent<Rigidbody>().velocity; // pull out
    vel.y+=0.3f; // number is trial&error
    GetComponent<Rigidbody>().velocity=vel; // put back
  }
}
```

The main point here is, sure, the physics system is magical and strange. But once we know that `velocity` is a normal Vector3, and what the units are, we can mess with it the regular way.

20.4.3 Collision callback

The usual term for something that could happen at any time is an **event**. A keypress is a typical event.

There are two basic ways of getting events. One way is a command that you have to write, which checks for it. `if(Input.GetKeyDown(KeyCode.A))` is an example of that. That's offiicially called **polling**. I like that name, because it sounds like what you do – you ask every Update, like you're taking a poll.

The other way is having a callback function. You write a function and register it somehow. When the event happens, your function automatically runs.

In the physics system, colliding with something is an event. You react to it using the second method – write a function which Unity automatically runs. The registration is in the name. `void OnCollisionEnter()` is automatically called when you hit something.

Here's a very short collision callback. It makes you a little smaller after each hit. There's no Start or Update (you can also have them, but don't need them):

```
float sz=1; // current size. 1 is normal

void OnCollisionEnter() {
  sz-=0.1f; if(sz<0.1f) sz=0.1f; // shrink down to 0.1
  transform.localScale = new Vector3(sz,sz,sz);
}
```

The neat thing is how ordinary it is. The body is something we already know how to do. If it was in Update, it would very quickly run 9 times and shrink all the way. Nothing mysterious about it.

As a function, it's also very ordinary. Anyone could call it to make us a bit smaller. The heading looks like nothing special. But like Start and Update, Unity decided to key off of that exact name. When there's a collision, it tries to run our `OnCollisionEnter` function because that's the rule it made.

Here's another one that uses a counter to re-drop us after the 4th bounce. It snaps us to the upper middle, with a random sideways speed:

```
int bounces=0;

void Start() { resetDrop(); }

void OnCollisionEnter() {
  bounces++;
  if(bounces>=4) {
    bounces=0;
    resetDrop();
  }
}

void resetDrop() {
    transform.position = new Vector3(0,5,0); // upper-center
    // toss it a little left or right:
    Vector3 toss; toss.z=0; toss.y=0;
    toss.x=Random.Range(-4.0f, 4.0f);
    GetComponent<Rigidbody>().velocity=toss;
}
```

I used a separate `resetDrop` function mostly to show `OnCollisionEnter` can call other functions. If you think about it, Start and Update are magically called by the system, and they can do anything. So OnCollisionEnter can, too.

20.4.4 The Collision struct

An obvious problem with `OnCollisionEnter()` is it doesn't tell you what you hit. Along with that, it might also be nice to know what part of us hit, and other stuff. There's a better version of `OnCollisionEnter` which tells you that.

To make it look nice, Unity created a struct who's only purpose is to hold collision data, named `Collision`. Here's a partial listing for it:

```
struct Collision {
  public Transform transform; // what we hit
  public Vector3 relativeVelocity; // combined hit speed
  ...
}
```

The idea for making this struct is the usual. To give someone collision data you could send them 10 variables, or you could give them 1 Collision variable holding all 10 things.

The name of the first field, `transform` is a good example of the basic struct rules. In our script, `transform` means us. But `colsn1.transform` is a different variable. As a field of a collision, sent to us, it clearly means the transform of what we hit.

To get collision data, write the collision callback with a `Collision` parameter: `void OnCollisionEnter(Collision col)`. The system checks for your function with and without `col`. If it's there, it sends you collision info. This is very standard. Most callback functions have an input describing details of the event.

A key thing is the system makes and send the Collision. You will never have to create one or even declare one. You never need to look at or understand the collision parts we don't care about. In fact, some of them are rarely used by anyone.

Here's a simple collision callback with an input. It prints the name of whatever we hit:

```
void OnCollisionEnter(Collision col) {
  string myName = transform.name;
  string hitName = col.transform.name;
  print( myName + " ran into " + hitName);
}
```

If your floor and walls have been renamed you'll see things like "Cube hit floor" and "Cube hit Rwall". Not very exciting, but it proves we can check for what we hit, using `col.transform.name`.

This next code checks and tries to climb the walls. It bounces normally off the floor, but hitting anything else shoots it upwards. It assumes the floor is named "floor," and works better if you use Start to give a sideways push:

```
void Start() {} // <-- give it a sideways shove here, to hit walls

void OnCollisionEnter(Collision col) {
  if(col.transform.name != "floor") {
```

```
    Vector3 vel = GetComponent<Rigidbody>().velocity; // pull out
    vel.y=Random.Range(5.0f, 10.0f); // snap to an upwards shove
    GetComponent<Rigidbody>().velocity = vel; // put back
  }
}
```

The code only changes the y-speed, leaving the sideways speed alone. When we hit a wall, we bounce off and go the other way, but with a big upwards push. It's a neat effect for such a small, simple amount of code.

We can take a look at other fields inside the collision struct. It has one named `impulse`. Let's print it after each hit:

```
void OnCollisionEnter(Collision C) {
  Vector3 imp = C.impulse; // copy to make the next line shorter
  print( imp.x + ", " + imp.y + ", " + imp.z );
}
```

I still don't know what impulse is, but this is definitely a step to figuring it out.

Notice how I named the input `C` this time. Normally we can name inputs whatever we want. Having OnCollisionEnter called automatically doesn't change that. The system happily fills in the Collision, no matter what we call it.

20.4.5 Misc event notes

In general, any event can be handled either way – polling or a callback – and different systems mix&match. For example, some C# systems handle keys using a callback. It looks a bit like this (I'm making up an example where A and D move us):

```
void OnKeyPress(KeyCode key) { // not legal in Unity
  if(key == KeyCode.A) moveDir=-1;
  else if(key == KeyCode.D) moveDir=+1;
  else moveDir=0;
}
```

When you press `A`, the system would automatically call `OnKeyPress(KeyCode.A)`. Update wouldn't have anything about key presses.

Unity knows about callbacks through the function names. Most other systems have you use a command to register, which also means you can name them whatever you want. For example:

```
// Ex of registering functions. Totally made up and won't run:
```

```
void bounceColorChange(Collision col) { ... }

void Start() {
  // register the bounceColorChange function as a collision enter event:
  collision.enter += bounceColorChange;
}
```

Whenever you want to check for something happening, you have to figure out, is it done with polling, or a callback? If it's with polling, what's the command? If it's a callback, you have to look up how they want you to write the function, and how to register it as the callback.

But then the rest is just programming.

Chapter 21

Float problems

This is a short practical section about the way `float` numbers sometimes round funny, and how we usually handle it. All languages have these problems, and similar solutions.

It's not a basic idea of programming, but to write real programs you probably have to know it. I thought this was a good spot for a relaxing chapter.

21.1 The Problem

In regular math, we get lots of infinitely repeating decimals: 1/3, most square roots, or trig like sin, cosin and PI. We carry them around – our final formula will have square roots and division by 3. As we work it out, some will even cancel out. When we need a final decimal number, we work it out once, from the final equation.

Computers can't do that. Everything needs to be immediately converted into its decimal form. `float a=1.0f/3.0f;` has to immediately round down to 0.333333. This results in more rounding errors and some weirdness. For example `a*3` isn't back to 1. It's 0.999999. If we take the square root of 12, then square it later, we have the same problem – we almost get 12.

Computer numbers have one more source of rounding. They store everything in binary. Binary numbers have a one-halves place, a one-quarters place, one-eighths and so on. In binary, 0.1 is an infinitely repeating number. Many decimals that are fine for us, are repeating and rounded in the computer.

For example, this adds 0.1 to `n` every Update. It never hits exactly 1 – the `if` will never print:

```
public float n=0;

void Update() {
  n+=0.1f;
  if(n==1.0f) print("one"); // this never happens!!
}
```

It's adding a number which is very close to 0.1. After ten times, it's very close to 1, but isn't 1. It's not really an accuracy problem. The main problem is that our `==` don't work. Other weirdness: `10*0.1f` is exactly 1. But `n=0.1f; n*=10;` isn't.

The end result is that computer rounding can seem random. Almost the same formula could give the correct result, or could be off by 0.0000001 higher and lower, depending.

To be clear, this is only a problem with rounding decimals. The computer doesn't have any problems with whole numbers. `float n=1.0f; n+=2.0f;` will always give you exactly 3.0.

21.2 Solutions

These are various tricks to handle off-by-0.000001 problems. These are when you have `if(n==10.0f)` and logically `n` should hit 10, but it hits 9.999999 instead.

Compare using "close enough."

We can fix the Update loop above by checking whether `n` is very close to 1:

```
public float n=0;

void Update() {
  n+=0.1f;
  if(n>0.9999f && n<1.0001) print("one");

  if(n>4.4999f && n<4.5001f) print("4.5");
}
```

The rounding errors are tiny so checking within 0.0001 is safe.

Doing the math inside of the `if` is a pain. This is why closeEnough functions exist. Unity has one built in (we also wrote one, way back):

```
bool closeEnough(float a, float b) {
  float diff=a-b; if(diff<0) dif*=-1;
  return diff<0.0001f;
}

void Update() {
  n+=0.1f;
  if(closeEnough(n, 1.0f) print("one");  // ours
  if(Mathf.Approximately(n, 4.5f)) print("4.5"); // built-in
}
```

This is a brute force approach. You could go through a program and replace every float `==` with a close-enough and it would probably fix everything without breaking anything new.

Don't worry about it

Suppose your program moves `n` up from 0, at various changing speeds, and you want to know when it crosses the edge at 6. Checking `if(n>=6)` is fine. In theory the real math might hit 6.0, while rounding causes it to be only 5.9999 – it will take an extra step to go past. But you don't care; you weren't counting steps anyway.

Or suppose the speed is always 0.1. Now it should clearly take exactly 60 steps to move from 0 to 6. Rounding down might cause it to incorrectly take 61 steps. But, again, you weren't counting and don't care. It might look better with 61 steps.

Something to note is the rounding isn't random. The same math will always give the same numbers. If it falls a little short due to rounding and takes an extra step, it will do that every time.

Account for rounding in your math

Sometimes we can adjust the checked number to account for rounding. This is a safe "are we past 6":

```
if(pos.x>5.999f) // are we past 6, compensate for rounding
```

Even more than that, we might decide we should stop when we're within a 1/2-move of 6. Our adjusted code is `if(pos.x>6-mv/2)`. The exact rounding won't matter any more.

Use integers

If you're storing money, use pennies – store $6.27 as 627. Divide by `100.0f` when you print it.

Say you're storing and printing gallons with fractions. But you only make them using quarts and cups. Since those all go evenly into each other, you can store everything in whole cups. Or you may be able to store everything in teaspoons, or half-teaspoons (as in `int halfTeaspoons;`.

Force it to use integers

There are times when it's not naturally an integer, but you can sort of turn it into one.

For the mover from 0 to 6 by 0.1, it should take 60 steps. I could rewrite it as an int counting from 0 to 60, like this:

```
public int steps=0;

void Update() {
  steps++;
  if(steps>60) steps=0;
  if(steps==30) {}// checking when we hit 3.0 exactly
  pos.x = steps/10.0f; // <- convert steps to actual position
   ...
}
```

The advantage is that my checks can use `ints`, so are perfect. The minor drawback is I have to remember that step 29 is really position 2.9 (or worse, if the range doesn't start at 0.)

Occasional resets

This goes with of the "don't worry about it" trick. If you're using rounded floats, adding a little every Update, you can get more and more off over thousands of additions. That might not be a problem. If it is, it can help to snap to a known value every so often.

An example is the back-and-forth platform. Suppose we add 0.1 from 0 to 6, then flip the speed and subtract, over and over. The rounding will probably cancel out, but it might not. The first time, we may hit 5.999, then the next trip we hit 5.998. Over a hundred trips (which might be a few minutes of time) we can get pretty far off.

If might just look funny on the screen, or we might jump over some important `if`. That's why I like to reset when I hit an edge:

```
  pos.x += xSpd;
  if(pos.x>5.999f) {
    xSpd=xSpd*-1;
    pos.x=6; // <- resets all rounding
  }
```

The rounding errors only have time to build up over 60 moves, which is a tiny amount of error. `pos.x=6;` starts us fresh.

21.3 < vs <= in float compares

Because we know there's rounding in floats, most people are pretty sloppy using `<` or `<=` when comparing them. For example, take `if(x<0)` when we know `x` is slowly moving down. Because of rounding, 99.9% of the time it will be a little

above 0 in one step, then a little below 0 on the next. The odds of it hitting zero exactly are crazy small and also semi-random, so `<` or `<=` will almost never matter.

And if we were trying, we'd have `if(x<0.001f)` anyway. That 0.001 is just a safe number to account for rounding. We just as easily could have picked 0.0005 or 0.002. It's often just simpler to use `<`, since it's shorter and we don't think it will matter anyway.

But this doesn't mean we never care. If you subtract 1.0 from 10.0 over and over, you'll hit 9.0, 8.0 . . . exactly. So `if(x<=0)` will for sure happen when it hits exactly zero.

Chapter 22

Loops

So far our programs can run lines top to bottom, skip some lines with an `if`, or jump to a function and come back. Things which say which order to run lines are called **Control Structures**. The last one is a **loop**. It lets us run lines over and over.

Here's a sample loop which you should never run:

```
// never run this
int n=0;
while(n<1) { print( "moo" ); }
print("done");
```

It runs the line `print("moo");` over and over. Not because it's in Update – because `while` runs the thing after it over and over. This particular loop freezes all of Unity.

More than most things, loops need us to have a plan. We need to think of how to make it repeat enough times to do it's work, and then quit and move on.

This non-useless loop runs 10 times, printing `n=1` through `n=10`:

```
int n=1;
while(n<=10) {
  print("n="+n);
  n++;
}
```

The rules for a `while` are probably obvious: put a true/false test in parens. It works the same as an `if` – runs the body if true. The difference is that a `while` always goes back and checks the test again. The body runs over and over until the test is false. Obviously, something inside must eventually turn it false.

22.1 Special infinite loop warning

This is the first time we can write something which will definitely freeze or crash Unity. It won't do any real damage, but you'll lose anything not saved. and it's a pain in general. If you're trying things out right away, you may want to pause that for a bit until you get a feel for making loops end.

It one of your loops never quits it will freeze everything – the entire Unity window locks. You can't press the Stop button. All you can do is force-quit Unity. Depending on the computer, right-click the icon below and force-quit, or go to the task manager. The rest of your computer will be fine – you'll be able to browse for "how to make a program quit."

Besides being a waste of a minute or two, you'll lose anything unsaved. You won't lose scripts (since you already saved them before running). But any Cubes you created, or moved or renamed, or added a script to – those will be gone.

Before running any risky loops, it's not a bad idea to use save. `Save->Scene` gets most things. `Save->Project` isn't as important, but do it occasionally. It saves changes in the Project panel, such as Materials (the things for colors and bounciness).

22.2 Number-sequence loops

Setting up logic to make a loop run and eventually quit is often known as *driving* it. A common way is with an increasing number. When it gets too big, we quit. This loop counts 2, 4, 6, up to 20 and then stops:

```
int n=2;
while(n<=20) {
  print(n);
  n+=2;
}
```

Let's do a walk-through. The first time is exactly like an `if`: `n` is 20 or less so we run. The body prints 2, and adds 2. Then the new rule for `while`'s sends us back to the test. The same thing happens again: 4 passes, we print 4 and add 2 more to `n`.

Clearly, this will continue, printing 6, 8, 10 .... The last time is on 20. `n<=20` is just barely true. We run, print 20 and `n` goes to 22. The loop jumps back, as usual. But finally `n<=20` is false and we're done.

Basically, the loop made `n` get bigger until it got too big. The details are about which exact values it hits.

We can make the same sequence in a few ways. This version adds to `n` before printing:

```
// not-so-good "increase first" loop for 2-20 by 2's:
int n=0; // start at 0 instead of 2
while(n<=18) { // 18 instead of 20
  n+=2;
  print(n);
}
```

We had to start `n` at 0 in order for it to print 2 as the first thing. Likewise 18 had to be the last number it liked. That feels funny. The loop looks like it should count from 0 to 18. The version with `n+=2;` at the end was much nicer.

Another idea is to figure out the exact number when we stop. In this case, we quit when `n` is 22. Here's how 2 to 20 by 2's works that way:

```
int n=2;
while(n!=22) { // <-change to !=22
  print(n);
  n+=2;
}
```

It works, but it's harder to read and we needed to do more math. If we were counting by 3's we'd need to figure out it stopped on 23. `n<=20` works for anything, which is why we prefer it.

This next one is the same idea as 2 to 20, except 50 to 100 by 10's:

```
int i=50;
while(i<=100) {
  print(n);
  i+=10;
}
```

They're a little spread out, but each value has a spot: start at 50, cut-off at 100, go up by 10.

Here's an interesting one. Try to trace what it prints:

```
int i=1;
while(i<=15) {
  print(n);
  i+=5;
}
```

The numbers will go 1, 6, 11, 16. Sixteen is too big, so the loop prints 1, 6, 11. It's nicer to have the cut-off be the actual last value. `(i<=11)` might have looked better. But sometimes that number represents some real cut-off value. Like we have a 15-day vacation and need to check-in every 5 days, starting when our plane lands.

As an exercise, we can turn the start, step, cut-off pattern into a function that runs a loop with those 3 things as inputs:

```
void printUpSeq(int start, int end, int stepBy) {
  int i=start;
  while(i<=end) {
    print(i);
    i+=stepBy;
  }
}
```

We can use this to quickly test out a few:
`printUpSeq(1,20,1)` prints 1 to 20 – it goes by 1's.
`printUpSeq(0,20,8)` prints 0, 8, 16. It's another one where the cut-off number isn't perfectly aligned.
`printUpSeq(-10,10,5` prints -10, -5, 0, 5, 10. Negative is fine.
`printUpSeq(1,10,99)` only prints 1. After adding 99 it's too big and quits.
`printUpSeq(1,-99, 1)` does nothing. The very first check is `while(1<=-99)`, which is false.

Counting backwards is the same pattern as counting up. We count down, and flip the test. This counts from 40 down to 10 by 1's:

```
int i=40;
while(i>=10) { // using >= which means "not too small"
  print(i);
  i--; // <-going down this time
}
```

We get 40, 39, 38 . . . 12, 11, 10. When it stops, `i` is 9, which is too far.

Suppose we forgot to change one of them. This version forgot to fix the `if`:

```
// this prints nothing. We forgot for change <= to >=
int i=40;
while(i<=10) { print(i); i--; }
```

The first test is `(40<=10)`, which is false. The loop doesn't run even once.

Suppose we forgot to count down, and went up by mistake:

```
// this runs forever:
int i=40;
while(i>=10) { print(i); i++; }
```

It counts up 41, 42, 43 and stops when it's less than 10. So this is an infinite loop. Oops. For real, it will eventually hit 2 billion. Then, depending on the

computer it will error-out and quit, or will keep running with `n` stuck there, or might wrap around to negative 2 billion and actually quit.

To sum up: loops can be *sensitive*. Getting them wrong can give unpredictable results.

Here's a fun general purpose function using all of our tricks so far, making a loop that counts up or down:

```
void printAnySeq(int start, int end, int stepBy) {
  int i=start; // no matter what, start at the start

  if(stepBy==0) { print("can't have stepBy of 0"); return; }
  if(end==start) { print(start); return; } // whole sequence is 1 number

  if(end>start) { // going up:
    if(stepBy<0) stepBy*=-1; // fix negative steps to be positive

    // standard going up (<=):
    while(i<=end) { print(i); i+=end; }
  }
  else { // going down:
    if(stepBy>0) stepBy*=-1; // fix positive steps to be negative

    // standard going down (>=):
    while(i>=end) { print(i); i+=stepBy; }
    // NOTE: stepBy is negative, so i+=stepBy is going down
  }
}
```

We don't need to only add. Anything which changes the number could eventually make us quit. This next loop doubles the number each time:

```
int i=1;
while(i<=99) {
  print(n);
  i*=2;
}
```

It will print 2, 4, 8, 16, 64. In this case the cut-off of 99 makes sense. We're printing every 2-digit power of 2. It's obvious we not trying to hit 99 exactly. Notice how everything else is the same, including `i*=2;` going last.

This is another good chance for an infinite loop. Accidentally starting `i` at 0 makes it so doubling does nothing. `i` would always be 0 and we'd really be stuck.

We can do that in reverse by dividing:

```
int i=150;
while(i>0) { // quit when we drop to 0
  print(i);
  i /= 2; // <- rare "divide me by this" shortcut
}
```

That prints 150, 75, 37, 18, 9, 4, 2, 1. It works because of integer division, which will take anything down to 0.

22.3 While loop rules

Here are the mechanical rules and some comments. You've seen most, but a few might be new:

`while` loops steal most of their rules from `if`'s. The form is:

```
while( TEST ) {
  BODY
}
```

- The test is any true/false, which can be as long and complicated as it needs to be. `while(i<10)`, or `while(i<10 && i>5)` or `while(transform.position.x<6)` or even `while(offEdge()==false)`.
- The parens around the test are required, and you can add all the extra math parens inside. Ex: `while(((i+1)<9) || (i>0))`.
- The body can be any kind of statements, and any number of them. They have to end with semi-colons, even the last one.
- You don't need curly braces if you only want one statement in the body.
- You don't need a semi-colon after the final close-curly-brace. But having one there won't cause an error.
- You shouldn't have a semi-colon after the test. Ex: `while(i<10);`. That ends the `while`. It has no body, so tests, does nothing, tests again, in an infinite loop.

Again, those are the same rules as an `if`. I just wanted to list them since it seems funny how much `if`'s and `while`'s are alike, at least rule-wise.

There's no `else`. It wouldn't make any sense. Actually, some languages have a weird special-case loop-else that might run one time. But a loop never has 2 bodies where it alternates.

The rule where it jumps back to the start is built-in. You don't do anything special. By writing `while` in front the computer knows after it runs the body it has to jump back and try again.

If seems funny how the end of a loop double-jumps, but it's no problem for the computer. What I mean is, take this silly loop that runs once:

```
int i=4;
while(i==4) {
  print("arf arf");
  i=0;
}
```

After the line `i=0;` we know it will quit. But the computer needs to jump back to the start. Then is sees `i==4` is false, and jumps right back to the end.

That sequence means it won't jump out midway. Loops only think about quitting at the end of the body. Here's an example with an `if`. It prints moo:

```
i=3;
if(i<10) {
  i=28; // more than 10, but does not quit. We already checked
  print("moo");
}
```

As a `while` loop it would print `"moo"` once. Only then would it jump back and quit because `i` was too big.

22.4 Do it 10 times loops

A loop is good for doing something 10 times - just write a loop that counts 1 to 10 and put the thing you want to do inside of it.

We almost always think of the counter as "how many times so far." We start at 0 and quit when it hits the count. Imagine counting sit-ups by holding out your fist and flipping up a finger after each one. Like that, a normal "do it 10 times" loop runs on 0 to 9, and quits on 10.

This prints 10 stars:

```
int i=0; // how much we did it so far
while( i<10 ) { // quit when the count hits 10
  print( "*" );
  i++;
}

// what programmers see:
do10times: print("*");
```

We like that style because the number of times, 10, is right there in the loop. And the `<` gives a hint that we care about how many times. If the point was to make 0 to 9 we'd have written `i<=9`.

After a while you can quickly tell a how-many-times loop from a number-sequence-making loop.

Suppose we want to make a string with fifty stars. Running `w+="*";` fifty times would do it. So we make a 50-times loop and put that line inside:

```
string w="";
int i=0;
while(i<50) {
  w+="*"; // <- run me fifty times
  i++;
}

// what programmers see:
string w="";
run50Times: w+="*";
```

Mentally, we group `i=0;`, `i<50` and `i++` as the loop driver for "run 50 times." Then we push those lines aside. We see just `w+="*";` as the real inside of the loop.

It's easy enough to make a generic "run N times loop." This function takes any string and a count and adds that many times. It looks almost the same as the fifty stars loop:

```
string makeCopies(string copyMe, int howMany) {
  string w="";
  int i=0;
  while(i<howMany) {
    w+=copyMe;
    i++;
  }
  return w;
}
```

`w=copyMe("*",50);` would give 50 stars. `w=copyMe("-arf",10);` would make ten arfs. Fun note: if we wanted ten arfs with correct dashes we could combine 9 with and 1 without:

```
// 1 normal arf, then 9 more with dashes in front:
string w="arf"+copyMe("-arf",9); // arf-arf-arf-arf-arf-arf-arf-arf-arf-arf

// or make 9 with ENDING dashes and hand-add the last:
string w=copyMe("arf-",9)+"arf";
```

I think that's a neat example of using return values and abusing functions with cleverness.

These kinds of loops can get harder to spot when we're doing other math. Suppose we want to compute 2 to the 10th. We start with 1 and double it ten times. The key is, it's just a 10-times loop. The doubling is what we're doing 10 times:

```
// compute 2 to the 10th:
int i=0; // loop counter
int n=1; // answer
while(i<10) { // <-- loop runs based on i, not n
  n*=2; // <- this is the real line
  i++;
}
print("answer is "+n); // 1024
```

Mentally, we pull out `i=0`, `i<10`, and `i++` as the do-10-times driver. In our minds `n*=2;` is the real body, being done 10 times.

Another "do X times" plan would be to count down. Imagine both hands with ten fingers up, flipping a finger down after each sit-up until we have two fists:

```
int i=10; // now i means how many we need to do
while(i>0) { // while there are times left
  print("*"); // runs 10 times
  i--;
}
```

There's nothing wrong with this, but everyone does it the other way, making this look weird, and a little more confusing than it needs to be.

22.5 Loops with floats

You can use a float to drive a loop. Everything works the same. This will print from 1.5 up to the cut-off of 10, going by 0.6:

```
float n=1.5f;
while(n<10.001f) {
   print(n);
   n += 0.6f;
}
```

The 10.001 is from the chapter on special float stuff, to account for rounding. When `n` hits 10 exactly, it might "randomly" be rounded to a tiny bit larger. Arrg! This is why we prefer `int`'s.

22.6 Go until done loops

This section is about a completely different loop plan, which doesn't involve counting or a sequence. Sometimes we want to run some lines over and over (and over) until we're "done". A loop is good for that.

There aren't any new rules, but this feels like a different type of loop because it uses a different plan.

Suppose we want to roll 1-6, but not 5's. My first non-loop plan is to act like we have a real 6-sided die. We'll roll it, reroll up to twice if we get 5's, and finally give up and make it some number near the middle, like 3:

```
int nn = Random.Range(1,7); // 1 through 6 integer
if(nn==5) nn=Random.Range(1,7); // got a 5 -- reroll
if(nn==5) nn=Random.Range(1,7); // two 5's in a row -- reroll
if(nn==5) nn=3; // three 5's in a row? Just pick some number
```

We could clearly add more `if`'s to have it be more fair. Or we could change the `if` to a `while`. Then it will re-roll as many times as needed:

```
int nn = Random.Range(1,7);
// reroll until we get something besides 5:
while(nn==5) {
  nn=Random.Range(1,7);
}
```

This is super-cool. That `while` is like an infinite number of identical `if`'s, which cut off when we don't need them any more.

The loop usually won't run, since we usually won't get a 5. That's legal and fine. Loops can run 0 times. But if we happen to get lots of 5's in a row, this will keep rolling until we don't.

That's the basic idea of an until-done loop. We don't care about exactly how many times it runs, or even if it runs at all. We need to repeat something and we'll know when we're done.

I once played a board game where you roll two dice, doubles add and re-roll. If you roll a 4 and a 5, you have a 9. But if you roll a pair of 4's, you keep going. Maybe you roll a pair of 3's, finally a 5 and a 2. Your roll this turn counts as $(4+4)+(3+3)+(5+2) = 21$.

This go-until-done loop does that. The stopping condition is "not doubles":

```
int total=0;
int d1=0, d2=0; // the two 1-6 die rolls (sneaky trick, below)
while(d1==d2) { // while last roll was doubles
  d1=Random.Range(1, 7);
  d2=Random.Range(1, 7);
```

```
  total += d1+d2; // add to total even if not doubles
}
print("roll is "+total);
```

Starting both at 0 is to get things started. We're tricking the loop into running the first time by faking that we got doubles. That's common for loops like this.

Here's a simpler go-until-done die-rolling loop. It rolls 1-100 but not 37 to 50:

```
int nn = 37; // pick any bad value to make loop run 1st time
while(nn>=37 && nn<=50) { // <- our first && inside a while!
  nn=Random.Range(1, 100+1);
}
```

If we roll 87 then we quit right away, which is great. But we could roll bad numbers 39, 47, 42 then finally 17 and quit.

Various math-type problems can be solved with until-done style loops. Suppose I want to find the first power of 2 past 300. My plan is to keep doubling 2 until I get past. The loop is the same as the old doubling loop but in our minds we don't care about the sequence – only finding that one number:

```
// find first power of 2 past 300:
int p=1;
while(p<=300) p*=2;
print("smallest power of 2 past 300 is "+p);
```

I used the no-curly 1-line trick. This just spins 2, 4, 8, 16, 32, 64, 128, 256 then stops on 512, which is the answer I wanted.

A slicker, trickier version would find the highest power of 2 which is `n` or less. For 20 it would tell us 16; asking about 300 would tell us 256. The plan is the same – double until we get past. Then, since that overshoots, divide by 2. We'll write it as a function:

```
int pow2ThisOrLess(int num) {
  // find first power of 2 _past_ num:
  int n=1;
  while(n<=num) n*=2;
  n/=2; // since we overshot, cancel out the last times 2

  return n;
}
```

We should test this. We should run it for every input from 1 to 100 and print the results. A good way to do that is a basic number-moving loop:

```
int num=1;
while(num<=100) {
  print("num=" + num + " pow=" + pow2ThisOrLess(num) );
  num++;
}
// partial output:
// num=7 pow=4
// num=8 pow=8
// num=9 pow=8
```

Using a number-moving loop for testing is a common trick.

Suppose some game uses perfectly square grids, like 4x4, or 5x5, and I want the smallest board which can hold all of the pieces. If one round uses 55 puzzle pieces, it needs an 8x8 board.

The plan is simple. Count up board sizes, 1, 2, 3, until the first one with enough space:

```
int getSizeSquareGridThisOrMoreSpaces(int num) {
  int n=1; // stands for a 1x1 board
  while(n*n<num) n++;
  return n;
}
```

Sample runs:
`getSizeSquareGridThisOrMoreSpaces(8)` is 3 (for a 3x3 board)
`getSizeSquareGridThisOrMoreSpaces(36)` is 6 (perfect!)
`getSizeSquareGridThisOrMoreSpaces(55)` is 8 (7x7 is 49, not quite. 8x8 is 64)

The condition looks really strange, but it's easy. A board size `n` has `n*n` spaces. The loop keeps going if the spaces we have is less than the spaces we need.

As a walk-though, for 8 the loop checks: `1*1<8`, `2*2<8`, and finally quits on `3*3<8`. A 3x3 board is the first one which can hold 8 spaces.

Sometimes the short loops are really sneaky.

Back to the dice examples, I want every Update to roll a number from 1 to 4, but never the same one twice. Suppose the last roll was a 2. My next roll should be 1-4, but not 2. Obviously, this will look like the old "1-6 but not 5" loop.

```
int oldRoll=-1; // previous roll, we can't roll this again

void Update() {
  int roll=oldRoll; // fool loop into running the 1st time
  while(roll==oldRoll) roll=Random.Range(1,5); // 1-4
```

```
  print(roll);

  // save what we just rolled as the old roll for next time:
  oldRoll=roll;
}
```

There were a few tricks here. Saving the old roll in a global is common. Usually we update it as the very last line. As soon as we leave Update, the number we rolled now turns into the old number we rolled.

`oldRoll` starts out-of-range, at -1. That's a trick so the first roll can be anything (it can't be -1, which is no restriction).

The last trick is convincing the loop to go at least once, by starting `roll=oldRoll`. A less sneaky method is to roll once at the start:

```
int roll=Random.Range(1,5); // 1st roll
while(roll==oldRoll) roll=Random.Range(1,5); // re-rolls
```

This is better, since it's easier to read. But worse since we had to copy the "roll 1-4" code into 2 places, even though it's logically the same roll both times.

That code will spray 1-4 in the console. If you let it run for a while and scroll, you'll see it gets all 4 values, but doesn't repeat. That's not absolute proof – it's possible the code has a bug which the randomizer never found – but it should be somewhat convincing.

Sometimes the condition gets too complicated to fit into the test. We can use our old `bool` tricks to precompute it. Here's the "1-6 but not 5" rewritten using a `done` flag:

```
bool done=false; // loop is not done
int nn=0;
while(!done) {
  nn=Random.Range(1, 6+1);
  if(n!=5) done=true;
}
```

This is overly complicated for this small example, but look how pretty it is: the loop runs while it's not done, at the start we say we're not done, and when we get a non-5, we say we're done.

Here's a good use of a `done` flag. Every few seconds a Cube should pop somewhere else, at least 3 away from where it is now. This function finds that new spot:

```
void setNewPos() {
  float oldX=transform.position.x, oldY=transform.position.y;
  bool done=false;
```

```
  Vector3 newPos; newPos.x=newPos.y=newPos.z=0;
  while(done==false) {
    newPos.x=Random.Range(-7.0f, 7.0f);
    newPos.y=Random.Range(-5.0f, 5.0f);

    // set DONE if we're far enough away:
    int dx=newPos.x-oldX; if(dx<0) dx*=-1;
    int dy=newPos.y-oldY; if(dy<0) dy*=-1;
    if(dx>=3 || dy>=3) done=true;
  }
  transform.position = newPos;
}
```

We can take as many lines as we need to decide whether we want to quit the loop. In this case, it took the last 3 lines of the loop.

Here's the rest of it, which doesn't need a loop:

```
int delay=0; // first teleport happens right away

void Update() {
  delay--;
  if(delay<=0) {
    delay=50;
    setNewPos();
  }
}
```

22.7 More oddball loops

The first loops we had moved a number by steps to a target. This is sort of like that, but it will count up or down to 5. We figure out which way each time, inside the loop:

```
// example of strange != loop:
public int num; // input

void Start() {
  int i=num;
  while(i!=5) {
    print(i);

    // Move i whichever direction to get to 5:
    if(i>5) i--;
    else i++;
  }
```

```
}
```

Checking `(i!=5)` makes it seem like we don't know if we're coming from above or below, which we don't. Clearly loops like these are especially easy to mess up and make infinite – often bouncing between two numbers.

Checking for prime numbers is a traditional loop. Basically, try to divide by every number less than you. To speed it up, we'll check for even numbers first, then test-divide by odds, up to half:

```
bool isPrime(int num) {
  if(num==1) return false; // 1 isn't prime
  if(num==2 || num==3) return true; // 2 and 3 are prime

  if(num%2==0) return false; // evens past 2 aren't prime

  int i=3; // start dividing by every odd number up to half:
  while(i<num/2) {
    if(num%i==0) return false; // a number goes into it
    i+=2; // next odd number
  }
  return true; // if we get here, nothing went into it
}
```

The actual loop is a boring 1, 3, 5 ... counter. It abuses the early return trick – if anything divides our number, we can quit and say it's not prime.

The special cases (1, 2, 3, and even #'s) took up half the function. That's common, and often a good idea. It's not as if we get a prize for writing a loop that handles everything.

22.8 Infinite/broken loops

It's really easy to mistype, or just have an idea that's wrong, and get a loop that runs forever. Here are a few ways. Don't try them, since they'll freeze all of Unity.

The most common is forgetting the go-to-the-next. This can happen in longer loops:

```
void Update() {
  int num=0;
  while( num<100 ) {
    // lots of complex checking involving num:
    if(num<10) { cats+= ... }
    else if(num>60) {...}
    else {...}
```

```
    // opps!! forgot num++; It will be 0 forever
  }
}
```

This spins forever with `num`=0, locking up the program.

In this one, I accidentally tried to double 0 over and over. `num` is always 0, so the loop will never quit:

```
// BAD double until 20 or more:
int num=0;
while( num<20 ) {
  num=num*2;
}
print("answer is " + num);
```

Here's another infinite loop, where I accidentally left in the stop condition for a dividing loop, in a doubling loop:

```
// BAD double until 20 or more:
int num=1;
while( num>0 ) { // <- oops
  num=num*2;
}
print("answer is " + num);
```

This runs forever, which we've seen, but it's different since doubling `num` gets very big, very fast. It will hit the maximum int (about 2 billion) quickly. It might have a normal crash with a *number too big*, but don't count on it.

This one forgot the {}'s for the body, so only runs the first line, forever:

```
// BAD print 10 stars:
int count=0;
while( count<10 )
  print( "*" ); // this is the only line in the loop body
  count=count+1; // opps! this is after the loop
```

It seems like computers should be able to detect infinite loops and stop themselves. The problem is, sometimes a loop really needs 5 minutes to run. Some loops are even supposed to be infinite, with a parallel thread stopping them.

This is a repeat, but a mistake can also cause a loop to never run. In this example, I'm counting down, but forgot to flip the `<=`:

```
int i=10;
while(i<=0) { // <- oops 10<=0 is false right away
  print(i); i--;
}
```

This prints nothing. It looks like the computer is just skipping our loop for no reason.

22.9 Move and count loops

I want to start with a mystery loop and puzzle out what it does:

```
public int num;
public int count;

void Start() {
  int count=0;
  while(num>1) {
    count++;
    num=num/2;
   }
}
```

This loop is adding 1 and also dividing by 2. Hmmm . . . . Dividing `num` by 2 is the last thing, so that's a hint. And `while(num>1)` confirms it – `num` drives this loop. It's a "cut in half until it hits 1" loop.

Adding 1 to count is what we do while we watch. So this loop counts how many times 2 goes into a number. Starting it with `num` at 32 it would go 16, 8, 4, 2, 1 and `count` would be 5.

In this next example, I'm curious about how many times it takes to roll a 6. I know it should average 6 times, but I'm wondering about the spread; or how rare it is to take 20 or more tries.

The loop will roll until we get a 6, counting how many times. Update is to make it run over and over:

```
void Update() {
  int rolls=1;
  // count how many rolls until we get a 6:
  while(Random.Range(1,7)!=6) rolls++;
  print("Rolls until we got a 6: " + rolls);
}
```

I'm not even saving the die roll since I don't care what it is unless it's a 6. This is another one that's very easy to mess up. If we accidentally used `Random(1,6)`, we'd roll only 1-5, can never get a 6, and it always runs forever.

22.10 Digit tricks

We've seen the trick where `n%10` gives you the one's place (divide by 10 and take the remainder). We've also seen the trick where dividing by 10 chops off

the 1's place: 327/10 is exactly 32. And the combined trick where `(n/10)%10` gives the ten's place.

A loop can use that to look at every digit in a number.

First let's test the idea. This divides and prints until we hit 0:

```
// test divide by 10 a lot:
public int num=405725; // sample big num with fun digits

void Start() {
  int n=num; // so we don't destroy num
  while(n>0) {
    print( "n=" + n);
    n /= 10;
  }
}

// Output:
n=405725
n=40572
n=4057
n=405
n=40
n=4
```

To make it useful, let's take that same loop and print the 1's place, using the `n%10` trick. This only has that one line changed:

```
  int n=num; // so we don't destroy num
  while(n>0) {
    print(n%10); // print 1's place
    n/=10;
  }
```

This happens to print the number backwards: 5 2 7 5 0 4 (on separate lines.)

A trick to flip them around is using a string and adding to the *front*. A quick test of front-adding:

```
string w="hat";
w = "the "+w; // the + hat
w = "in "+w; // in + the hat
w = "cat "+w; // cat + in the hat
print(w); // cat in the hat
```

Here's the each digit loop changed to add each digit to the front of a string, putting them in order:

```
// nicer "print all digits"
public int num=405725; // input

void Start() {
  int n=num;
  string result="";
  while(n>0) {
    int dd=n%10;
    result = dd+", "+result;
    n=n/10;
  }
  print(result); // 4, 0, 5, 7, 2, 5,
}
```

A cute thing we could do with this is add the `numToWord` function. Change the adding line to: `result=numToWord(dd)+","+result;` to get "four, zero, five, seven, two, five."

We can change the digit-using line to count how many 7's our number has:

```
public int num=374701; // input
public wantNum=7;   // change to any 0-9

void Start() {
  int count=0;
  int n=num;
  while(n>0) {
    int dd=n%10;
    if(dd==wantNum) count++; // <- replaces string-add line
    n=n/10;
  }
  print(wantNum + "'s in " + num + " is " + count);
  // 7's in 374701 is 2
}
```

Even easier, just count how many digits the number has. This is written as a math function:

```
// number of digits function:
int digits(int n) {
  int count=0;
  while(n>0) { // divide by 10 until done loop
    count++;
    n/=10;
  }
  return count; // ex: for 568 count is 3
}
```

The function destroys input `n` as it runs, which is fine, since it's short and `n` is only a local variable.

A sort of interesting non-loop comment: this gives the wrong answer for 0 and negative numbers (-251 counts as 3 digits, right?) It's another example of a nice loop made ugly by needing extra `if`'s in front for special cases.

22.11 Fencepost errors

Counting how many times a loop runs can easily be off by 1. Take a look at this loop that prints 13 to 16, and, without thinking too hard, figure out how many times it runs.

```
int i=13;
while(i<=16) { print(i); i++; }
```

My first instinct is that 16-13 is 3, so it runs three times. But if we count the numbers: 13, 14, 15, 16, it really runs four times.

We think of a number-line as a fence. Each number is a post, with a section of fence stretched between. The part that fools people is a length three fence needs *four* posts: 13 to 16 are three apart and also four numbers.

Whenever you're doing number math, figure out whether you want to count the posts, or the fence sections. If someone tells you to fill 13 to 16 with size one Cubes, you need three – each Cube is like a fence section, running between two numbers. But if someone asks you to place markers on 13 to 16, you need four – the markers are like posts. If you rent storage lockers 13 to 16, that's four, with three adjoining walls.

The most common off-by-one error is like that first example. An obvious one: 10 to 20 are clearly ten apart, but are eleven numbers. `high-low+1` is a common formula. A loop that prints 10 to 20 runs 20-10+1=11 times,

Being off by one because of this problem is most often called a fence-post error.

22.12 for loops

A typical counting **while** loop has the "driving" part spread across three lines. We write so many of these loops, that it might be worth it to combine `i=0`, `i<10` and `i++` into one line. It would make the loop just a little easier to read, and maybe avoid a few mistakes (like forgetting `i++` and getting an infinite loop).

A **for** loop does this. It's just a shortcut, and can't do anything that a **while** loop can't do, but it's very common. They were invented way back and most languages have copied them.

Here's a **for** loop that prints 0 to 9, with the explanation later:

```
for( int i=0; i<10; i++ ) {
  print( i );
}
```

The inside of the `()`'s is a pile of special **for-loop** rules. Declaring `i` and setting it to 0 was moved inside. The same `i<10` is there. And our `i++` has been moved from the end of the loop, to inside the parens.

Here are the official rules:

- A **for** loop has to have 2 semi-colons inside the parens, which divides it into three parts. They aren't normal end-line symbols. They're special required **for**-loop semicolons.
- The middle item is the usual test – it works exactly the same as in a while loop.
- The last item counts as being the last line in the body (but still inside of it). It *doesn't* run when the loop starts, only after the body runs.
- The first item happens once, at the start of the **for** loop. It's used to set up the starting value. You're allowed to declare a special "loop" variable here, but don't have to. If you do, it lives during the entire loop, then vanishes (it's a special type of block scope).
- The first and third item can be blank, but you still need the semicolon. `for(;i<10;)` is the same as `while(i<10)`.

We like **for** loops because they let us gather the loop driving lines. We can look at the top of the loop and easily see how it moves.

Some more typical **for** loops:

```
// count down 10 to 0:
for( int i=10; i>=0; i-- ) {
  print( "time left to explosion: "+ i );
}
```

The top part has the exact three lines we'd use in a `while` loop, even the `i>=0` to show we're counting down to 0. But they're grouped up there, making it easy to see the loop is really one print statement.

This is a basic move-by number loop printing 5, 15 up to 105:

```
// 5 to 105 by tens:
for(int i=5; i<=105; i+=10) {
  print( i );
}
```

The print-power-of-2 loop looks the same. Since the body now only has one line, we can leave off the curly-braces:

```
for(int i=1; i<5000; i*=2) print(i);
// 2, 4, 8, 16 ... 4096:
```

Floats are nothing special, but it's nice to see an example with them:

```
for(float xNow=-7.0f; xNow<7.0f; xNow+=0.8f) {
  print("make something at " + xNow );
}
```

Usually we like how we can insta-declare the variable inside the for-loop, and how it vanishes when the loop ends. But we don't have to. This finds the first power of 2 past `cows` by declaring `n` before the loop, the way a `while` does:

```
int n;
for(n=1; n>=cows; n*=2 ); // do nothing inside -- double until big enough
print("pos of 2 past cows is " + n);
```

After the loop, we still have `n`, with whatever the loop doubled it to.

Most people use a `for` loop for simple sequences, then the odder stuff is a `while`. But you can use either for anything. Here's "how many digits" rewritten using a `for` (I tweaked it a little. It stops dividing at one digit, and starts the count at 1 to account for that):

```
// count digits:
int digits=1;
for(int n=num; n>9; n=n/10 ) digits++;
```

While-not-done loops look nicer as whiles, but some people just love `for`'s. Here's a for-loop to roll 1-6 but not 5:

```
int num;
for(num=5; num==5; num=Random.Range(1,7));
print("roll is "+num);
```

That one is really freaky. The semi-colon means it has no body. It doesn't need one. It rolls 1-6 and keeps going until it gets a non-5. The starting `num=5` makes it run the first time. You'd have to love `for` loops to think this is a good way, but it works.

22.13 Count plus formula loops

When you want to run through a sequence of floats, it's often nicer to use an int loop.

For example, suppose you want 5 numbers that start at 12.5 and go up by 0.8. Use a 0 to 4 counting loop with a formula inside:

```
for(int i=0; i<5; i++) { // 5 times, going 0 to 4
  float pos=12.5f+0.8f*i; // <- this is a very pretty formula
  print("do something using " + pos);
}
```

It's easy to see this runs 5 times. The formula looks pretty good written on one line, and is easy to change. Start `i` at 0 makes it easy to see how 12.5 is the first number.

Even better, we can easily have it count down. Change the formula to `12.5f-0.8f*i`, and that's it.

A little different version, suppose we don't know how many we want. We want to go from 12.5 to at most 20. The int plan still works, using a flag:

```
bool done=false;
for(int i=0; !done; i++) { // count up until done
  float pos=12.5f+0.8f*i;
  if(pos>20) done=true;
  else {
    print("do something using " + pos);
  }
}
```

22.14 do-while loop

A `do-while` loop is the same as a `while` loop, but the test is at the end. The body always runs once, before checking the test the first time. Other than that, it's the same as a `while`. It looks like this:

```
do {
  print(n);
  n++;
} while(n<10);
```

The only difference between this and a while loop is when it would run 0 times. do-while loops always run at least once. For `n=20`, this loop prints 20 then quits (a `while` loop would just quit).

You never need to use one of these. While loops are all you ever need, and `for` loops are a nice shortcut. do-while's are one of the "shortcuts" we tried back in the old days that weren't all that useful.

If you get a chance, look up a repeat-until loop.

Chapter 23

Index Loops

This section is about using loops to look though every letter of a string. For example, finding the first z, or counting how many z's there are. But it's really about using loops and indexes to look through any kind of list. Searching strings is just an easy way to practice.

Once we know how to look through letters in a string, we can use the same tricks to search a list of Customers or enemy monsters.

23.1 Indexes

If you remember, way back I wrote a string is made from individual characters. If you have `string w="cowbell";` it's really a list of 7 characters. `w = "cow"+"bell";` is also a list of 7 characters. It doesn't remember those two parts. Even `""+"cowb"+""+"ell"` works out to the exact same list of 7 characters.

The characters in a string are numbered, starting from 0. Here's the traditional picture of some strings with the numbering underneath:

```
string w="cat";

//  c a t
//  0 1 2

string poem="fish - swimX2.";

//  f i s h   -   s w i m X 2 .
//                      1 1 1 1
//  0 1 2 3 4 5 6 7 8 9 0 1 2 3
```

`"cat"` is obvious – it has three letters, numbered 0, 1, 2. I mixed things up in `poem`, but it's just 14 characters, numbered from 0 to 13. Spaces, dashes, and

numbers all count as 1 character each.

To look up one character, use the number, with square brackets around it:

```
string w = "cat";

print( w[0] ); // c
print( w[1] ); // a
print( w[2] ); // t

string q="cowbell";

print( q[1] ); // o
print( q[2] ); // w
```

The official name for the number in square brackets is an **index**.

Most computer numbering starts at 0. We usually call that a **zero-based index**. Here's a fun list of stuff that happens automatically because we start from zero:

- If a string has 10 letters, they're numbered from 0 to 9.
- The first letter is always `w[0]`.
- If you want the 6th letter, use have to subtract one and write `w[5]`.
- The last letter is always *one less* than the length.

After reading that, it seems like it would have been easier to start from 1, but you get used to 0, and it does work out.

Obviously, indexes are always `ints`. You have to pick out one character (what would `w[1.5f]` even be?)

Using an index really gives us a character. We can assign `w[0]` to a character, or compare it to one. If you remember, the name is `char` and they use single-quotes:

```
string w = "Leopard";
char first = w[0]; // character 'L'

if(w[1] == 'x') {} // 2nd letter is 'x'

if(w[1]=="x") {} // ERROR -- can't compare char and string
```

Indexes *always* start at 0. There's no way to tell a string to start numbering itself at 1. Here's an example where I try to trick the computer (but it doesn't work):

```
string a="cat";
print( a[0] ); // c
a = "were"+a; // renumbers, so 'w' is at 0
print( a[0] ); // w
print( a[4] ); // c  cat pushed down to 4,5,6
print( a[5] ); // a
print( a[6] ); // t
```

Out-of-range errors

Using indexing can give us an exciting new error. `w="cat"` has indexes 0, 1, 2. What should happen when someone uses `w[3]` or `w[50]`? It should be some sort of error. But `w[3]` isn't wrong. It's the correct way to ask for the 4th letter, and `w` could have had four letters. Maybe it will by the time we look.

So, we can't give an error ahead of time, only when we come to the line and have a problem. That's a **run-time** error, which we haven't seen much before. This program crashes:

```
string w="goat";
print( w[3] ); // 't'
w="rat";
print( w[3] ); // ERROR
```

The error message is *IndexOutOfRangeException: Array index is out of range.* That's not too bad, especially since now we know an index is the number in the `[]`'s.

Using a negative index gives the same error. You'd think the computer would know ahead of time that -1 is never legal. But it's simpler to have off-the-edge in either direction be the same error.

This is the first place we'll be getting regular run-time errors, so we should know the funny way Unity handles them. In a regular program, a crash quits the program – tablets go back to the screen, PC programs stall, shut off and bring up the "submit a report" window. If you build and Release a program made with Unity, it crashes that way, too.

But in the Editor it tries to be friendlier. It gives the red error and quits out of Update or Start, but then keeps running more Updates. For example, this will print A, but never B, spraying errors:

```
void Update() {
  print("A");
  string w="cat";
  print(w[3]); // crash - red error
  print("B"); // never reaches
}
```

Each time it hits `w[3]` it crashes, skipping the B, but then runs Update again. So don't let a blast of red errors scare you – it's just the same run-time error over and over.

23.1.1 Length of a string

This is another of those things that uses a real rule for now it's magic. You can find the length of a string `w` by using `w.Length`. Examples:

```
string animal="bear";
print ( animal.Length ); // 4
w="camel";
print( w + " has " + w.Length + " letters"); // camel has 5 letters
w=""; print( w.Length ); // 0
```

23.2 Variables as indexes

The most clever and useful trick with indexes is that you can use variables and formulas. We already knew you could do that with other numbers, but the way it works with indexes is just so extra clever.

These two examples use an all-number equation as the index. They aren't good for anything, except as examples:

```
string w="abcdefg";
// a b c d e f g
// 0 1 2 3 4 5 6
print( w[3+2] ); //  same as w[5], f
print( w[8-2*3] ); //  same as w[2], c
```

These next three are more realistic, using an actual variable, `num`, for the index. Even cooler, since `num` is a variable, we can change it in-between lookups:

```
int num=0;
print( w[num] ); // a
num++;
print( w[num] ); // b

print( w[num+1] ); // c
```

This is really no different than the rule "any place that takes an int, can use anything that works out to be an int."

The most common trick using a variable index is in a loop. Suppose `w` has length ten. It's numbered 0 to 9. If we make a loop counting 0 to 9, we can use the loop variable to look at each position:

```
string w="abcdefghij";
for( int i=0; i<10; i++ ) {
  print( w[i] );
}
// Output (on different lines):
// a, b, c, d ...
```

There's one thing left to make it perfect – we should look up the current length of `w` and use that to end the loop:

```
string w="abcdefghij";
for( int i=0; i<w.Length; i++ ) {
  print( w[i] );
}
```

It's easy to be off by 1, so let's double-check the math: suppose `w` is `"goat"`. That means `w.Length` is 4, and `w` is numbered 0 to 3. The loop will run while `i<4` and counts 0,1,2,3. It exactly hits every index it should. Everything seems to check out.

It's good to check the weird examples, too: what if `w` is just `"a"`. The loop runs while `i<1`. That's once, with `i=0`, which is the `'a'`. It's perfect.

This is going to be our standard look-at-every-letter loop.

23.3 String Loop examples

We can fill in the body of our basic loop to do a few things. I'm going to write these as functions because I think they look nicer.

We can count how many copies of a letter are in a string. I think it makes sense here to have the letter be a `char` input:

```
int timesInString( string w, char countMe ) {
  int count=0;
  for(int i=0; i<w.Length; i++) { // <- standard every letter loop
    if( w[i]==countMe ) // <- checking w[i]
      count++;
  }
  return count;
}

// samples:
print( timesInString("cattle talk", 't') ); // 3
print( timesInString("banana", 'x') ); // 0
```

The loop driver is showing me every letter, in `w[i]`. Inside the loop, `if(w[i]==countMe)` is asking "is the current letter the one we want?"

We can use the loop to make a new string from the old one, one letter at a time. This example adds a slash after each letter:

```
string addSlashes( string w ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    ans = ans + w[i] + '/';
  }
  return ans;
}

// sample:
print( addSlashes("abcd") ); // a/b/c/d/
```

The same idea can reverse the string. We just add each letter to the *front* of our answer. I'm adding a testing line, so we can watch it work:

```
string reverse( string w ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    ans = w[i] + ans; // add to front
    print( ">" + ans ); // testing
  }
  return ans;
}

// sample:
print( reverse("frog") ); // gorf
// >f      <- the testing lines
// >rf
// >orf
// >gorf
```

That seems pretty impressive for such a short loop.

A note: this returns a reversed copy. We've seen this sort of function before. If we want to change a string to be backwards we'd use it like `ani=reverse(ani);`.

Removing a letter can be done by making a 1-letter-at-a-time copy, but using an `if` to skip the letter we don't want. We don't really skip it, we just don't copy it over:

```
string remove( string w, char removeMe ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    if( w[i] != removeMe ) ans += w[i];
  }
  return ans;
}
```

`remove("elephant ear", 'e' );` would give us `"lphant ar"`. It also works if the letter isn't there – it copies everything with no changes. `remove("abc", 'X')` gives you back `"abc"`.

Removing spaces is fun (spaces are perfectly good characters). `remove("in a tent", ' ')` gives us `"inatent"`.

If we want to remove two letters, we can use it twice:

```
// remove < and >:
w="<goat> <cow> <pig>";
w=remove(w,'<'); // goat> cow> pig>
w=remove(w,'>'); // goat cow pig
```

Very similar to `remove` is doing a replace. We still check for that letter, but instead of skipping it, we add the replacement letter. Since the loop body uses `w[i]` twice, I'm copying `w[i]` into a temp character variable:

```
string replace( string w, char oldChar, char newChar ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    char ch=w[i]; // get a copy to save typing
    if( ch != oldChar ) ans += ch;
    else ans += newChar
  }
  return ans;
}

// sample:
print( replace("elephant ear", 'e', 'X' ); // XlXphant Xar

print( replace("cat,dog,cow", ',', ' ' ); // cat dog cow
```

23.3.1 Index inputs/outputs

Once we know about indexes, it seems natural to use them as inputs and outputs. For example, removing the 3rd letter from a string, or finding the position of the first `'a'`.

Everything will use zero-based indexes. If something tells us a letter is at 1, that means it's the second letter. If we want to remove the 5th letter, we'll send it a 4.

This function removes a letter from whatever index you say. It does the usual trick of building a string from letters, skipping one:

```
string removePos( string w, int removeIndex ) {
  string ans="";
  for(int i=0; i<w.Length; i++) {
    if( i != removeIndex ) ans+=w[i];
```

```
  }
  return ans;
}

print( removePos("werecat", 2) ); // weecat
print( removePos("abc", 0) ); // bc
```

The `if` is the interesting part. Before, we compared letters, using `w[i]`. Now we're comparing index numbers, so just using `i`.

Some fun uses: `w=removePos(w,0);` gets rid of the first letter of `w`. And `w=removePos(w, w.Length-1);` gets rid of the last. `w.Length-1` is the mini-formula for the last letter.

Here's a version which removes a range. The first number is the start index, the second is how many to remove. I'm doing it that way, since everyone else does.

The math for the ending index is fencepost stuff. `removeRange(w, 3, 4)` says to remove 4 things, starting at index 3. That works out to 3, 4, 5 and 6. The formula for the end index to skip works out to `start+howMany-1` (testing: 6 is 3+4-1). The function:

```
string removeRange( string w, int startIndex, int howMany ) {
  string ans="";
  int endIndex=startIndex+howMany-1; // "-1" to fix fencepost problem
  for(int i=0; i<w.Length; i++) {
    if( i < startIndex || i > endIndex ) ans+=w[i];
  }
  return ans;
}
```

It's the same as removing one thing, except the `if` checks for a range. Notice how it's a rare "not-in-range" test.

Some examples:

```
removePos("abcdefghijk", 6, 4); // abcdefk ("ghij" is gone)
removePos("werecat", 2, 3); // weat ("rec" is gone)
removePos("abc", 0, 1); // bc
removePos("abc", -10, 999); // "" (but not an error)
```

For fun, here's another way to write `removeRange`, using two loops. Loop one adds everything before the range. Loop two adds everything after:

```
string removeRange( string w, int startIndex, int howMany ) {
  string ans="";
  int endIndex=startIndex+howMany-1;
  for(int i=0; i<startIndex; i++) ans+=w[i]; // front 1/2
  for(int i=endIndex+1; i<w.Length; i++) ans+=w[i]; // back 1/2
  return ans;
}
```

I like this because you have think about not being off by 1. The first loop uses `<startIndex`, since we don't want that one. The second loop begins at `endIndex+1`, since we don't want the letter at `endIndex` either.

We can also write a function for the opposite thing – getting only letters in the range. That's officially called a **substring**. This will loop only over the indexes we want:

```
string substring( string w, int startIndex, int howMany ) {
  int endIndex=startIndex+howMany-1;
  string ans="";
  // loop over only part of the string:
  for(int i=startIndex; i<=endIndex; i++) ans+=w[i];
  return ans;
}

// sample:
print( substring( "werecat bowling", 4, 8) ); // cat bowl
```

The loop from `startIndex` to `endIndex` is obviously right, but only because we've been using that math a lot.

This one can crash on bad indexes. If the start is negative, it crashes right away. If `howMany` is too big it goes off the end and crashes.

A different type of function searches for a letter and tells us the position, as an index. If a word has 10 letters, it tells us either a 0-9 or -1 for not found. It cheats using the return-from-middle trick when it finds one. The function:

```
int indexOfFirst( string w, char findMe ) {
  for(int i=0; i<w.Length; i++) {
    if(w[i]==findMe) return i;
  }
  return -1; // we only get here if no matches
}

// samples:
print( indexOfFirst( "old deer", 'd') ); // 2
print( indexOfFirst( "old deer", 'f') ); // -1
```

It looks funny seeing `return -1;` at the end. But, it's like any other return-from-middle – the answer is -1 only if we didn't find our letter and quit early.

Here's the same thing, but not using return-from-middle. The condition says to stop when we find it or get to the end:

```
int indexOfFirst( string w, char findMe ) {
  int ans=-1; // index of findMe. not found, yet
```

```
  // loop quits at end of word OR when ans is changed:
  for(int i=0; i<w.Length && ans==-1; i++) {
    if(w[i]==findMe) ans=i;
  }
  return ans;
}
```

`ans=-1;` is for the fall-through trick. If we never find the letter, `ans` stays at -1. The loop test also needed to quit when `ans` got a value (otherwise we'd keep going and return the last instead of the first).

This is more awkward, which is why people use return-from-middle.

We can improve find-the-first by adding the ability to say where to start looking – find the first `'e'` past index 4. All we do is start at whatever index they tell us:

```
int indexOfFirst(string w, char findMe, int startPos ) {
  for(int i=startPos; i<w.Length; i++) // the only change
    if(w[i]==findMe) return i;
  return -1;
}
```

A fun use of this is finding the second place something is in a string. Find the first one starting from 0, then look again from just past there:

```
int e2=-1; // position of second e
int e1 = indexOfFirst(w, 'e', 0); // find 1st one
if(e1>=0) // we found an 'e'
  e2 = indexOfFirst(w, 'e', e1+1); // look past location of 1st
print("second e at index " + e2);
```

To find the *last* `'e'` in a string we can use a backwards string loop. This is our first backwards loop, but it's not as exciting as we'd hope:

```
string indexOfLast( string w, char findMe ) {
  for(int i=w.Length-1; i>=0; i--) { // standard backwards string loop
    if(w[i]==findMe) return i;
  }
  return -1;
}
```

It's nice to double-check these. `w.Length-1` is the last one. And it runs `i>=0`, which means it hits 0. So it hits exactly last to first.

23.4 Odd string loops

I'd like to check if one string starts with another, for example if `"theater"` starts with `"thi"` (almost, but not quite). We can do that using a loop which walks through both strings at the same time.

```
t h e a t e r
t h i
0 1 2
```

The loop will run 0, 1, 2, comparing the matching letters. To make it simpler, I'll say the strings have to be in order: longer one first, then the shorter one to check:

```
bool startsWith( string w, string front ) {
  for(int i=0; i<front.Length; i++) { // go to end of shorter word
    if( w[i] != front[i] ) // <- use i to get letters from both words!
      return false;
  }
  return true;
}
```

The new thing here is using the same index for both strings. In two side-by-side lists it's a common trick. But you have to be careful you don't run off the end of the shorter one.

We can make a loop to check whether a string has the same two letters in a row: for examples `"ballet"` (yes) and `"elephant"` (no). We'll compare each letter to the one after it, using `w[i+1]`. This is our first real use of index math.

We have to stop at the second-to-last position, since we can't compare the last letter to the one after it:

```
bool hasDoubleLetter(string w) {
  for(int i=0; i<w.Length-1; i++) { // only to 2nd-to-last letter
    if( w[i] == w[i+1] ) // compare letter at i with letter at i+1
      return true;
  }
  return false; // we didn't find any == pairs
}
```

We can do this another way. Instead of comparing to the letter after `i`, we can compare to the letter before it, We shift the loop by +1: start on the second letter and go all the way to the end:

```
// version where we check the letter before i
bool hasDoubleLetter(string w) {
  for(int i=1; i<w.Length; i++) {
    if( w[i-1] == w[i] ) return true;
  }
  return false;
}
```

A funny thing is that both versions compare the same sequence: 0 and 1, 1 and 2 .... But they "feel" different, and easier-to-read code is important (I

prefer the `i-1` version: skipping 0 is easier, and when you see a loop starting at 1 you know something funny is happening).

This next one is very different. A palindrome is the same forward and backward. The fun ones are sentences, like "Madam, I'm Adam" (if you remove spaces and punctuation and ignore upper/lowercase). But it's really anything where the back half is the front half reversed: `abcdcba`. Notice there's only one `d`. When the length is odd the middle letter can be anything.

The plan is to make two counters, at the first and last letter. A loop will compare them, then move both towards the middle. We quit when they cross. We've never written a loop like this, but the plan sounds good:

```
bool isPalindrome(string w) {
  // start 2 index variables, at first and last letter:
  int i1=0, i2=w.Length-1;
  while(i1<i2) { // keep going until they cross in middle
    if(w[i1] != w[i2]) return false;
    // move both 1 step towards the middle:
    i1++; i2--;
  }
  return true;
}
```

This is the first loop we've ever written where 2 things in the test are changing at once.

Adding a test print is a nice way to check (especially when we get wrong answers). Let's try printing the letters, and the next indexes:

```
 ...
  while(i1<i2) {
    print("compare "+w[i1]+" and "+w[i2]); // <- test 1
    if(w[i1] != w[i2]) return false;
    i1++; i2--;
    print("move to "+i1+" and "+i2); // <- test 2
  }
  ...
```

Running lets us watch it. It might help us find bugs, or to realize our idea was just wrong:

```
isPalindrome("abccba");
compare a and a
move to 1 and 4
compare b and b
move to 2 and 3
compare c and c
move to 3 and 2  <-- the loop quits here
```

If we hate the double-moving-vars plan, we can rewrite it as a regular loop. We'll go from 0 to the middle, using a formula to find the opposite position:

```
bool isPalindrome(string w) {
  int mid=w.Length/2;
  for(int i=0; i<mid; i++) { // loop goes only through 1st half
    int i2=(w.Length-1)-i; // last spot, minus i
    if( w[i] != w[i2] ) return false;
  }
  return true;
}
```

It's comparing the exact same spots as version 1. I even re-used `i2`. Formulas like `(w.Length-1)-i` are easy to get wrong, especially to be off-by-one. I usually write out a test string with numbering.

Here's one last loop with even trickier index math. It checks whether a string *ends* with a certain word. Here's a picture of how we line up `"edwinton"` and `"ton"` to check for a match:

```
e d w i n t o n
0 1 2 3 4 5 6 7
          t o n
          0 1 2
```

We obviously want a simple loop going through `"ton"`, comparing matching letters. The problem is the indexes in the words aren't the same – they're all off by 5. We need to compute the "shift" and then compare using it.

It takes a lot of thinking, but the off-by is the difference in their lengths: 8 for "edwinton" minus 3 for "ton" means we skip the first 5 letters. Otherwise it's like the `startsWith` loop:

```
bool endsWith(string w, string end) {
  // compute so end[0] lines up with w[shift]:
  int shift=w.Length - end.Length;

  for(int i=0; i<end.Length; i++) { // loop through shorter one
   if(end[i] != w[i+shift] ) return false;
  }
  return true;
}
```

This is two tricks in one: using a single index for two arrays, and doing math inside of the `[]`'s. We should watch how it runs with debugging lines:

```
bool endsWith(string w, string end) {
  print("running with ("+w+") and ["+end+"]");
```

```
  int shift=w.Length - end.Length;
  print("shift="+shift);

  for(int i=0;i<end.Length;i++) {
   int i2=i+shift;
   print("compare "+i+"/"+i2); // print the numbers
   print("  "+end[i]+"/"+w[i2]); // and the letters
   if(end[i] != w[i2] ) return false;
  }
  return true;
}
```

A sample test:

```
endsWith("cowbell", "bezl");
running with (cowbell) and [bezl]
shift=3
compare 0/3
 b/b
compare 1/4
 e/e
compare 2/5
 z/l // <- returns false here
```

Chapter 24

Pointers

This section is about a trick where one variable can point to another. We can't use it with the variables we have now – we have to use a `class`, which is like a struct, but uses pointers. Pointers also require you to create variables in a different way than declaring them – you `new` them – so this section is also about that.

That's a lot of stuff at once. This is one of the more complicated things in programming, and especially C#, so don't worry if you have to read it a few times.

But pointers are very useful. They're one of the main concepts of an intermediate level coder. I promise that hurting your head reading this chapter will be worth it.

24.1 First walk-through

Before giving the rules and theory, I'll give a working simple example using classes, pointers and `new` (which is completely different from the `new` with structs).

I'll say what happens in each step, but I'll save the real explanation for later, including what they're good for:

A class is defined and used the same as a struct – fields and the dot-rule. This defines a class `Dog`. It's exactly like a struct `Dog` would be, except for the word `class` in front:

```
class Dog {
  public string name;
  public int age;
}
```

A special rule for classes says you can never declare actual `Dog`'s. When you think you're declaring one, you're really making a pointer to a `Dog`:

```
Dog d1, d2; // not Dogs. pointers to Dogs
```

These are real variables, which can point to any Dog. We have to create the Dog separately. Since we didn't yet, these point to nothing.

We can prove it. Try `d1.name="spike";`. For a struct, that would work. But now we get a *'Use of unassigned variable'* error. It's telling us `d1` isn't pointing to a Dog yet.

The only way to create a real `Dog` is with the command `new Dog()`. It creates a fresh free-floating `Dog` with no name. If we put `d1=` in front, it hooks `d1` up to it:

```
Dog d1, d2;
d1 = new Dog(); // create and hook-up a dog

// d1: o--> | name:
//          | age:
//
// d2 -> (nowhere)
```

In the picture, the box on the right is `not d1`. It's the `Dog` we made with `new`, and it doesn't have a name. `d1` happens to be pointing to it, but that could change.

Even though it has this funny set-up, we can now use `d1` like a struct. These lines understand to follow `d1` to where it points:

```
d1.name="Spike"; d1.age=4;
print( d1.name ); // Spike
```

These dots are doing one more step than the old dots – they have to follow the arrow coming out of `d1`. But the end result is the same.

This next part is where arrows matter. `d2=d1;` causes `d2` to point at the same dog as `d1`. This is a totally new thing and something we can only do with pointers:

```
d2 = d1; // make d2 point where d1 points

// d1: o-\
//        -> | name: Spike
//        -> | age: 4
// d2: o-/
```

There's one free-floating dog, and two ways to get to it. We can show this dog-sharing by using one variable to change and the other to read:

```
d1.name = "rover";
print( d2.name ); // rover

d2.age = 9;
print( d1.age ); // 9
```

If you understand how d1 and d2 are looking at the same dog, this should make sense. If we co-own a poodle and I shampoo it, "your" poodle was also shampooed.

I'll add one more line to the example, another `d1=new Dog();`. It makes a second free-floating Dog with d1 changed to aim at it:

```
d1 = new Dog(); // make a 2nd dog, aim d1 at it:

// d1: ---------------------> | name:  (the second Dog)
//           | name: rover    | age:
//        -> | age: 9
// d2: o-/   (original Dog)
```

d2 is now the sole owner of the original Dog. They no longer share one. `d1.name="Scruffles";` no longer changes d2.

24.2 Terms and Theory

This section has the rules for each part: pointers, how `new Dog()` really works, and then the exact rules C# uses for pointers and `new` together. It's still a lot at once. After this are more real examples of how we use them.

24.2.1 Pointers

A pointer variable can't store anything on it's own. All it's good for is pointing to a real variable. Suppose you have `int a,b,c,d;` and int-pointer p1. The program only has 4 ints in it. p1 isn't an int – it's a way to pick out one of them. p1 can point to a, b, c or d. The most common way to think of a pointer is as an arrow.

Pointer isn't a type by itself. It has to be a pointer to some real type – `Dog d1;` is a pointer to a Dog, and can only point to Dogs. There's no such thing as a generic pointer which can point to a Dog or a Horse or any type. We just use the word pointer as a shortcut for "pointer to whatever type we were just talking about."

Inside, all pointers are really the same – just arrows. Even if Horse was a huge class with 20 fields, a Horse-pointer would still be just a simple arrow.

A pointer variable can do two things: you can make the arrow point somewhere, or you can follow the arrow and change what's in that box. Since it's a

variable, you can make it point somewhere, use it, then make it point somewhere else.

This uses p to change d1, then to change d2:

```
Dog d1 = new Dog(), d2 = new Dog();
// now we have 2 pointers, pointing to 2 Dogs:


//d1: o--> | name:
//         | age:
//
//d2: o--> | name:
//         | age:

Dog p;
p = d1; // aim p at 1st dog
p.name = "Spike"; // use p to change 1st dog
p = d2; // aim p at 2nd dog
p.name = "Rover"; // use p to change 2nd dog

//d1: o--> | name: Spike
//         | age:
//d2: o--> | name: Rover
//         | age:
```

We used `p.name=` twice. But it was aimed at a different `Dog` each time, so we changed two different names.

24.2.2 `new` vs. Declare, Heap vs. Stack

The big thing here is that `new Dog()` is a command. It creates a Dog every time it's run. Normal variables don't work that way.

We need to back up a little into computer theory. There are two general kinds of variables: Stack and Heap. Everything we've used before, including structs, is a Stack variable. The computer is very good at automatically managing those.

For example, variable declarations aren't in the final program. The compiler scans your program, finds all declares, and pre-makes a chart of them. Then it finds everywhere they're used and replaces them with the pre-set spot in the chart. Before the program starts to run, `n=7;` is turned into "memory location 24 = 7".

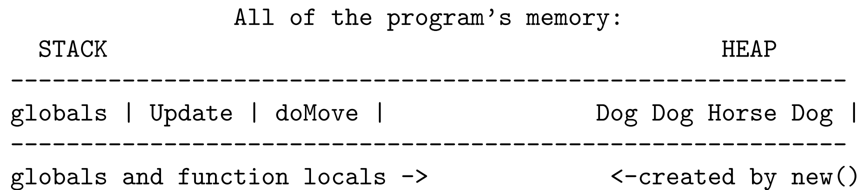
Functions are mostly the same. The compiler does all of that work, using the locals. The resulting chart is called a stack frame. When you call a function, it grabs enough space to hold the stack frame, in one step. Every local variable is created at once. When the function ends, the whole stack frame is popped off.

If you have a chain of function calls, each adds its stack frame on top. That's why the computer never gets confused by re-using the same variable name for different locals. They're stored in completely different stack frames. Computer chips even have this built-in – they have a spot for the start of the current stack frame. Local variable look-up circuits use it.

As cool as those details are, you don't need to know them. The main point is that normal declared variables are pre-made.

As you might guess, Heap variables are the opposite. You start with none. `new Dog()` is a real command that makes a Dog. Dog's are put into a big area with everything else ever created using `new`. That's why it's called a heap. It's a big pile of whatever, in no special order and without very good record-keeping. The only way to find anything on the heap is through a pointer.

The stack starts at one end of memory. The Heap starts at the other. Here's a picture of a program which has run for a while (Update has called the function doMove. And someone ran `new Dog()` 3 times and `new Horse()` once):



When `doMove` finishes, its part pops away. The same for Update. But those Dog's and the Horse are just floating on the Heap, living their lives.

One more example, a useless function that makes some Dogs:

```
void dogMaker(string nm) {
  int n = Random.Range(2,6); // dogs to make
  for(int i=0; i<n; i++) {
    int age=i+2;
    new Dog(); // <- each time it runs, makes a Dog
  }
}
```

When we run this, the system instantly creates `nm`, `n` and even `age` for the loop. When it quits they snap away. No Dogs are made yet. When the loop runs, it creates 2 to 5 permanent Dog's. They don't go away when the function ends. Every time it runs, we get 2 to 5 more Dogs. They all live together on the heap:

```
               All of the program's memory:
  STACK                                              HEAP
------------------------------------------------------------
globals | Update | dogMaker |           D D D D D D D D D |
                   nm:                  D D D D D D D D D |
                   n:                   D D D             |
                   age:         (every Dog ever created)  |
------------------------------------------------------------
globals and function locals ->            <-created by new()
```

Heap variables don't have names. They don't care which function made them. They're just free-floating anonymous Dogs. The only way to use one is if you have a pointer to it. So all of those D's are useless space-wasting garbage Dogs. That's why `new Dog();` returns a pointer – it has to. `Dog d1=new Dog();` is the pointer `d1` catching the return value of the `new Dog();` command.

Things on the heap live forever, sort of. Some systems have a command to individually destroy one. Other systems, like Java and `C#` use what's called Garbage Collection. When a Dog get lost – there aren't any more pointers to it – the system eventually auto-deletes it (more on this later).

Fun facts about other `new`'s:

- A reminder: the `new` from structs, like `v=new Vector3(-7,0,0);` is fake. It's not a real `new` command.
- `C#` actually triples-uses the word `new`! There's another one that goes in front of certain functions. It's totally different. I'm so, so sorry.

I mentioned heap variables don't have names, but it seems like `Dog d1=new Dog();` makes a Dog named `d1`. This shows how that's not true. It uses `p` to make two Dogs, handing them off to `d1` and `d2`:

```
Dog d1, d2;
Dog p; // temp

p = new Dog(); // fresh Dog
d1 = p; // hand the Dog over to real owner, d1

p = new Dog(); // another fresh Dog (re-use p as temp holder)
d2 = p; // hand that one over to d2
p = new Dog(); // p now aimed at 3rd Dog
```

`d1` is aimed at a Dog originally made for `p`. We could think of that Dog as `d1`, unless we plan to hand it off somewhere else. `d2` is the same way.

24.2.3 Reference types

In general, pointers are like add-ons. You can have an `int` and also a pointer to an `int`. But several languages, including C#, simplify this with a trick. They make it so a type can either never use pointers, or has to use them. That's simpler since you never have a choice.

Our old types are the first way – they can't use pointers. So none of these new rules apply to ints, strings. floats or structs. You can't have a pointer to an int, or use `new int()` to create one out of thin air.

A `class` can use pointers. It has to. You can never declare a normal Dog. This is why `class` and `struct` look like the same thing. They are, in both flavors – never pointer, or always pointer. In practice, when you're creating a struct you have to decide – struct or class? Does this need pointers?

A summary, plus some new rules:

- `Dog d1;` is automatically a pointer since Dog is a class. When a programmer sees `Frog f1;`, they won't know what `f1` is until they check struct or class.
- `new Dog()` is the only way to create an actual Dog. There's no way to declare one.
- For a class, `d1=d2;` doesn't copy the contents. It makes `d1` point to `d2`. As easy way is to remember that `d1` is an arrow. `d1=` tells it where to point.
- `d1==d2` checks whether they point to the same Dog. It's always false if they point to different Dogs. Again, remember they're arrows and it's not all that confusing.

The technical term for pointer-only types is *reference type*. Struct is normal. Class is a Reference Type. I personally would have called them Arrow types, but reference is like "refers to", so not too bad.

24.3 Common pointer/class use

In practice, 90% of class variables are used like structs, with no pointer tricks. The rules were specially tweaked to do this. This code would work the same whether Dog's were classes or structs:

```
Dog d1 = new Dog(); // create d1's permanent Dog
Dog d2 = new Dog(); // create d2's forever Dog
d1.name="Spot"; // exactly the way a struct would do it
d1.name+="ster"; // complicated math is also exactly the same
d1.age = Random.Range(2,10+1); // even this -- looks like a struct
```

We declared `d1` to be the forever "owner" of its Dog. We'll never use any pointer tricks. It's easier to think of `d1` as a normal Dog variable. `d1=d2` will result in pointer weirdness, so we won't do that.

The other way we'll use them is as real pointers. When we declare them, we'll mentally decide they'll never "own" anything. They'll only point to other people's dogs. Here `p` is a pointer-style dog:

```
Dog d1 = new Dog(); // a normal-syle Dog

Dog p; // a sneaky dog pointer. Notice no new
Dog p=d1; // p is temporarily peeking at d1

d1.age=8; // normal setting our age
p.age=8; // sneaky p increasing d1's age
```

Technically that makes 1 Dog with both `d1` and `p` pointing to it. But logically it makes Dog `d1` and a pointer temporarily aimed at it.

Here's a semi-realistic example. There are two "real" dogs and one pointer showing the active one. Pressing A or S switches the pointer. The age of the active dog increases:

```
// two real dogs and a pointer:
public Dog pet1, pet2; // "real" Dogs
Dog activeDog; // used to select pet1 or pet2

void Start() {
  pet1 = new Dog(), pet2 = new Dog(); // make the real Dogs
  activeDog = pet1; // select pet1 for now
}

void Update() {
  if(Input.GetKeyDown("a")) activeDog = pet1;
  if(Input.GetKeyDown("s")) activeDog = pet2;
  // add 1 to whichever dog is active:
  activeDog.age++;
}
```

If you see this picture in your head, you've got the idea of pointers:

```
pet1: o-> | name:      pet2: o-> | name:
          | age:                 | age:
                ?     ?
                 \   /
               activeDog
```

`actveDog` selects `pet1` or `pet2`, by aiming at it. `activeDog.age++;` is really increasing pet 1 or 2's age.

24.4 Functions and pointers

Classes can be function inputs and return values. They're still always pointers, but we'll use our two ways of thinking: most of the time we'll pretend they're normal variables; but occasionally we'll use them as real pointers.

24.4.1 Pointer inputs

We can call a function with class inputs and pretend they're structs. This normal-looking function converts a `Dog` into a string:

```
string showDog(Dog d) { return d.name+": "+d.age+" years old"; }

string dWord = showDog(pet1); // Rover: 2 years old
```

There's nothing special about it. But a neat thing: `d` is a pointer. It's aimed back at the actual Dog that called us.

Because of pointers, functions can change their inputs. This fills a Dog with the name and age we give:

```
void setupDog(Dog d, string nm, int howOld) {
  d.name=nm; d.age=howOld;
}
```

`setupDog(pet1, "Gary", 4);` actually changes `pet1`. It works because `d` points back to what called us. Changing `d` is really changing `pet1`:

```
pet1: o-> | name:
          | age:
              ^
Function      |
           d: o
          nm: Gary
         age: 4
```

Even though they're in different areas, `d` can aim at `pet1` and change it remotely, through the magic of pointers.

This next one is about the same. It adjusts a `Dog` away from wrong values:

```
void adjustDog(Dog checkMe) {
  if(checkMe.age<0) checkMe.age=0;
  else if(checkMe.age>25) checkMe.age=25;
  if(checkMe.name=="") checkMe.name="dog";
}
```

`checkMe(pet1);` changes `pet1` for real, if it had a bad age or name.

A common class function copies one into another. The first Dog gets the contents of the second:

```
void copy(Dog toDog, Dog fromDog) { // like toDog=fromDog
  toDog.name = fromDog.name;
  toDog.age = fromDog.age;
}
```

`copyDog(pet1, pet2);` turns `pet1` into a copy of `pet2` without making them share a Dog. This are useful, since there's no built-in way to copy the contents of one Dog into another.

24.4.2 Pointer outputs

When a function looks like it returns a Dog, it's really returning a pointer to a dog. But most of them seem like normal functions. Only a few return a "real" pointer.

Normal `Dog` outputs

The simplest dog-returning functions work like `new` – they create a dog and return a pointer to it.

Here's a very basic one which is merely a shortcut for `new`:

```
Dog makeDog() { return new Dog(); }
```

We could run it like `Dog d1 = makeDog();`. It works since `new Dog()` creates a general purpose free-floating heap Dog. It's not tied to the function like local variables are. When the function ends, that Dog is still there and `d1` is pointing to it.

This version does that same thing using a middle-man variable:

```
Dog makeDog2() { Dog d=new Dog(); return d; }
```

When the function ends, local variable `d` is destroyed. But once again, the actual Dog is still there and we get it. This version shows that even more:

```
Dog makeDog3() {
  Dog d = new Dog();
  d.name="dog"; d.age=2;
  return d;
}
```

It's easier to see how `d` is basically a temporary. It's used to set up the Dog's base stats, before it's passed along to us. `Dog pet1=makeDog3();` gets a 2-year old dog named dog.

Finally, here a useful version, which allows us to give a name and age:

```
Dog makeDog(string dogName, int yearsOld) {
  Dog dd = new Dog();
  dd.name=dogName; dd.age=yearsOld;
  return dd;
}
```

`Dog pet1=makeDog("Spike", 6);` is now a nice substitute for a `new` and two assignment statements.

Let's backup a little and compare `setupDog` and `makeDog`:

```
Dog d1=makeDog("Rufus", 6); // creates Dog and fills it in

Dog dog2 = new Dog();
setupDog(dog2, "Roofus", 5); // this works on created Dogs only

Dog dd;
setupDog(dd, "A", 5); // ERROR - dd isn't aimed at a Dog
```

MakeDog creates a Dog, setupDog assumes you already have one. It's about that extra step classes need to create the real object.

For more fun, suppose we call makeDog twice in a row. It's not really a problem:

```
Dog d2;
d2=makeDog("B",6);
d2=makeDog("BB",8); // abandon 1st Dog for this one
```

That makes 2 Dogs and throws away the first one. No one has a pointer to it, so it can never be found. A picture:

```
 d1: o     | name: B  (original Dog, lost)
     |     | age: 6
     |
     \ ->  | name: BB  (Dog from 2nd makeDog)
           | age: 8
```

It's the same thing that happens if you write `d1=new Dog();` twice in a row. No harm, but a wasted Dog.

Another common Dog-making function is a clone. It creates a copy of another `Dog`:

```
Dog clone(Dog cloneMe) {
  Dog dd = new Dog();
  dd.name = cloneMe.name; dd.age = cloneMe.age;
  return dd;
}
```

It's pretty much the same thing as the function taking a name and age. `Dog d2 = clone(d1);` gets a copy.

To sum up:

```
Dog d1 = makeDog("X", 6); // start with a created d1
Dog d2; // 2 ways to copy it into d2

// both of these make a fresh Dog, with same value as d1:
d2 = clone(d1); // clone creates a Dog
d2 = new Dog(); copyTo(d2, d1); // copyTo assumes you have a Dog

// This makes them share a Dog. Not the same as a copy or clone:
d2=d1;
```

Pointer `Dog` outputs

More rare functions return a true pointer. They give you back something aimed at an existing Dog.

This function takes 2 dogs and returns a pointer to the oldest:

```
Dog oldest(Dog a, Dog b) {
  if(a.age > b.age) return a;
  return b;
}
```

The advantage is that we can use the output like a pointer, like `activeDog` in an earlier example. We can use it to change the selected Dog:

```
Dog dOld = oldest(pet1, pet2); // dOld is a pointer to pet1 or 2
dOld.name += " the elder"; // changing pet1 or 2
```

This next one cheats a little by assuming we have globals named `pet1` and `pet2`. It randomly chooses one:

```
Dog randomDog() {
  if( Random.Range(0,2)==0 ) return pet1;
  else return pet2;
}
```

`Dog pp = randomDog(); pp.age++;` would randomly age pet1 or 2. The logic is the same. In out minds `pp` is merely a pointer, aimed at some existing Dog, used to remotely play with it.

Turn-input-into-output problem

A neat example of how structs and classes are very different is the "change the input" trick. If Dog was a struct, this would give us a baby Dog, named after the mother:

```
// this would work if Dog was a struct:
Dog babyDog(Dog adultDog) {
  adultDog.name += " Jr.";
  adultDog.age=0;
  return adultDog;
}
```

`adultDog` is a copy. It's fine to change it and return it as the answer.

But with Dog as a class this is a mess. First it changes the original Dog, then it returns a pointer to the same Dog instead of making a fresh one. `Dog baby1 = babyDog(pet1);` causes both Dogs to be aimed at the same thing.

To make a fresh baby Dog, we need a `new`. This works when Dog is a class:

```
Dog babyDog(Dog adultDog) {
  Dog baby = new Dog(); // required to create a different Dog
  baby.name = adultDog.name+" Jr.";
  baby.age=0;
  return baby;
}
```

This version is basically a modified clone function. It gives us a copy of `adultDog`, but younger.

24.5 null

Pointers are allowed to point nowhere. The official value is `null` (all lower-case.)

We often use it to initialize a "real" pointer, as a hint:

```
Dog d1=new Dog(), d2=new Dog(); // real dogs
Dog activeDog=null; // pointer, currently aimed nowhere
```

As we all know from movies, a null-ray is the most devastating sort of ray. It's common to think `null` in a program is like that – that it destroys what we're pointing to. But it's harmless. `null` is simply a permanent place any pointer can change itself to, that means nowhere. If it helps, `null` is really just 0.

An example of `null` not causing havok:

```
Dog p = pet1;
p=null; // p moves off of pet1. pet1 is fine
p=pet2; p.age++; // p can be used again. The null didn't break it
```

A common use for `null` is showing that you're done using a pointer:

```
p=oldest(pet1, pet2);
p.age-=2; // oldest dog gets younger
p=null; // p pointing nowhere signals that we're done with it
```

If the function continues, `p=null;` let's us know we're done changing the oldest Dog, and gives an error if we accidentally use `p` again without aiming it somewhere else.

The most special thing about `null` is, obviously, you can't use it. Trying to is an error:

```
Dog d1=null;
d1.age++; // error

d1=pet1; d1.name="X"; // fine
d1=null; d1.name="X"; // error
```

We get a run-time error: *NullReferenceException: Object reference not set to an instance of an object*. That seems clear enough.

We check for `null` using `==` and `!=`:

```
if(d1!=null) d1.name="X"; // this is safe
else print("can't set name -- d1 isn't pointing to anything");
```

Checks for `!=null` are very common. Often they're error checks at times when the variable should never be `null`. Here's a copy function that won't crash if you give it null dogs:

```
void dogCopy(Dog dTo, Dog dFrom) {
  if(dTo==null) return; // nowhere to copy to. May as well quit

  if(dFrom!=null) { dTo.name=dFrom.name; dTo.age = dFrom.age; }
  // if nowhere to copy from, blank us out:
  else { dTo.name=""; dTo.age=0; }
}
```

The correct thing to do on an error can be tricky. If we have `dogCopy(d1, dOoops)` and `dOoops` is accidentally `null`, what should happen? If it really should never be `null`, maybe we should print an error message and crash. That gives us a chance to fix it. Maybe I should have set the name to NULL COPY

and age to -999.

Sometimes pointing nowhere isn't a mistake. Often we want a "nothing selected" option. `null` is the best way to say that. Here's the dog-age-adder from way back where the D-key selects "no dog":

```
void Update() {
  if(Input.GetKeyDown("a")) activeDog = pet1; // pet1 and pet2 are globals
  if(Input.GetKeyDown("s")) activeDog = pet2;
  if(Input.GetKeyDown("d")) activeDog = null; // <- no dog selected

  if(activeDog!=null) activeDog.age++;
}
```

Logically, `activeDog` can can be 3 things, each as good as the other: pet1, pet2, or nothing.

The common term for a pointer set to null is a **null pointer**. For example `if(p!=null)` is checking for a null pointer.

24.6 Errors

Almost all the errors we get will be crashes from trying to follow a pointer set to `null`. The error message is: *nullReferenceException. Object not set to an instance of an object.*

A common way to get that error is forgetting to `new` it. If you remember, all global variables are auto-inited to 0 or `""`. Pointers are auto-inited to `null`:

```
Dog dg; // global starts at null

void Start() {
  dg.age = 5; // null reference exception
  int n = dg.age; // same error (except it crashed on the line above)
}
```

This is a run-time error, which means the program will run normally, then crash when it comes to the line. As usual, all of Unity won't crash. Getting these is no worse than getting any other error (except an infinite loop).

The same problem with a local Dog gives a little different error:

```
void Start() {
  Dog d1; // forgot the new. We think d1 is a struct
  d1.name="Canine"+Random.Range(1,10);
  d1.age=0;
```

We get a normal error this time *Use of unassigned local variable 'd1'*. That's because `d1` is still a pointer, and local variables aren't automatically initialized. It's not even `null`. The computer can tell it's definitely an error to follow an arrow which has never been set.

24.7 Pointer compare (==)

Oh, no!! More rules! These are some examples of the pointer `==` rule. If you remember, `d1==d2` checks whether they are sharing the same Dog.

Here's an example why we like this rule. The A key switches between pet 1 and 2:

```
void Update() {
  if(Input.GetKeyDown("a")) { // switch between dogs:
    if(activeDog==pet1) activeDog=pet2;
    else activeDog=pet1;
  }
  activeDog.age++;
}
```

This works even if the pets have the exact same values. `if(activePet==pet1)` checks whether we're pointing exactly at `pet1`.

Then here's a wrong way to use pointer `==`. The test is always false since they are two different Dogs:

```
if(pet1==pet2) {
  print("please re-enter dogs using different data");
}
```

We'd need to write it out: `if(pet1.name==pet2.name && pet1.age==pet2.age)`. Don't get confused about `==` with non-pointers. `if(pet1.name==pet2.name)` is a normal string compare.

24.8 Garbage Collection

This is a fun section that's not super important. The short version: remember how in the top section I showed how you could waste memory by creating lots of Dogs with `new`'s you didn't need? That's not a problem. The computer will eventually clean that up. Don't worry about using too many.

The slightly longer version: those extra Dog's are called *garbage*. If enough get made, memory runs out and the program crashes. Every so often, C# automatically hand-checks and deletes them. That's called *garbage collection*.

And now the long, boring version: the computer is excellent at managing regular variables, using the stack frame idea. Heap variables – Dogs made with `new` – are more versatile, but the system can't manage them. When you lose your last pointer to one, the system doesn't know.

The only way to find useless Dogs is to look at every declared Dog in the entire program, see where they point, mark those Dogs as in use; then go back and remove every unused Dog. That's garbage collection. It happens automatically. There are some tricks, but it's not fast.

In theory, a sloppily written program, that creates and loses lots of extra Dogs, will have a small hiccup every few minutes. It's pausing to run garbage collection to free up memory. In practice you don't notice.

It's common to make garbage without trying. This program has a 1% chance to reset the Dog's name and age, except it uses `new`, so turns the old Dog into garbage:

```
public Dog d1;

void Update() {
  ...
  // 1% chance to reset the Dog's name and age:
  if(Random.Range(0,100)==0)
    d1 = new Dog("Spot",0); // creates new Dog. Old one is garbage
    // non-garbage way:
    // { d1.name="Spot"; d1.age=0; }
}
```

It's not many extra Dogs, so it's fine. I think the "slow" version with `new` is easier to read

If you know you're done using a Dog, it seems as if there would be a command to delete it, just to speed things up, but there isn't. `d1=null;` is the closest we can come. The next garbage collection step will clean up your old, lost Dog.

The Reference Type system was invented to make automatic garbage collection work. That's partly the reason we can never have pointers to `int`'s or declare a normal Dog. Eliminating those options made garbage collection possible.

There are two other systems, not in C#, to handle garbage. C++ has no garbage collection. Instead it has a `delete(d1)` command. If you forget to use it, the garbage is there forever. C++ programs are very careful that every `new` has a `delete`. In return, C++ pointers can be from any type and can go anywhere.

Other languages use Reference Counting. Every Dog has an extra counter for how many things point to it, automatically updated. When the count hits 0,

the system destroys it. The same as C#, you don't need to do anything. It runs a little slower, but makes it up by not needing to pause for garbage collection.

24.9 Why struct new's?

C# purposely makes it so classes and structs use `new` to mean different things. That's odd. Let's compare them again, then try to see why they made that rule.

Vector3 is a struct. It's a normal variable. `new` is only used for the optional constructor shortcut to fill in several variables at once. But for a class, `new` is required to make it exist:

```
Vector3 v; // v now exists and is ready to use
v = new Vector3(); // not needed. Shortcut to set to (000)

Dog d1; // there are no Dogs, just a pointer to nowhere
d1 = new Dog(); // required. creates the Dog
```

We don't normally like to double-use like this. For example we invented `==` for comparing since we didn't want to use `=` to mean assign and compare.

For `new`, the idea is that not everyone knows class vs. struct very well. Here are the "real" commands (this is how every other language does it):

```
Vector3 v = Vector3(4,0,1); // simple assign
Dog d1 = new Dog(); // create a Dog on the heap
```

That's nice – `new` means to actually make something. No `new` means you're not making something new.

But C# was partly written for beginners to just jump in. They don't know what `new` is, and why you'd use it one place but not the other. For them they added the rule that structs get a fake `new`. Everything looks the same.

24.10 Old-style pointers

C# has real pointers. You aren't allowed to use them, but it might be nice to see them, for comparison. They require you to write out every step, so you get to see what the shortcuts are doing.

To repeat, you aren't allow to use these. They only run in "unsafe mode", and no C# programs are ever written in unsafe mode (this includes Unity programs).

This example declares `ints` and pointers to `ints`:

```
int n1, n2; // normal ints
int* p1, p2; // pointers to ints (int* means "pointer to int")
```

Remembering `int*` means "pointer to an int" can be a pain. That's why `Dog d1;` just knows to make a pointer to a Dog. But being able to write both types of things can be nice.

Likewise, we can point to a real int, but need another symbol:

```
p1 = n1; // error - can't copy an int into a pointer to an int
p1 = &n1; // p1 points to the real box n1
p2 = p1; // so does p2. Copy pointer to pointer
```

This is cool because the types always match. `p2=p1;` is two int-pointers. `&n1` calculates a pointer to `n1`, so the types also match. It's confusing at first to remember those symbols, but once you do, it's easier to read.

We have to use a special symbol to follow a pointer's arrow:

```
p2 = &n1; // p1 aims at n1
p2 = 5; // error. Trying to make p2 point to "5"
(*p2) = 5; // follow arrow and change n1
```

But once you figure out the rule, `(*p2)=5;` spells out how `p2` is a pointer remotely changing something.

Those written-out rules allow us to use `=` both ways, pointer assign and copy:

```
*p1 = *p2; // copy the data between the boxes they point to
p1 = p2; // make p1 point where p2 does
```

You don't need struct vs. class with real pointers. A struct can be either:

```
Cat c1; // actual cat
Cat* cp = &c1; // pointer to c1
c1.name="Tabby"; // change directly
(*cp).age=5; // change c1 using the pointer
```

And again, this is completely not important to know. But you might get a feel for reference types. These are the original rules.

Chapter 25

Pointers in Unity

A basic use of pointers is to aim at someone else's stuff and play with it. Unity has a bunch of Cubes. We should be able to aim pointers at them.

A main reason for using a fancy program like Unity is how easily it allows us to do that. There will be an empty slot, and we'll drag in the Cube we want. Inside our program, we'll have a properly aimed pointer. In Unity, that slot will be in the Inspector.

Once we can get pointers to several Cubes, we can use our old tricks to move them around.

25.0.1 Pointers to existing Cubes

The things in a Unity scene are `GameObject`'s, configured to be Lights, Cameras, and Cubes. GameObject is a regular class, which means we can declare `GameObject g1;`. It will be a pointer.

The class has a field named `transform`. It's the transform we've been using for ourself all along. If `g1` is a GameObject, we can move it with the familiar-looking `g1.transform.position = new Vector3(-7,4,0);`.

Put together, this script, on any object, will let us reach out and move some other Cube:

```
public GameObject g; // drag in a Cube

void Start() {
  g.transform.position = new Vector3(0,4,0); // just some spot
  g.GetComponent<Renderer>().material.color = new Color(1,1,0); // yellow
}
```

As usual, `g` appears in the Inspector. It's currently `null` but it says `GameObject(none)` to be a little nicer. We're suppose to find a Cube, in the panel with names, and drag it in. When our program runs, `g` will point to that Cube.

That's almost too easy. It's the reason programs like Unity exist. They let us cheat by setting up pointers without needing to write code.

Before using them more, let's check that `GameObject` is mostly a normal class. We're allowed to use `new` with it:

```
void Start() {
  GameObject g = new GameObject();
  g.transform.name="fire engine";
}
```

When we run, an empty object named fire engine appears in the panel. That's a slight cheat – Unity tracks GameObjects. It won't track our own classes, Dogs, for example.

Back to our regular examples, this one uses two pointers. You'd drag a different thing onto each:

```
public GameObject g1, g2; // drag in 2 different cubes

void Start() {
  // reach through each to change colors:
  g1.GetComponent<Renderer>().material.color = new Color(0, 1, 0); // green
  g2.GetComponent<Renderer>().material.color = new Color(0, 0, 1); // blue
}
```

Nothing special, but it's new. We weren't able to change 2 different things before, only ourself.

We can test this a little by dragging the same Cube into both slots. It's legal for 2 pointers to go to the same place. It will turn blue. `g1` turns it green, then `g2` immediately turns it blue.

We could leave `g1` null. Click the little circle on the far right of `g1` and select None from the dropdown. We'll get a standard null reference exception. `g2`'s Cube won't change – the error cut off that line.

A fun example which moves two Cubes at different speeds:

```
public GameObject c1, c2; // drag in two Cubes
int x1=-7, x2=-7;

void Update() {
  x1+=0.1f; if(x1>7) x1=-7; // move&wrap cube 1
  x2+=0.07f; if(x2>7) x2=-7; // move&wrap cube 2

  c1.transform.position = new Vector3(x1, 2, 0); // place both cubes
  c2.transform.position = new Vector3(x2, -1, 0); // using different y's
}
```

This is nothing new, except it's neat to have one script run a Cube race.

We can use the trick where a 3rd pointer chooses one of them, like the `activePet` example from the last chapter:

```
public GameObject c1, c2; // drag in two Cubes
GameObject g; // choose between c1 and c2

void Start() { g=c1; } // using 1st Cube, for now

void Update() {
  if(Input.GetKeyDown("a")) {
    if(g==c1) g=c2; // flip g between c1 and c2
    else g=c1;
  }

  // move whichever g points at:
  Vector3 v=g.transform.position;
  v.x+=0.1f;
  if(v.x>7) v.x=-7;
  g.transform.position=v;
}
```

An alternate way of getting a pointer is the `Find` command. It searches through all gameObjects in the panel, by name, and returns a pointer. This mini-program finds `"cube1"` and `"cube2"` and turns them different colors:

```
void Start() {
  GameObject c1, c2;
  c1=GameObject.Find("ball1"); // <- aim c1 at ball1
  c2=GameObject.Find("ball2"); // <- aim c2 at ball2

  c1.GetComponent<Renderer>().material.color=Color.red;
  c2.GetComponent<Renderer>().material.color=Color.blue;
}
```

Notice how `Find` uses the namespace trick. It's name is `Find` and it's in the `GameObject` namespace.

The moral here is that once you have a pointer, it doesn't matter how you got it. We can use `c1` and `c2` in the usual way.

Finally, let's use `null` some more. This next code hopes `c1` was preset. If not, it guesses some names. It uses the standard null-check:

```
public GameObject c1; // drag something in?
```

```
void Start() {
  // try to fill c1:
  if(c1==null) // they forgot to drag something in
    c1=GameObject.Find("clover");
  if(c1==null) // we couldn't find "clover"
     c1=GameObject.Find("rabbitFoot");

  // use c1 if it goes anywhere:
  if(c1!=null)
    c1.GetComponent<Renderer>().material.color = new Color(0,1,0);
  // if it's still null, we're just giving up on it
}
```

As you might guess, `Find` returns `null` is there isn't anything with that name. `null` is really the perfect not-found value.

25.1 Instantiate

Unity has a type of clone function named `Instantiate`. It takes a pointer to one gameObject, makes an identical new one, and returns a pointer.

25.1.1 Instantiate as a copy

Here's some simple code making 10 random copies of a Cube. Warning, warning, warning: do not use this to copy yourself. The copy will run its copy of the script, copying itself, and so on forever. `c1` should point to something without this script on it:

```
public GameObject c1; // drag in a Cube to copy

void Start() {
  for(int i=0; i<10; i++) { // basic 10-times loop
    GameObject bb = Instantiate(c1);
    // bb is now aimed at the freshly made copy of c1

    Vector3 pos; pos.z=0;
    pos.y=Random.Range(-5.0f, 5.0f);
    pos.x=Random.Range(-7.0f, 7.0f);

    bb.transform.position = pos;
  }
}
```

In a normal program we'd need to save those 10 pointers somewhere, or else we're simply making time-wasting garbage. Here we're taking advantage of how

Unity tracks all gameObjects.

This next one is mostly the same, except it makes them 1-at-a-time, when we press space. Then, to use one extra pointer, the previous one also turns red:

```
public GameObject ball1; // drag in some other item
// the most recently created ball:
GameObject previousBall;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    // change the most recent ball's color:
    if(previousBall!=null)
      previousBall.GetComponent<Renderer>().material.color=Color.red;

    GameObject newBall = Instantiate(ball1);
    // move to random spot on screen:
    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    newBall.transform.position = new Vector3(x, y, 0);

    previousBall=newBall; // update to the new most recent ball
  }
}
```

Notice how `ball1` never changes. Its job is to remember the "master ball" for the Instantiate copy command. Meanwhile `previousBall` is a typical moving pointer, temporarily remembering one ball.

Instantiate is also overloaded. We're allowed to give it a position where we want the new item:

```
public GameObject ball1;

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    Vector3 newBallPos; // pre-make target position
    newBallPos.x=Random.Range(-7.0f, 7.0f);
    newBallPos.y=Random.Range(-4.0f, 4.0f);
    newBallPos.z=0;

    // clone ball1 and set the position:
    GameObject bb = Instantiate(ball1, newBallPos);
  }
}
```

That's a typical time-saving overload. All it does extra is run the `position=` line which we ran before. But 1 line saved for something we do often is a useful shortcut.

25.1.2 Proper Unity use of Instantiate

This entire section has nothing to do with programming. But if you use Instantiate with Unity for real it's probably good to know.

Suppose we want lots of explosions using Instantiate. We need a master explosion, but it should be hidden – off the edge of the world or something – and also frozen somehow. That's a pain, so Unity provides a nice way.

Take any Cube and drag it down into the Project window. Unity copies it there, with a corner-cube icon. It counts as a Cube, but it's not visible anywhere and its scripts won't run. It's the perfect master object for copying. Unity calls that a *prefab*.

Any of the scripts above work with these. Find a prefab (cube icon which you made in the Project panel) and drag it into the `gameObject` slot. Your script is now making new objects out of thin air.

The whole process looks like this: suppose you know how to make explosions. Make one (using a particleSystem), then drag it into Project to make a prefab from it. Delete the original. Anyone with `public GameObject blast;` can now get a link to the master explosion, for copying.

As long as we're here, we may as well see the other use of Unity prefabs. Once you have a prefab, you can quickly make real copies of it by dragging into the scene. More than that, the copies are linked. Changing the prefab (the original, in the Project panel) changes all copies.

But don't let that confuse you about how programs work. In a program, copies are copies – there's no link to the original.

25.1.3 Pointers for "caching"

You may have noticed `GetComponent<Renderer>().material.color` is a 3-stage lookup. It calls a function to get our renderer, then the material, then finally the color. Material is a class, which means we can get a pointer to one. We can create a shortcut pointer to our own material:

```
Material myMat; // will be a short-cut to our Material

void Start() {
  // create the shortcut, using a partial look-up:
  myMat = GetComponent<Renderer>().material;
}
```

```
void Update() {
  // 1% chance to turn red or blue:
  if(Random.Range(0,100)==1) myMat.color = Color.red;
  if(Random.Range(0,100)==1) myMat.color = Color.blue;
}
```

The old stuff is still there. Start has `GetComponent<Renderer>().material`, which it saves, and Update has dot-color. `Material myMat` is a pointer, since Material is a class.

Not super useful, but it's pretty neat that we can do it. We can't get a shortcut to color, since Color is a struct. You can never have a pointer to a struct.

We can do the same trick with not-us. Suppose we have a link to the floor, and often want to change the floor's color. We can save the floor's material as a shortcut:

```
public GameObject floor; // drag the floor into this
Material floorMat; // saved shortcut for changing floor color

void Start() {
  // set up floor color shortcut:
  floorMat = floor.GetComponent<Renderer>().material;
}

void Update() {
  // increase the floor's red and wrap it around:
  Color cc = floorMat.color; // using the shortcut
  cc.r+=0.02f; if(cc.r>1) cc.r=0;
  floorMat.color=cc; // using the shortcut
}
```

I think `floor.GetComponent<Renderer>()` is pretty neat.

25.2 New'ing Inspector variables

You may have noticed that our classes in the Inspector don't need `new`'s. Unity does it automatically. But we're allowed to use `new` if we want:

```
// recall we need this line to make it display:
[System.Serializable]
public class Dog {
  public string name;
  public int age;
}
```

```
public Dog dg; // now dg is in the Inspector

void Start() {
  dg.age=7; // legal, it's been new'd magically
}

void Update() {
  if(Input.GetKeyDown("a"))
    dg=new Dog(); // new blank Dog
}
```

Pressing the A key does the usual thing: it creates a fresh Dog, abandoning the old one. The Inspector looks like it's only erasing the age, but it's really tracking that fresh Dog.

Put another way, do anything you want with classes visible in the Inspector. It won't restrict anything – it merely tracks and displays the current one.

Chapter 26

Pointer examples

This chapter is more Unity-type things that show off various pointer and class tricks.

26.1 Text change

A label on the screen is just another thing you can create: `GameObject->UI->Text` makes one. It makes something called Canvas, which we can ignore, with the Text inside of it.

If you select the Game tab, on the big window, you should see a small, ugly "New Text" in the lower-left corner. That's good enough for testing, but here are some non-programming Text notes:

- Color is near the bottom (in the Text's Inspector, in the area labelled Text.) It starts as an ugly dark grey. As usual, clicking it brings up the Color Picker. Change it to pure black, or all white, or green . . . .
- Notice how the top now says RectTransform and has PosX and PosY. Positioning Text uses special coordinates. `(0,0)` is centered, and the screen is a few hundred across. If you want it centered on the top, try `(0,200)`. As usual, you can slide-drag the numbers (put the mouse slightly above the box.)
- The contents is the big box labelled `text`. Just so you know, it will let you enter multiple lines (they won't fit, at first.)
- To make the text larger you can slide drag where it says FontSize (in the middle of the Character sub-heading, under Text.) But it will wink out if it gets too big. Here's how to really do it:

  Go to Scene view and double-click either Canvas or your Text. This should snap your view there. Use scroll-zoom and right-button-spin or the upper-right gizmo to get a mostly head-on view. You should see a big, white

border – that's the entire "Text screen." If you click on your Text object, you should see a faint white border around it.

If the white borders are hard to see, you could turn off the fancy sky. In Scene, on the top bar of that window, there's a little mountains icon. Select the drop-down next to it and uncheck SkyBox. That should give you a nice grey background.

Your text can't be any larger than the white border around it. You can grow that using the new Width and Height values (in the Inspector, in RectTransform.) It uses the same numbers as position – about 800 makes it across the entire screen. For Height, it's fine to increase it to have plenty of room.

Again, we can run our code with the crummy, small greyish text in the corner. Making it look nicer is just for fun.

Adding text labels wouldn't have been helpful before, since we didn't know how to find them. But now we do – using a `GameObject` pointer. All we need to know is the magic command to change the `text` part. It's `GetComponent<UnityEngine.UI.Text>().text = "cow";`. The part after the `=` is just a string.

Here's the left-to-right laps program, using an on-screen lap-counter:

```
public GameObject lapText; // drag the Text object into this
int laps = 0;

Vector3 pos;

void Start() {
  pos = new Vector3(-7,0,0);
  lapText.GetComponent<UnityEngine.UI.Text>().text="laps: 0";
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) {
    pos.x=-7;
    laps++;
    lapText.GetComponent<UnityEngine.UI.Text>().text="laps: "+laps;
  }
}
```

It's not much different from our old program – we're just slapping `laps` into the label using a pointer and the new rule for using `Text`.

26.2 More time, and APIs

Now that we know about `GetComponent<Text>().text="cow";`, we can probably change some more things. `Text` is just a class, which means it probably has more fields.

For example, there's a color picker in the Inspector under Text. If you look, you'll see dot-color works for `Text`. This turns it orange:

```
public GameObject someLabel; // drag text into here

void Start() {
  Color col = new Color(1, 0.5f, 0); // it so happens this makes orange
  someLabel.GetComponent<UnityEngine.UI.Text>().color = col;
}
```

The really neat thing about this is it's just old rules. We know classes like `Text` have fields, like `color`. And once we see `color` is just a `Color` struct we know how to use that.

Our code can change the text's alignment. Make sure the text has extra room on the sides, then click the Inspector pictures for Left/Center/Right alignment and watch the text readjust. We can also do that in the code – there's an `alignment` field which says it's a `TextAnchor`.

Looking up `TextAnchor`, it's an enumerated type. It has values like UpperLeft and LowerCenter, which pop-up when we type TextAnchor-dot.

Here's some code using it:

```
public GameObject someLabel; // drag text into here

void Start() {
  someLabel.GetComponent<UnityEngine.UI.Text>().alignment
        = TextAnchor.MiddleRight;

  // Or in two steps. Declare a TextAnchor variable:
  TextAnchor ach = TextAnchor.UpperRight;
  someLabel.GetComponent<UnityEngine.UI.Text>().alignment = ach;
}
```

The second version looks funny, declaring `TextAnchor ach;`, but it's no different than pre-computing color using `Color col;` then assigning `col`.

Ways to change built-ins through code are usually called the API – Application Programming Interface. We say something is *exposed* in the API if a program can find and change it. Text color and alignment are exposed. It's not automatic, and not everything has to be – there are a few sliders that the program has no way to change – but they try to expose as much as they think

you'll need.

I want to use color and alignment for something interesting, but first I want to show the official Unity timing trick, to really make something that delays for 1 second – not just counting out 60 ticks and hoping.

The trick is to read from the built-in clock. Most systems have one. Unity's counts in seconds, starting from when you press Play. Each time you press Play, it resets to 0, and it's always in just seconds. If you press Play and wait a minute and 10 seconds, it will say 70.

To read it, just look in the built-in global `Time.time` (you might remember this from the namespace section. It's the `time` variable in the `Time` namespace.) Here's a test program showing it change in the Inspector:

```
public float theTime; // copy of time, which we can watch

void Update() {
  theTime = Time.time;

  // this would also work, but would scroll like crazy and be hard to read:
  // print( Time.time );
}
```

The trick is that, sure, `Time.time` is just a variable, but Unity is always secretly increasing it, so it really is the time.

Here's the plan for using `Time.time` to make a delay: suppose dinner is ready is 20 minutes, and it's 6:10 now. That means dinner is ready at 6:30. That's the only number we have to remember. We can walk around, check the clock occasionally, and at 6:30 we can open the oven.

In computer terms, suppose you want a 3-second delay. Look up `Time.time` now and add 3. Save that in a global. For example, if we want to wait for 3 seconds at time 15, we save 18. Then, in Update we keep checking if the time is past 18.

Here's a very simple program which prints every 3 seconds:

```
// If this is 20, we're waiting to print until time 20:
public float nextPrintTime=0;

void Update() {
  if(Time.time>=nextPrintTime) {
    print( Random.Range(0,999) ); // cheap way to print different things
    nextPrintTime = Time.time+3.0f;
  }
}
```

The last line is resetting the time. In the old version, it was `delayCounter=60;`. Now, it's like saying "OK, I just printed at time 15, the next one is 3 seconds from now, at 18."

Here's a slightly silly program using everything at once. It counts by 1's, every 1.5 seconds, and moves the text back and forth, with a color change as it gets higher:

```
public GameObject theText; // drag Text here

float nextTime; // Time.time delay trick
public float delaySecs = 1.5f; // here so we can change it
int num=0;

void Update() {
  if(Time.time>nextTime) {
    nextTime = Time.time + delaySecs; // reset the delay
    num++;

    // show the number:
    theText.GetComponent<UnityEngine.UI.Text>().text=""+num;

    // color it as it grows:
    Color cc;
    if(num<=3) cc = new Color(1,1,1);
    else if(num<=8) cc = new Color(1,1,0);
    else cc=new Color new Color(1,0,0);
    theText.GetComponent<UnityEngine.UI.Text>().color = cc;

    // move left/right:
    TextAlign ta;
    if(num%2==0) ta = TextAlign.MiddleLeft;
    else ta = TextAlign.MiddleRight;
    theText.GetComponent<UnityEngine.UI.Text>().alignment = ta;
  }
}
```

The thing I like about this example is on one hand, it's just a cascading if and setting a few values. The most complicated thing about it is `if(n%2==0)` to check whether `n` is even. But on the other hand it's full of gibberish like UnityEngine.UI and TextAlign.

A lot of real code is like this. What's it's doing is pretty simple, but there happen to be a lot of pre-defined classes and types making it look complicated.

26.3 Three platforms

I'd like to use one script to move three platforms back-and-forth in customizable ways. Obviously, this is going to use the trick with GameObject pointers, plus we can make some fun custom classes and use a function.

I'll assume we have a front -7 to 7 view. We'll pre-make the three platforms. Just cubes will work, but we could make them nicer: make a Cube, hand-set scale to long and flat, about`(1, 0.2, 1)`, then make 2 copies. May as well name them A, B and C.

So they don't overlap, we might stack them at about `(0,2,0)`, `(0,0,0)` and `(0,-2,0)`. Our old code ignored where the Cube started – this improved code will start Cubes moving from where we placed them. The exact spots won't matter (I put my platform A on top.)

Each platform will move left and right, but will have it's own settable left/right sides, speed and a pause at each edge. For example, we want to be able to set platform A to slowly move between -7 and 3, but set platform B to move quickly from -5 and 0, pausing for 1.5 seconds before reversing.

Each platform needs the same group of variables, so this is a good place to make a class:

```
// "blueprint" type data about platforms
// including extra stuff to make it show in Inspector:
[System.Serializable]
public class PlatformStats {
  public float minX=-7, maxX=7; // left/right of how it moves
  public float moveSpd=0.05f;
  public float edgeDelaySecs=0.5f; // pause at edge
  public GameObject g; // pointer to actual platform Cube
}
```

I'm sneaking in a small `C#` rule here. A class allows you to set starting values for variables. `minX=-7` isn't a real assignment statement. But when you `new` the class, you get that value instead of 0. You can't use this trick with structs, just because.

The point of this class is we can easily create variables for all three platforms by declaring 3 of them. These are `public`, to put them in the Inspector. GUI-magic will let us pop each open and closed:

```
public PlatformStats PA, PB, PC;
```

A new thing is how our real link to the platform – `public GameObject g;` – is inside the class. We've never put a `GameObject` as a field in a class before, but there's no reason we can't. Pop them open and drag your three platforms

into the slots for `PA.g`, `PB.g` and `PC.g`.

We need the usual extra variables to run the movement: a `Vector3` for the current position, one to remember if we're moving left or right, and one to time the hit-an-edge delays. We don't need these to be in the Inspector, but still need one set per platform, so it seems fine to group them into another class:

```
// running data about platforms
class PltfrmActiveData {
  public Vector3 pos; // current position
  public float waitUntilTime=-99; // for end-pause delay
  public int mvDir=+1; // +1 = moving right; -1= moving left
}
```

Then we make one set for each platform:

```
PltfrmActiveData padA, padB, padC; // not in Inspector
```

The Inspector variables are `new`'d and filled already, but we'll have to create and set these in Start:

```
void Start () {
  // standard required new, for each:
  padA = new PltfrmActiveData();
  padB = new PltfrmActiveData();
  padC = new PltfrmActiveData();

  // start them all moving right, just because:
  padA.mvDir = padB.mvDir = padC.mvDir = +1;

  // copy starting positions from the real linked platforms:
  padA.pos = PA.g.transform.position;
  padB.pos = PB.g.transform.position;
  padC.pos = PC.g.transform.position;
}
```

The last three lines are a mouthful, but it's a good trick. `PA.g` is a link to the real platform A, which means `PA.g.transform.position` is where we hand-placed platform A before the game started. `padA.pos` is our standard variable controlling how it moves.

So those lines say to start each platform's position at the actual spot we hand-placed them. A nice bonus is our code doesn't have to decide how high up each platform is – we did that when we arranged them in Unity.

Since we have to run the same code for all three platforms, a function seems like a good idea. It needs all the data for that platform, which is now just two variables. This looks messy, but it's the same old move code, with a possible edge-delay added:

```
void movePlatform(PlatformStats P, PltfrmActiveData pa) {
  // are we still paused on an edge?
  if(Time.time<pa.waitUntilTime) return;

  pa.pos.x += P.moveSpd*pa.mvDir; // move at our speed in our direction

  if(pa.pos.x<P.minX) { // off left edge?
    pa.pos.x=P.minX; // not past edge
    pa.mvDir=+1; // start moving right
    if(P.edgeDlySecs>0) pa.waitUntilTime = Time.time+P.edgeDlySecs;
  }
  else if(pa.pos.x>P.maxX) { // off right edge?
    pa.pos.x=P.maxX; // not past edge
    pa.mvDir=-1; // start moving left
    if(P.edgeDlySecs>0) pa.waitUntilTime = Time.time+P.edgeDlySecs;
  }
  // position the actual platform:
  P.g.transform.position=pa.pos;
}
```

Notice how this counts on `pa` being a pointer. It needs to reach inside and change `pa`'s position and such.

Update merely needs to call that function for each platform:

```
void Update() {
  movePlatform(PA, padA);
  movePlatform(PB, padB);
  movePlatform(PC, padC);
}
```

26.4 More prefab use

The fun thing about prefabs is they don't feel like simple copies – using one feels like making something pop into existence. It's extra fun to do that with "physics" objects.

For this we need a falling ball from before: create a Sphere, add a Rigidbody component, possibly color it, drag it into Project to make a prefab of it, delete the original.

This script randomly fires them from the left, in a little arc. For fun, I'll shoot a few, wait, shoot a few more, in 5 waves. I'm using the new timer trick to wait:

```
public GameObject ballPF; // dragged in ball with rigidbody prefab
```

```
int waveNum=0;
float nextWave=-99; // next wave fires after this time

void Update() {
  // stops after firing 5 waves:
  if(waveNum>=5) return;

  if(Time.time<nextWave) return; // delay between each wave
  // make 2 to 5 sec delay for the next wave:
  nextWave=Time.time + Random.Range(2.0f, 5.0f);

  // make 3-5 at random heights, random speed, random size:
  Vector3 bPos; bPos.x=-7; bPos.z=0; // always on left side
  Vector3 bSpd; bSpd.z=0; // z speed is always 0
  int count = Random.Range(3,5+1); // fire 3-5 balls
  for(int i=0; i<count; i++) { // standard "this many times" loop
    GameObject bb = Instantiate(ballPF);

    bPos.y=Random.Range(-3.0f, 1.0f); // not too high
    bb.transform.position = bPos;

    float bSz = Random.Range(0.3f, 0.7f); // random size
    bb.transform.localScale = new Vector3(bSz, bSz, bSz);

    bSpd.x=Random.Range(5.0f, 12.0f); // arcing speed
    bSpd.y=Random.Range(3.0f, 8.0f); // to the right
    bb.GetComponent<Rigidbody>().velocity = bSpd;
  }

  waveNum++;
}
```

Nothing new here, but a lot at once. Some notes about the design:

- It might have looked a little nicer if the body of the `for` loop was moved into a function `fireOneBall();`. It wouldn't make the program smaller, but just breaking a long thing into parts is fine.
- Not being a function let me cheat a little with the variables. Notice how before the loop I pre-made position and speed, pre-setting the two parts that will be the same for all balls.

Instead of blasting out the balls in each wave all at once, a wave could quickly spit 4 balls one-at-a-time, wait longer, and repeat (you can really see the difference).

There's nothing new here, just being clever with variables and `if`'s:

```
int ballsThisWave=0;

void Update() {
  if(waveNum>5) return;
  if(Time.time<nextTime) return;
  ballsThisWave++; // a wave has 4 balls
  if(ballsThisWave<4) nextTime = Time.time+0.1f; // short delay (same wave)
  else { // this wave is done, reset and wait for next:
    nextTime = Time.time+Random.Range(2.0f, 5.0f);
    waveNum++;
    ballsThisWave=0; // reset
  }
  // same shoot one ball code goes here:
}
```

`ballsThisWave` is like the loop counter, sort of, except it goes up by 1 every 0.1 seconds.

An interesting-looking thing is creating objects at our position. This moves us back and forth along the bottom of the screen, and lets the space bar fire balls from us:

```
public GameObject ballPF; // prefab for the ball
Vector3 pos; // controls where we are

void Start() {
  pos = new Vector3(0,-3,0); // start us at bottom center
}

void Update() {
  if(Input.GetKey("a")) pos.x +=-0.2f; // keys move left/right
  if(Input.GetKey("s")) pos.x +=+0.2f;
  pos.x = Math.Clamp(pos.x, -7, 7);
  transform.position = pos;

  if(Input.GeyKeyDown(KeyCode.Space)) { // fire a ball
    GameObject bb = Instantiate(ballPF);

    Vector3 ballPos = pos; // <- start new ball where we are now
    ballPos.y+=1.0; // a little above us
    bb.transform.position = ballPos;

    // shoot straight up, random speed for no reason:
    Vector3 ballSpd = new Vector3(0, Random.Range(8.0f, 10.0f) ,0);
    bb.GetComponent<Rigidbody>().velocity = ballSpd;
  }
}
```

The lines moving us aren't the most efficient – if we aren't pressing a key it checks the edges and "moves" us by 0. But they're short and easy to read.

26.5 Multiple scripts

It would be fun to have the balls we shoot shrink and then wink out of existence. Way, way back I mentioned that in a typical game set-up, each object was driven by its own script. We'll do that here. Balls will have a simple script that slowly shrinks them. It would go on the original – the prefab ball in the Project panel.

The system know that prefabs are frozen, and won't run there. But each created ball gets a copy. It runs exactly like it looks:

```
float sz; // current size (x, y, z are always the same)
float shrinkStart; // what time to start shrinking

void Start() {
  // copy our real size as the starting size:
  sz = transform.localScale.x; // x, y and z are all the same
  // wait a few seconds before we shrink:
  shrinkStart = Time.time + Random.Range(3.0f, 6.0f);
}

void Update() {
  if(Time.time<shrinkStart) return;
  sz-=0.01f;
  transform.localScale = new Vector3(sz,sz,sz);
  if(sz<=0)
    Destroy(gameObject); // <- new command. Delete ourself
}
```

`Destroy` isn't a `C#` command. It's like the opposite of an Instantiate – it removes you from the Scene.

If you've shot 4 balls, there will be a copy of this on each ball, with different variables. Each ball shrinks and vanishes indepedently.

It seems like cheating, but not really. A "real" program has one main spot controlling everything. But it's common for that to be pre-written. We write our little parts and plug them in. Writing separate programs and letting Unity run them all isn't that far off from the way serious programs work.

Chapter 27

Using indexed lists

This chapter is about making and using a list of items. The chapter on using indexes with strings was a warm-up for this. Looking through a list is about the same as looking through the letters in a string.

The new thing is that each box in a list counts as a real variable. We can assign to them as well as read from them. A list is often a fast way to create a pile of similar variables all at once, giving us the ability to use them in a loop.

27.1 Intro

Every list holds one type, which goes inside a set of angle-braces. This declares a list of integers:

```
List<int> N;
```

It's our first 2-part type. You can't use just `List` by itself, you always need the second part, with a type in those brackets. The boxes in each list can be any type, but they all have to be the same.

Let's pretend `N` is [20 60 90 1]. This loop prints every box:

```
for(int i=0; i<N.Count; i++)
  print(N[i]); // 20 60 90 1
```

It's the same as a loop to print every letter in a string, except lists use `Count` for the length (instead of Length, the way strings do. Why change the name? Beats me).

We can assign to boxes, which is new. In a list, `N[0]` is a working `int` the same way as `d.age`. Boxes in lists are *L-Value*'s. These are legal:

```
// N [20 60 90 1]
N[0] = 333; // [333 60 90 1]
N[3] = -22 // [333 60 90 -22]
```

This loop changes everything in a list to 6:

```
for(int i=0; i<N.Count; i++) // basic "every box" loop
  N[i]=6;
```

We can't add to a List as easily as to a string. We can add one thing at a time, only to the back end, using the `Add` command:

```
// pretend N is [1 2 3]
N.Add(12); // now N is: [1 2 3 12]
N.Add(3); //  now N is: [1 2 3 12 3]

int cc = N.Count; // 5, N knows it has 5 things in it
```

The same as for strings, going off the edge of a list is a run-time *null reference exception* error. For examples `N[-3]` or `N[999]`. If `N` has length 8 then `N[8]` is one past the end, and also an error.

27.2 Creating Lists

Now for the details we skipped and explanations:

As mentioned, `List`'s need to have a type. `List A;` is nothing, and an error. `List<string>` is a list of strings. This should make complete sense. When we look at `N[i]` we need to know exactly what type it will be. We can't have a list where any box could be any type.

More fun list types we can declare:

```
List<string> AnimalNames; // ex: ["cow" "pig" "ant"]
List<float> F1, F2; // 2 lists of floats

List<Vector3> P; // list of points. N[0] is a point
List<Cow> Barn; // list of the Cow class we made
List<GameObject> Balls; // list of links to balls in the game
```

`<` and `>` as parenthesis are called *angle brackets*. We borrowed them from HTML (and XML and JSON). The computer uses them for types.
`GetComponent<Renderer>()` uses them that way – Renderer is the type of thing we're looking for.

`List`'s are a reference type (oh, no!) But it's OK. That only means that we're required to `new` them. There won't be any other monkey business. New lists start with nothing in them. Ex's:

```
List<int> L = new List<int>(); // all on one line is fine
AnimalNames=new List<string>(); // (declared above)
List<float> F1=new List<float(), // same as: int a=0, b=0;
```

```
            F2=new List<float>();
List<Vector3> P=new List<Vector3>(); // list of points
```

Notice how the same type goes in both places - the declare and the `new`. That was always the rule (like `Dog d = new Dog();`), but we didn't notice as much before since the types didn't have those extra `<>`'s.

We often like how lists start empty. We add what we want, stopping when we have enough. The final list is as big as it is. Here we want a list of random numbers that add to 20 or more:

```
List<int> N=new List<int>(); // size 0
int sum=0;
while(sum<20) {
  int n=Random.Range(1,10+1); // 1 to 10
  sum+=n;
  N.Add(n);
}
```

The final `N` might be [3,8,1,7,4]. The 4 put the total over 20, quitting the loop. We didn't know the final list would have 5 slots.

Other times we want the list to start at an exact size and stay there. For example, we have 6 seats and are arranging people. The list should start out with 6 `""`'s. We can make that with a quick loop:

```
// Make list with 6 empty strings
List<string> SeatingChart=new List<string>();
for(int i=0; i<6; i++)
  SeatingChart.Add("");
```

You may have noticed the funny way `Add` looks – it has a dot like a field. It's a *member* function, which we'll see later. For now, it's enough to know `N.Add(3)` grows `N`, while `Q.Add(3)` grows `Q`.

It also takes a different type depending on the list. It's sort of like an overloaded function – int-list Adds takes ints, and so on. It doesn't say it adds to the back, since that's the obvious best place. The front would cause us to change the index of everyone else's box.

`Add` also grows the list by a box and puts the new value there, which is like 2 things at once. Usually we like that – `Flowers.Add("Petunia");`. If all we want to do is add a box, we can use a dummy value like `N.Add(0)`.

An altogether list example:

```
List<string> W=new List<string>();
print(W.Count); // 0
W.Add("latin"); // [latin] ,count is 1
```

```
W.Add("greek"); // [latin, greek], count is 2
print( W[0] ); // latin
W.Add("latin"); // [latin, greek, latin], count is 3
for(int i=0;i<3;i++)
  W.Add("r"); // [latin, greek, latin, r, r, r], count is 6
```

27.3 More playing with indexes

As usual, we sometimes need variables and math for the indexes. Assume `W` has several boxes:

```
int i=0;
W[i]="duck"; // box 0
i++;
W[i]="goose"; // box 1
W[i*2]="turkey"; // box 2
```

Setting values of int-lists can look odd, with numbers in and out of the `[]`'s. For examples:

```
// N [0 10 20 30 40]
int i=2;
N[i] // 20
N[i+1] //  30 (the next box)
N[i]+1 // 21 (our box plus 1)
N[i+1]-1; // 29 (next box, minus 1)
```

It can be equally funny-looking assigning int's using some math. Real programs can have lines like these:

```
// N [0 10 20 30 40]
int i=2;
N[i]=i; // [0 10 2 30 40]
N[i+1]=i; // [0 10 20 2 40]
N[i]=i+1; // [0 10 3 30 40]
```

We can use the regular shorcuts. `N[0]+=4;` adds 4 to box 0, and `N[0]++;` adds 1. For string lists, `W[i]+="y";` adds a y to the end of that box.

A new thing we can do is a "slide". `N[k]=N[k+1];` feels like `k` is pulling the next box into it. `N[k-1]=N[k];` is like `k` pushing itself into the next lower box. Both are slide-lefts.

`N[k+1]=N[k];` feels like `k` is pushing itself to the right. `N[k]=N[k-1];` is like `k` pulling from the next lower box. They each slide right. These can all be used in loops to slide the entire list.

Swapping two boxes is fun. Recall a normal swap is `int tmp=a; a=b; b=tmp;`. An array swap is the same:

```
// swap 0 and 1:
temp=N[0]; N[0]=N[1]; N[1]=temp;

// switch positions i1 and i2:
temp=N[i1]; N[i1]=N[i2]; N[i1]=temp;
```

27.4 List loop examples

27.4.1 list-Initializing loops

If we want a pre-made list with certain values, there are a few tricks.

This is a repeat of when we want 10 slots, and will fill them later. We'll `Add` ten empty-strings:

```
List<string> W = new List<string>(); // size 0
for(int i=0; i<10; i++) W.Add(""); // size 10, all ""'s
```

With int lists, it's semi-common to start each box with its index. To do that, use `i` inside the `Add`:

```
List<int> NN = new List<int>();
for(int i=0; i<10; i++) NN.Add(i); // [0 1 2 3 4 5 6 7 8 9]
```

If we wanted it to be 1,2,3 instead, we'd use `NN.Add(i+1);`. For 10, 20, 30 it would be `NN.Add((i+1)*10);`

If our list starts with a more interesting sequence, we can have a loop hit the exact numbers, `Add`'ing. This makes a list with 20, 18, 16 ...2. It will be as long as it needs to be for them to fit:

```
List<int> Nums=new List<int>();

// loops hits exactly 20 18 16 ... 2:
for(int i=20; i>=2; i-=2) Nums.Add(i);
```

Sometimes it's easier to create the list first, then use loops to set the values. Here evens are "goat" and odds are "cow". One loop handles each:

```
List<string> Ani=new List<string>();
for(int i=0;i<20;i++) Ani.Add(""); // "empty" size 20 list
// set evens to goat, odds to cow:
for(int i=0; i<Ani.Count; i+=2) Ani[i]="goat"; // even loop
for(int i=1; i<Ani.Count; i+=2) Ani[i]="cow"; // odd loop
```

Sometimes, for testing, it's nice to fill a list with random numbers. This loop puts 1 to 100 in each slot:

```
List<int> R=new List<int>(); // list of random #'s
for(int i=0; i<20; i++) R.Add(Random.Range(1,101));
```

Character lists aren't very common. Strings are better, usually. But char lists allow us to easily change each character. You might use one for a Hangman game starting with all underscores:

```
List<char> Letters = new List<char>();
for(int i=0; i<12; i++) Letters.Add('_'); // twelve underscores
```

Keeping to the hangman theme, we could use a list of 26 bools to remember which letters were guessed. There's nothing special – each slot is `true` or `false`, but it looks odd:

```
List<bool> LetterWasUsed=new List<bool>();
for(int i=0; i<26; i++) LetterWasUsed.Add(false);
```

Often we simply hand-`Add` each item:

```
List<string> ColorWords=new List<string>();
ColorWords.Add("red");
ColorWords.Add("green");
ColorWords.Add("violet"); // [red, green, violet], so far
```

In Unity we can cheat. `public List<string> ColorsWords;` appears in the Inspector. We can pop it open, type a size (it starts at 0), and hand-enter the values. When our program starts, the entire list is magically created.

An older use that you don't see as much is a reading loop. Pretend `readLine()` reads one line from a file, with `""` for when it's past the end. We can write a "go until done" while loop with an `Add` as the body:

```
string nextCol=readLine();
while(nextCol.Length>0) {
  ColorWords.Add(nextCol);
  nextCol=readLine(); // at end, so we stop and don't add ""
}
```

Most systems don't read this anymore. But it's a fun loop, regardless. `readLine` is driving it, and you can check it works for a 0-line file, and Adds the last line, but not the `""` past-the-end line.

27.4.2 Printing

Like structs, there's no built-in command to print an entire list. You usually use an index loop to look at each part. This turns an int list into a string with commas and parens. To get the commas correct, I'm going to do the first one by hand, then have the loop start at 1:

```
string w = "("+N[0]; // add 1st item
for(int i=1; i<N.Count; i++) // loop for everything else
  w+= ", "+N[i];
w+= ")";
print(w); // Ex: (0, 3, 0, 0)
```

A `bool` list is fun to print, since we can shorten true and false to T and F. I'm going to leave out the commas:

```
string w = "(";
for(int i=0; i<LetterWasUsed.Count; i++) {
  string tf;
  if(LetterWasUsed[i]) tf="T";
  else tf="f"; // lowercase f stands out from T better
  w+= tf;
}
w+= ")";
print(w); // Ex: (fffTffTT)
```

It's semi-common to dress up `float`s a little bit when printing a list. A list full of 8.333333333's is going to overflow the screen. Unity sometimes rounds them to tenths, for printing. We can do that (I'm using a different trick for the commas):

```
string w; w = "(";
for(int i=0; i<N.Count; i++) {
  if(i!=0) w+=", "; // all but first gets a comma in front
  float nn = ((int)(N[i]*10))/10.0f; // round to 0.1's
  w=w+nn;
}
w+= ")";
print(w); // ex: (5, 4.1, 8.6, -3.2)
```

This would show numbers like 0.003 as just 0. That's usually fine. If your list is full of very small numbers, you wouldn't print it this way.

27.4.3 List Searching

A basic list search is the same as a string search, like counting how many `'e'`s. But, with numbers, we can search for different sorts of things.

This warm-up loop counts how many 7's there are in an `int` list. Nothing special about it:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  if(N[i]==7) count++;
}
```

We can count how many positive numbers there are. This is still pretty much the same, but it *feels* like it should have been harder to do:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  if(N[i]>=1) count++;
}
```

We can do all the math we need inside the loop. This one counts how many are 11, 22, 33 ... 99. If you hate the math, the important part is `N[i]` is the current int, and we can compute lots of stuff for ints:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  int nn = N[i]; // so I don't have to keep typing N[i]
  bool inRange = nn>=11 && nn<=99;
  int d1=n%10; // 1st digit
  int d2=(n/10)%10; // 2nd digit
  if(inRange && d1==d2) count++;
}
```

This counts how many strings in a list are 6 letters or longer:

```
int longWords=0;
for(int i=0; i<W.Count; i++) {
  if(W[i].Length>=6) longWords++;
}
```

`W[i].Length` is 2 parts. `W[i]` is a string. Maybe it's `narwhale"`. So `W[i].Length` is the number of letters in narwhale. To compare, `W[i].Count` would be an error. Strings don't have a Count.

Finding the largest number is a clever rewrite of the Price-is-Right trick. Without a list, I did something like `biggest=a; if(b>biggest) biggest=b; if(c>biggest) biggest=c;` and so on. One `if` for each number.

The beauty of a list is we can do the same thing, but using a loop to run every `if`:

```
int biggest = N[0]; // 1st is biggest, so far
for(int i=1; i<N.Count; i++) { // check the rest
  if(N[i]>biggest) biggest = N[i];
}
```

In my mind, the loop says "now give every box past 0 a chance to win." If we "unroll" the loop and write all the lines, it's just `if(N[1]>biggest) biggest=N[1];`, then again for every number after that. Lists are pretty cool.

Here's the same idea, but finding the shortest string in a list. It compares the lengths of the strings. I added test prints for fun:

```
string shortest= W[0];
// print("start: "+shortest); // testing
for(int i=1; i<W.Count; i++) {
  string ww = W[i]; // save to avoid typing W[i] twice
  if(ww.Length<shortest.Length) {
    shortest=ww;
    //print("new shortest: "+shortest); // testing
  }
}
```

If we ran this on `["aardvark", "cow", "horsey", "ox", "bear"]` we'd see it print aardvark, then cow, then ox, and the answer is ox.

27.5 Random item

My random town example used a big `if-else` to pick town names. A list can make picking a random word easier. Put all the words in a list, then pick a random index and use the word there.

This picks one word at random from any list:

```
List<string> Ani = new List<string>();
Ani.Add("frog"); Ani.Add("toad"); // give it some sample animals
Ani.Add("crawdad"); Ani.Add("polywog");

int ai = Random.Range(0, Ani.Count); // 0 to length-1, Perfect!
string aName = Ani[ai];
print("random word is " + aName); // ex: "crawdad"
```

With four words, the indexes are 0 to 3, which is exactly what `Random.Range(0, Ani.Count)` can roll. This is the real reason random int works that way – back in the day everyone knew "random 4" meant 0,1,2, or 3.

With randomness it's hard to notice code that's off-by-one. For example, `Random.Range(0, Ani.Count+1)` could roll past the end and crash. But you might test it 400 times, never have that happen at random, and think it works. It's also hard to notice if there's 1 animal you can never get.

A hacky trick to pick one word more often is to repeat it in the list. If we add an extra `Ani.Add("toad");` then we have 2 of them. We'll roll a toad twice as often.

27.5.1 A list as a sequence

In the random word example, the list is just a bunch of stuff, and the order doesn't matter. We also like to use lists to make a sequence: use item 0, then a little later use item 1, and so on.

For example, suppose I want the space bar to print words one at a time, from a list.

The trick is to think of it like a slow loop. We'll have an index starting at 0, add 1 each time, and when it hits `Count` we've gone off the edge. Here's how it looks:

```
public List<string> W;
// sample Inspector values: ["eat", "at", "Joes", "food", "gas"];
public int phraseIndex = 0; // index variable for W

//public GameObject theLabel; // optional for on-screen Text

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    string oneWord = W[phraseIndex];
    print(oneWord);
    phraseIndex++;
    if(phraseIndex>=W.Count) phraseIndex=0; // may as well wrap-around

    //theLabel.GetComponent<UnityEngine.UI.Text>().text=oneWord;
  }
}
```

Here's a completely different-seeming program using the same idea. I want a Cube to pop to preset spots, once a second. The preset locations will be stored in a slow-walked list.

For examples, [-7, 7, -5, 5, -3, 3] would make it dance around the center, closing in. But using [0, 2, 1, 3, 2, 4, 3, 5, 4, 6] makes it stagger-walk to the right. The code:

```
public List<float> PosX; // set size and enter x-values in Inspector
public int pi = 0; // index of next position in PosX
int delayCounter=0;

void Update() {
  delayCounter--; // the "wait 50 ticks" delay trick
  if(delayCount<0) { // make 1 step:
    delayCount=50;

    float xx = PosX[pi]; // get next position
    pi++;
    if(pi>=PosX.Count) { // check for wrap-around
      pi=0;
      delayCount+=50; // little extra delay on wrap looks nice
    }
```

```
    transform.position = new Vector(xx, 0, 0); // go to next position
  }
}
```

This is just the movement code and the slow-walk-list code crammed together.

The trick can work for anything. Here's a version using a list to control the delay. To keep it simple, this Cube pops left-to-right by 0.5 each tick. The code:

```
public List<int> DelayAmt;
// ex: [10, 10, 100, 100] goes fast, fast, slow, slow
public int di; // index into DelayAmt
int delay; // ticks remaining until next pop

Vector3 pos;

void Start() {
  // start us on left side, with the first delay:
  pos.x=-7; pos.y=0; pos.z=0;
  transform.position = pos;
  // set-up wait for first delay:
  delay=DelayAmt[0];
  di=1; // this will be the next delay
}

void Update() {
  delay--;
  if(delay<0) {
    delay=DelayAmt[di]; // read next delay from the sequence
    di++; if(di>=DelayAmt.Count) di=0;

    // move a little right, with wrap-around. Nothing special:
    pos.x+=0.5f; if(pos.x>7) { pos.x=-7; }
    transform.position = pos;
  }
}
```

It's kind of fun to try different numbers. Changing to `[10, 10, 10, 100]` would have it scramble 3 quick steps, then a pause. `[80, 60, 40, 20, 10, 5]` would have it appear to gather momentum.

This is another one of those programs that took a lot more work that it looks. It took me a few tries to make it start waiting using the first delay, and more work to have it not double-use the first delay.

27.5.2 Variable variables

Sometimes our program needs a pile of nearly identical ints. Using one list to make them all is easier than declaring them individually, and can make the program much simpler through the magic of indexing (seriously, this is a great trick).

In this example, I want to roll a 6-sided die 50 times and count how many of each number. We need six int's to hold the results, which we can get with a size-6 list. In our minds. `Count[0]` is how many 1's we rolled, up to `Count[5]` is how many 6's. The code:

```
List<int> Count = new List<int>();
for(int i=0;i<6;i++) Count.Add(0); // now we have 6 zeroed-out numbers

// roll 50 dice and count:
for(int i=0; i<50; i++) {
  int roll = Random.Range(1, 7);
  Count[roll-1]++; // <- the very sneaky line
}
```

The magic part is `Count[roll-1]++;`. In our minds, `Count` holds six regular variables. `Count[roll-1]` is using `roll` to look up which one. If `roll` is 1, that gives us `Count[0]`, which holds how many 1's we've gotten. If we roll a 6, the whole line says `Count[5]++;`, which is adding one to the 6's box.

It's a really clever trick to make a "variable variable." Without a list, if we had 6 individual variables, we'd need a 6-part `if`: `if(roll==1) count1++; else if(roll==2) count2++;` and so on.

Here's the same trick for saving high scores. Pretend we have 10 levels in a game, with the high scores stored in a list. `HighScore[0]` is the high score for the first level, and so on.

The player can jump around between levels somehow. Code like this would look up the high score for the level they're leaving, and update it if needed:

```
List<int> HighScores = new List<int>();
for(int i=0;i<10;i++) HighScores.Add(0); // game has 10 levels
int levelNow; // moves from 0-9 as we change levels

  // check current score as a possible new high score:
  int oldHS = HighScores[levelNow]; // <- like a 10-way if
  if(score>oldHS) {
    HighScores[levelNow]=score; // <- new high score
    print("new high score"); // testing
  }
```

It's another case where `HighScores[levelNow]` saves us a ten-part `if`.

27.5.3 Moving parts of a list

There are some general tricks involving moving around the boxes in a list. These pop up in a few places, and are good index practice.

A useful one is a shuffle. The final list will have the same things in it, but mixed up. The simplest way to do this is using one loop. Go through each item, and switch it with some random position:

```
for(int i=0; i<N.Count; i++) {
  int newPos = Random.Range(0, N.Count); // random index
  int temp = N[i]; N[i]=N[newPos]; N[newPos]=temp; // swap
}
```

One funny thing – we might switch with ourself. `newPos` could be equal to `i`. That won't break anything, and is actually good. A real shuffle has a small chance of keeping each thing in the same place, and so does ours.

One use of a shuffle is randomly rolling 1-6 with no repeats. First create a list of 1 to 6, shuffle it, then slow-walk as you need the numbers:

Partial code:

```
List<int> RandNums = new List<int>();
for(int i=0;i<6;i++) RandNums.Add(i+1); // 1-6
int rnIndex=0; // used to slow-walk through RandNums
// shuffle goes here

int getNextRandNum() {
  if(rnIndex>=6) return -1; // too many
  int answer = RandNums[rnIndex];
  rnIndex++;
  return answer;
}
```

It's reverse thinking. It feels like we should do the random part as we request each new number. But we can pre-load the randomness. In other words, it's like shuffling a deck of cards, then getting random cards by dealing from the top.

Some fun exercises, which you don't use much for real, are rotating a list – moving every variable one space left or right, and wrapping around. Here's a picture of both types of shifts:

```
0  1  2  3  4  5
10 20 30 40 50 60  // starting list

shift left:
20 30 40 50 60 10
```

```
shift right:
60 10 20 30 40 50
```

The main problems are the wrap-around and not erasing anything. To rotate left we save `N[0]`, then shift everything from `N[1]` onward to the next lower box, then copy the saved `N[0]` to the right side:

```
int first = N[0];
for(int i=1; i<N.Count; i++)
  N[i-1]=N[i]; // push left
N[N.Count-1] = first; // copy saved first to end
```

As a check, with `i` starting at 1, the first slide is `N[0]=N[1];`, which is correct.

For practice, we could also write that using a pull-left: `N[i]=N[i+1];`. We'd start at 0 and go to 1 less than the end:

```
// different way to rotate left:
int first = N[0];
for(int i=0; i<N.Count-1; i++)
  N[i]=N[i+1]; // pull next higher value into me
N[N.Count-1] = first;
```

Same check: since `i` now starts at 0, the first is `N[0]=N[0+1];` – still correct.

Rotating a list to the right is also good for index practice. We save the last item, then go right to left, sliding items forward one space (for fun, try going left to right instead, it just copies `N[0]` into every box):

```
int last = N[ N.Count-1 ];
for(int i=N.Count-1; i>=1; i--) // right to left, stopping early
  N[i]=N[i-1]; // pull next lower into me
N[0] = last;
```

Another quick check: the last slide is at `i=1`, so is `N[1]=N[1-1]`, which is correct.

27.6 Two list tricks

There are some fun things we can do using two lists.

With lists, copying `List<int> B=A;` doesn't work at all. It just makes `B` point to `A`, with both sharing the same real list. Instead you need to copy the list, a box at a time:

```
// make B a copy of A. Copy from A into B 1 at a time:
List<int> B=new List<int>();
for(int i=0;i<A.Count;i++) B.Add(A[i]);
```

Combining two lists end to end uses the same idea. Start with a fresh one, copying `A` then `B` into it:

```
List<int> A, B; // pretend these are both created and filled in

List<int> C = newList<int>();
for(int i=0; i<A.Count;i++) C.Add(A[i]);
for(int i=0; i<B.Count; i++) C.Add(B[i]);
```

The `Add` in the second loop always fools me. It's running 0,1,2 ... through `B`. But those numbers are only for `B`. The `Add` puts then on the end of `C`, at whatever box that happens to be.

27.7 Size 0 and 1 lists

Usually a size-0 list is temporary until we add our items. But sometimes the final list is size 0. That's fine. If `D` is a list of the dogs you hate, and you like all dogs, it's size-0. A loop over a size-0 list won't cause an error – it quits right away.

In a size 0 list, `D[0]` is an error. You can't look up any boxes, because there aren't any. That's usually not a problem since a loop won't try to check it.

Having a final list be size 1 also seems like a waste. But it's often what we want. You only hate 1 dog, but you could have hated a lot more.

Even though `D[0]` is the only box, you still have to write it out. `D="Spike";` is an error. The computer won't figure out that `D[0]` is the only place it could go.

27.8 Functions with lists

List inputs and outputs to functions don't have any new rules, but they're nice to see.

27.8.1 Lists as inputs

A list can be an input to a function. You just put the type, like `List<int>` or `List<string>`.

This checks whether a string list contains a certain word:

```
bool hasWord(List<string> W, string findMe) {
  for(int i=0; i<W.Count; i++)
    if(W[i] == findMe) return true;
```

```
  return false;
}
```

You pass a list the usual way, with just the name:

```
List<string> Cows; // ex: ["bessy", "moo-moo", "Lou"];

if( hasWord(Cows, "Lou") ) ...
```

Double equals, `==`, won't properly compare lists, since they're pointers. As usual, `if(A==B)` checks whether both point to the same list, which is not useful. To check whether two different lists are the same, we need to hand-walk through both, comparing pairs at each index:

```
bool equals(List<int> A, List<int> B) {
  if(A.Count != B.Count) return false; // not same size is not-equal
  for(int i=0; i<A.Count; i++) {
    if(A[i]!=B[i]) return false;
  }
  return true;
}
```

Same as before, you run it with just the names: `if(equals(N1,N2))`.

A fun one is checking whether a list has all numbers in increasing order, like [3, 8, 25, 26, 30]. It works like the double-letters example for strings. Every number compares to the one before it, and needs to be larger:

```
bool isIncreasing(List<int> N) {
  for(int i=1; i<N.Count; i++) // start at 2nd item
    if(N[i-1]>=N[i]) return false; // greater than the one before me?
  return true;
}
```

The early return true/false logic can be tricky. If just one pair isn't going up, the overall answer is false. We may as well quit and say that. But we can't say the entire list is increasing until we're checked them all.

Since lists are pointers, changing a list in a function is changing the real list. In other words, you can write functions whose purpose is to change a list. This changes negative numbers in a list to zero:

```
void fixNegatives(List<int> N) {
  for(int i=0; i<N.Count; i++)
    if(N[i]<0) N[i]=0;
}
```

We'd call it like `fixNegatives(N1);`.

27.8.2 Returning lists

We can return lists from functions. Usually the function creates the list and returns a pointer to it.

This function makes a list full of 0's of whatever size you want. Notice how the return type is `List<int>`:

```
List<int> zeroList(int sz) {
  List<int> A = new List<int>();
  for(int i=0; i<sz; i++) A.Add(0);
  return A;
}
```

List inputs and outputs together are common. Here's a basic list clone. It's merely the list-copy code from above, in a function:

```
List<string> getListCopy(List<string> A) {
  List<string> B=new List<string>();
  for(int i=0;i<A.Count;i++) B.Add(A);
  return B;
}
```

`List<string> W2 = getListCopy(W1);` runs it.

We can put the combine end-to-end code in a function. I think it's clear enough this takes two lists and returns another:

```
List<string> combineEtoE(List<string> A, List<string> B) {
  List<string> C = new List<string>(); // same code as above
  for(int i=0;i<A.Count;i++) C.Add(A[i]);
  for(int i=0;i<B.Count;i++) C.Add(B[i]);
  return C;
}
```

Another exercise which is fun, but not all that common, is pair-wise adding two lists (pair-wise means using matching positions, like `A[0]+B[0]`). It looks like this:

```
  0 1 2 3 4 5 <- indexes
A 3 2 0 4 1 1
B 1 1 1 2 3
  ------------
  4 3 1 6 4 <- pairwise sums
```

If they aren't the same length the options are to stop when the shorter runs out, or to pretend the rest are 0's. I did it the first way, since it's simpler. Here's a function returning a pair-wise list add:

```
List<int> pairwiseAdd(List<int> A, List<int> B) {
  // length of shortest:
  int len=A.Count;
  if(B.Count<len) len=B.Count;

  List<int> C = new List<int>();
  for(int i=0;i<len;i++) C.Add(A[i]+B[i]); // <- math in the Add
  return C;
}
```

Returning a list computed form the input list is always fun. This returns only the even numbers. It's pretty much the same as removing all `z`'s from a string:

```
List<int> evensOnly(List<int> N) {
  List<int> Result=new List<int>();
  for(int i=0;i<N.Count;i++)
    if(N[i]%2==0) Result.Add(N[i]);
  return Result;
}
```

Notice this could return a length-0 list, if everything in the input is odd, and that's fine.

27.9 errors

Lists give you the usual pointer errors. Using one that hasn't been `new`'d will give run-time error *null reference exception*:

```
List<int> A; // as a global, this is set to null

A[0]=4; // null reference exception
print( A.Count ); // null reference exception
```

We'll get all the usual list-index-out-of-bounds errors if we go past the end. And the last index is still `N.Count-1`.

A common off-end is assuming two lists are the same length, forgetting to compare sizes. This throws a null-ref error if `B` is shorter than `A`:

```
for(int i=0; i<A.Count; i++) A[i]=B[i];
// out-of-range error if B is shorter than A
```

Another semi-common one is not checking for size 0 (or sometimes even 1.) This function returns the smallest item in the list. It crashes if the input is size 0:

```
int indexOfSmallest(int[] N) {
  int ans=N[0]; // <-- oops. forgot to check this exists
  for(int i=1;i<N.Count;i++) if(N[i]<ans) ans=N[i];
  return i;
}
```

The first line should be: `if(N.Count==0) return -1;`.

Another reminder: since these are pointers we get the same pointer non-error errors as with classes: `A=B;` and `if(A==b)` work in the funny pointer way.

`List` lives in the namespace `System.Collections.Generic`. Pre-made Unity files have that line on the top. If you write your own from scratch, `List` by itself will give "not found" errors.

You can write out `System.Collections.Generic.List<float> F;`. Or, easier, copy that `using` line to the top of your file.

27.10 Assign shortcut

For testing, it's a pain to make sample lists with lots of `Add`'s. There's a way to make them all at once, which is very ugly, but still shorter:

```
List<int> A = new List<int>(new int[]{5,8,2,12});
List<string> Animals = new List<string>()(new string[]{"cow","hen","pig"});
```

It makes a normal list, so we could still use `Animals.Add("worm");` later, if we somehow needed to.

If you were wondering, there are some languages where you can just write `N=(3,5,8);` and have a list. But those languages have their own places you have to work extra hard to do something that seems like it should be easy.

27.11 List variables as pointers

We're generally happy pretending that lists are normal variables. We might take advantage of their pointerness to change the contents inside of functions, but that's about it. Even so, we can do real pointer things with lists.

Here `L1` and `L2` count are considered regular lists, while `Lp` acts as a pointer:

```
public List<string> L1=new List<string>(), L2=new List<string();

void Start() {
  List<string> Lp; // pointer
  Lp=L1; // alternate way to get to L1
  Lp.Add("cherry"); // adding to L1
```

```
  Lp=L2; // alternate way to get to L2
  Lp.Add("emerald"); // adding to L2
}
```

We can use this trick to choose between 2 lists in the slow-word display:

```
public List<string> W1; // ["eat", "at", "Joes", "get", "gas"]
public List<string> W2; // ['loose", "lips", "sink", "ships"]
List<string> CurWords = null; // a pointer to W1 or W2

void Start() {
  // randomly aim at W1 or W2:
  if(Random.Range(0,1+1)==0) CurWords=W1;
  else CurWords=W2;
}
```

The rest of the code can use `CurWords` like it was a normal list, even though it's always `W1` or `W2`.

We can write a function returning a pointer to whichever list has more 7's:

```
List<int> getMore7List(List<int> A, List<int> B) {
  int s0=sevenCount(A); // pretend this function exists
  int s1=sevenCount(B);
  if(s0>s1) return A;
  return B;
}
```

The return value isn't a new list – it's a pointer to one of the lists we gave it:

```
List<int> C=getMore7List(N1, N2);
C.Add(99); // adding to end of either N1 or N2
```

Chapter 28

Nested loops

Putting a loop in a loop (a *nested loop*) doesn't have new rules, but works in surprising ways, and there are some common tricks.

This chapter is mostly various nested loop examples, But I added two new loop rules: `break;` and `continue;`. They are odd, but commonly used and make writing loops easier.

The main trick with one loop inside the other is that the inner loop restarts each time. This nested loop below runs 48 times, printing every combination from (0,0) to (7,5):

```
// demo nested loop
for(int i=0; i<8; i++) {
  for(int j=0; j<6; j++) {
    print( "i="+i+" j="+j );
  }
}
```

If prints (0,0) then (0,1) up to (0,5), then restarts with (1,0) to (1,5), ending with (7,5)

`j` seems like a bad choice for the inside loop variable, since it looks so much like an `i`. But it's traditional. If you have good names for the loop variables, use them. If not, use `i`, and then `j` for an inside loop.

Sometimes it helps to rewrite as a simpler loop. Here's the middle loop as a while, where we can clearly see `j` resets to 0 each time:

```
// inner while loop:
for(int i=0; i<8; i++) {
  int j=0;
  while(j<6) {
    print( "i="+i+" j="+j );
```

```
    j++;
  }
}
```

28.1 As a grid

Those numbers look like grid coordinates, and that's a common way nested loops are used: rows and columns. Here's a program using the nested loops from above to make an 8 by 6 grid (it uses `Instantiate` and assumes you have a 1x1x1 Cube being copied, probably a prefab).

```
public GameObject blockPF; // link to a 1x1x1 Cube (or anything, really)

void Start() {
  Vector3 blockPos; blockPos.z=0;

  for(int col=0; col<8; col++) {
    for(int row=0; row<6; row++) {
      GameObject bb = Instantiate(blockPF);
      blockPos.x = -6 + 1.2f*col;
      blockPos.y = -3 + 1.2f*row;
      bb.transform.position = blockPos;
    }
  }
}
```

We already know that the inside runs 48 times, so this Instantiates 48 blocks.

In our minds, the loop's job is to make `row` and `col` be every combination from (0,0) (7,5). The body of the loop is really "make a block, positioned where `row` and `col` say."

The exact positioning numbers I used aren't important. Block (0,0) is at position (-6,-3), which is near the bottom-left corner of our screen. I spaced them `1.2f` apart, so we can see the gaps.

A fun thing for testing is to color or somehow change some of them, to see it better. Putting this in the loop makes the (0,0) Cube yellow, turns row 0 red, and column 0 green:

```
  // color some blocks, to get a feel:
  color cc = new Color(1,1,1); // white (if nothing else)
  if(row==0 && col==0) cc = new Color(1,1,0); // 00=yellow
  else if(row==0) cc = new Color(1,0,0); // row0=red
  else if(col==0) cc = new Color(0,1,0); // col0=green
  bb.GetComponent<Renderer>().material.color = cc;
```

We should see yellow in the lower-left corner, red along the bottom and green along the left side. So row 0 is the bottom, column 0 is the left.

I color things since it's easy to get positions mixed up. For example, this change, to the `blockPos.y` line, makes the same arrangement, but puts row 0 at the top:

```
// 8 wide, (0,0) upper-left
blockPos.y = 2 - 1.2f*row; // row 0 at 2.0, going down
```

Or we can flip how row&col change x&y. This would make it 6 wide and 8 tall (instead of 8 wide and 6 tall.) It's confusing, since the "rows" now run up and down:

```
// 6 wide, (0,0) lower-left:
blockPos.x = -6 + 1.2f*row; // row is x
blockPos.y = -3 + 1.2f*col; // column is y
```

These examples seem a little silly, but they point out what the nested loop does and doesn't do. It makes all the combinations of (0,0) (0,1) .... If the outer loop counts to 10 and the inner counts to 14, you get 140 items in a 10x14 pattern. But how you orient and position them, if at all, is up to you.

28.2 Pattern in pattern

A grid is a good picture of a loop – it's easy to imagine a loop touching each row, then the second loop making everything in that row. But a nested loop can be any pattern in a pattern.

Suppose we need 11 12 13, and then 21 22 23, then 31 32 33 ... up to 93. That's a pattern in a pattern (count by 10's, count 1,2,3 for each.) We can make it with a nested loop:

```
// loops to count 11 12 13,  21 22 23 ... :
for(int i=10; i<100; i+=10) { // 10, 20 ... 90
  for(int j=1; j<=3; j++) { // 1, 2, 3
    int num=i+j;
    print( num );
  }
}
```

It seems like `tens` might be a better name for the first loop variable, then `one`s for the second. I kept it as `i` and `j` since the pattern doesn't have to be 10's and 1's.

For this next one, I'd like a line of blocks repeating small, medium, large over and over, like a sawblade. I can do that with a nested loop – the inner makes the three sizes, the outer makes several groups:

```
// repeating small, medium, large blocks:
Vector3 pos = new Vector3(-6, 0, 0); // left edge of next block
for(int group=0; group<4; group++) {
  for(int sz=0; sz<3; sz++) { // 3 different sized blocks
    float bSz = 0.3f+sz*0.4; // equation for the actual size
    GameObject gg = Instantiate(blockPF);
    gg.transform.localScale = new Vector3(bSz, bSz, 1);

    pos.x += bSz/2; // move over 1/2 a block, to center
    gg.transform.position = pos;
    pos.x += bSz/2; // now move other half, to right edge
  }
}
```

Placing them side-by-side with changing sizes is tricker than it looks. Instead of trying to write a formula, I used a running total. `pos.x` starts on the left, and is increased it by the size of the current block (1/2 over, then place, then the other 1/2 over is to account for block placement using the center).

This next one is a variation of that, but with colors. We'll make a list of colors, with help from the Inspector, any amount. Then we'll make 2-4 balls of each color, in random positions. That's a nested loop: the outer one hits each color, the inner one makes 2-4 of that color:

```
List<Color> Col; // sized and filled in Inspector
public GameObject blockPF; // drag in some prefab

void Start() {
  Vector3 pos; pos.z=0;
  for(int ci=0;  ci<Col.Count; ci++) { // each color
    int count = Random.Range(2,4+1);
    for(int j=0; j<count; j++) { // how many of each
      GameObject gg = Instantiate(blockPF);

      gg.GetComponent<Renderer>().material.color = Col[ci];

      pos.x=Random.Range(-6.5f, 6.5f);
      pos.y=Random.Range(-3.5f, 3.0f);
      gg.transform.position = pos;
  }
}
```

I like to double-check the indexes: the inside uses `Col[ci]` to pick the color. The `ci` loop goes from 0 to `<Col.Count`. So that gets them all, and doesn't go off the edge.

With those last two I wanted to show how a nested loop feels like a grid, but doesn't have to be. The sizes one was a straight line, where-as this is just random.

The important thing about using a nested loop to make things is the description. "6 rows, with 8 each" is a nested description, so is "2-4 balls of each color."

28.3 Wedge exercises

An old trick to get practice with nested loops and indexes is drawing various triangles – really, grids with some of the boxes left out. Instead of making 7 in each row, we can change it up. For example, a triangular grid, where rows have 1, then 2, then 3 then 4:

```
0000
000
00
0
```

To keep things simple, I'm going to keep (0,0) on the lower left, with `i` going up, and `j` going right.

To simplify the loop code, and to show off functions and pointers more, I'll move creating and placing the blocks into a function. The pos x/y formulas are unchanged:

```
GameObject newBlockAt(int i, int j) {
  GameObject g = Instantiate(blockPF);
  Vector3 pos; pos.z=0; //
  pos.x = -6 + 1.2f*j; // +j goes right
  pos.y = -3 + 1.2f*i; // +i goes up
  g.transform.position = pos;
  return g;
}
```

This is mostly a wrapper around `Instantiate`. It returns a pointer to the block it made, just in case we want to make more changes. A normal use would be `GameObject blk=newBlockAt(0,0);`. That makes a block in the lower-left 00 position. We're allowed to ignore the return type. If we don't need to make any more changes, `newBlockAt(0,0);` by itself is fine.

An obvious triangle is the picture from above: 1 block in the first/bottom row, left side, then 2 in the row above that, and so on. Each row is as long as the row # plus 1:

```
// upper-left triangle loop:
for(int i=0; i<4; i++) { // 4 rows
```

```
  int numInRow = i+1; // will be 1,2,3,4
  for(int j=0; j<numInRow; j++) {
    newBlockAt(i,j);
  }
}
```

Before, the inner-loop always ran 8 times. Now, it runs based on an equation.

A shorter, more common version leaves out the extra `numInRow` variable. The equation for the row length goes directly into the inner loop test:

```
for(int i=0; i<4; i++)
  for(int j=0; j<i+1; j++) newBlockAt(i,j);
```

This is a mess to read, at first. The secret is that `i` isn't changing as the inner loops runs. When `i` is 2, `j<i+1` is locked at `j<3`. Clearly, this is very prone to infinite loops if the `i`'s and `j`'s are scrambled.

A variant of that triangle is putting the blocks on the right side, making a lower-right triangle. Before, each row started at 0 and went a different amount. For this one, each row starts in a different spot, and always goes to the end:

```
    0
   00
  000
 0000
00000
// lower-right 5x5 triangle loops:
for(int i=0; i<5; i++) {
  int rowStart = i; // each row starts 1 further in
  for(int j=rowStart; j<5; j++) {
    newBlockAt(i,j);
  }
}
```

The first row runs all the way from 0-4, the one above it 1-4, 2-4, 3-4 then the top is just 4.

If we changed the `rowStart` line to `rowStart=4-i`, we'd have an upper-right triangle. The bottom row would be 4-4, then 3-4, 2-4, 1-4 and 0-4 for the top row.

When I need to make something like this, I have to draw a numbered grid, mark each row and check the start/end numbers, guess a formula, run it, tweak it . . . .

We can use both tricks to make a pyramid. The first row will go all the way across, but then each row with start at one more, and end at one less. This makes rows of length 9, 7, 5, 3, 1:

```
    0
   000
  00000
 0000000
000000000
// pyramid:
for(int i=0; i<5; i++) { // there are 5 rows
  int rowStart = i, rowEnd=8-i;
  for(int j=rowStart; j<=rowEnd; j++) {
    newBlockAt(i,j);
  }
}
```

The inner loop could have been `for(int j=i; j<=8-i; j++)`, but I think the extra `rowStart` and `rowEnd` make it easier to read.

For more fun, here's an upside-down 1, 3, 5, 7, 9, 11 pyramid:

```
// pyramid:
for(int i=0; i<6; i++) { // there are 6 rows
  int rowStart = 5-i, rowEnd=5+i;
  for(int j=rowStart; j<=rowEnd; j++) {
    newBlockAt(i,j);
  }
}
```

A quick sanity-check: the bottom row, when `i` is 0, goes from 5-0 to 5+0, which is just one thing, in slot 5. When `i` gets to the last row, on top, it's 5. `rowStart` is `5-i = 0` and `rowEnd` is `5+i = 10`. So the top row is 0 to 10, which is 11 across, which is correct.

A checkerbox is every other square. It's not a wedge, but it seems like it belongs in this section. A really slick way is to go through every square and skip the ones where row+column is an even number:

```
// checkerbox:
for(int i=0; i<6; i++) {
  for(int j=0; j<8; j++) {
    bool useMe = (i+j)%2==0;
    if(useMe) newBlockAt(i,j);
  }
}
```

Another try at a checkerboard is to have the loop making the rows go by 2's, starting on 1 for even rows and 0 for odd rows. This isn't as nice, but it works:

```
// checkerbox:
for(int i=0; i<6; i++) {
  int rowStart=0;
  if(i%2==0) rowStart=1; // even rows skip 1st square
  for(int j=rowStart; j<8; j+=2) {
    newBlockAt(i,j);
  }
}
```

28.4 break, continue

Loops have two special commands which seem like cheating, but are so nice we use them anyway.

The first one is `break;`. It quits a loop immediately. If it's a `for` loop, it won't even run the third `i++` part.

In some function examples, I've been using a mid-loop `return;` to quit a loop right away. `break` is the same idea, but better.

This clumsy `break`-using loop searches an int-list for the first negative number. As soon as we find one, we can quit, with `i` on that number:

```
// find index of first negative number:
int i;
for(i=0; i<A.Count; i++) {
  if(A[i]<0) break; // quits loop right now
}
if(i>=A.Count) print( "none are negative" );
else print( "index " + i + " is " + A[i] );
```

Notice how `i` needed to be declared before the loop. Otherwise it would be local to the loop and would be thrown away.

A neater way is using extra variables to record what you need. This checks for the first `'e'` in a string and saves the index:

```
int eIndex=-1; // fall-through "not found" value
for(i=0; i<w.Length; i++) {
  if(A[i]=='e') { eIndex=i; break; } // set answer and quit
}
```

Without the `break`, this finds the last `'e'`.

Then here's a fakey one. I want to count the `'z'`s, but strings with a space don't count (if we see a space, the answer is 0):

```
int zCount=0;
for(i=0; i<w.Length; i++) {
  if(A[i]==' ') { zCount=0; break; } // any space ruins it
  if(A[i]=='z') zCount++;
}
```

`break` won't let us do anything we couldn't do before, but it can make the code look nicer. When you see a `break` in a loop, it means we've seen all we need to see. It's often nicer than a `done=true;` style loop.

A common trick is a loop that runs forever, but you plan to use `break;` to stop it. The loop test is usually `while(true)`. Everyone knows that means you're using `break` or `return` to get out.

Here's a forever loop that rolls 2 6-sided dice until they get different numbers:

```
int d1=-1, d2=-1;
while(true) {
  d1=Random.Range(1,6+1);
  d2=Random.Range(1,6+1);
  if(d1!=d2) break; // yay! we got what we wanted
}
print("rolled " + d1 + " and " + d2);
```

A way to read this is "keep rolling both dice, quit when they're different numbers."

A funny thing is how the `break;` test is the opposite of what the loop test would be. As a normal loop, this would be `while(d1==d2)`. But the break uses `if(d1!=d2)`. The loop condition is when we keep going, the `break;` condition is when we stop.

Here's a `for(true)` plus `break`, abused to make a loop hitting every box in a list (never do this, but it's still pretty cool):

```
for(int i=0; true; i++) { // counts 0,1,2 ... forever
  if(i>=A.Count) break; // quit past end of list
  print(A[i]);
}
```

You usually don't have to think about `break;` while you write code. A loop will just naturally have a place where it would be nice to quit right now, and you remember the `break;` command.

Just so you know, `break`'s only quit one loop. A `break;` inside a nested loop will quit only the inside. The outer loop will keep going as normal. That's almost always what we want.

The other loop shortcut command is `continue;`. It's also used only by itself, inside a loop. It "skips to the next one" – it jumps to just before the final close-curly.

In this example it helps us print even numbers above 10, from some list:

```
// print even numbers over 10:
for(int i=0; i<A.Count; i++) {
  int n = A[i];
  if(n<=10) continue; // too small, skip to next
  if(n%2!=0) continue; // not even, skip to next
  print(n);
}
```

This loop hits every box in `A`. The `continue` command doesn't quit the whole loop early, just the step we're on now.

But otherwise `continue` is like `return` or `break`, since it magically changes flow-of-control. That last `prints(n)` loop line always runs, but only if the `continue` lets it get there.

`continue` is mostly used when you'd need to use complicated `if`'s. For example this uses `continue` to count words in a list that don't start with # and have the same first and last letters:

```
int firstLastSame=0;
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length<2) continue; // skip words without 2+ letters

  char first=w[0];
  if(first=='#') continue; // skip words starting with #

  chart last=w[ w.Length-1 ];
  if(first==last) firstLastSame++;
}
```

Without continue, we'd have to use two ugly `if`s:

```
// first same as last, without continue:
int firstLastSame=0;
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length>=2) {
    char first=w[0];
    if(first!='#') {
      chart last=w[ w.Length-1 ];
      if(first==last) firstLastSame++;
    }
```

```
  }
}
```

A common `continue` mistake is using it in a `while` loop and skipping the final `i++` (that can't happen with a `for` loop). This prints every word in `W`, skipping empty-strings, but has a fatal bug:

```
// infinite loop if we have "":
int i=0;
while(i<W.Count) {
  string w = W[i];
  if(w=="") continue; // skip last two lines. Repeats on same i !
  print(w);
  i++;
}
```

On an empty string, the `continue` jumps back before `i` increases. It will see the same empty string again, go back, forever.

Fun `continue` observation: adding it as the last line in a loop does nothing. It skips to the end, past 0 lines.

You can use `break;` and `continue;` in the same loop. They each do their normal thing.

Suppose we want to search through a list for short words (5 letters or less) and add them into one long string, but not past 25 letters total. If a word would go over, we skip it. This loop does that:

```
// silly loop to add all short words in W together, not past 25 letters:
string sum="";
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length>5) continue; // skip long words
  if(w=="") continue; // there's no point adding ""

  int len1=sum.Length, len2=w.Length;
  int totalLen = len1+len2;
  if(ltotalLen>25) continue; // this word is too long to add, but keep trying

  sum += w;
  if(totalLen==25) break; // may as well quit, since at the max
}
```

The `break;` happened to be at the end, but there's no rule about that. I think this reads pretty well, if you remember `continue` means go to the next one and `break` means quit.

28.5 List nested loops

There are some natural `List` nested loops, for example checking whether two lists have any numbers in common.

The obvious way is how we'd do it by hand. To check two pages of numbers, put your finger on the first number on page 1, and scan page 2 for a match. Repeat for the next number on page 1. That's a nested loop:

```
bool nothingInCommon(List<int> A, List<int> B) {
  for(int i=0; i<A.Count; i++) {
    // compare A[i] to everything in B:
    for(int j=0; j<B.Count; j++) {
      if(A[i] == B[j]) return false;
    }
  }
  return true;
}
```

The outer loop selects `A[0]`, then the inner loop compares it to everything in `B`. Then the outer loop moves to `A[1]`, and the inner loop compares that to everything in `B`, and so on.

`i` goes with `A`, and `j` goes with `B`. I get them mixed-up a lot, using `A[j]` and `B[i]` by mistake, which off-end crashes and gives wrong answers.

Here's the same idea, but it counts how many things in `A` are also in `B`. It uses a `break;` so nothing gets double-counted ([3,6] and [3,3] have only one number in common):

```
// how many in A are also in B:
int inOtherCount(List<int> A, List<int> B) {
  int count=0;
  for(int i=0; i<A.Count; i++) {
    for(int j=0; j<B.Count; j++) {
      if(A[i] == B[j]) { count++; break; }
    }
  }
  return count;
}
```

The `break;` quits the one loop it's in, which is perfect. We check whether `A[0]` matches anything in `B`. If we find a match, we count it, quit the `B` loop, and move onto `A[1]`.

Checking just one list for duplicates is the same idea: take each item, and compare it to every other. But there are two differences. One is we can't compare an item to itself (if we did, every list would appear to have duplicates).

The other is we may as well skip double-compares: we compare box 0 to box 1, so there's no need to compare box 1 to box 0.

Combining those ideas, we get this rule: the inner loop should compare `A[i]` to everything that comes after it in the list.

Another rule, not as important: the outer loop can stop one before the end (the last item has already been compared to everything):

```
bool hasDuplicate(List<int> A) {
  for(int i=0; i<A.Count-1; i++) { // no need to check last one
    for(int j=i+1; j<A.Count; j++) { // compare with everything past i
      if(A[i]==A[j]) return true;
    }
  }
  return false;
}
```

That inner loop is a little like the triangle nested loops, the way it starts based on `i`. For a double-check: suppose `i` is 3. The inside loop starts by comparing `A[3]` to `A[4]`, which is what should happen.

28.5.1 Sorting

There are some very sneaky nested loops that sort a list. This is getting into the area of algorithms – where three days of work comes down to a few innocent-looking loops. But these aren't too bad.

We know how to find the smallest item in a list. To sort, we'll do that, add it to a new one, and cross it out from ours. Then we'll do it again: find the new smallest (since we crossed-out the old smallest,) add that to the end of the new list, and cross that one out.

It works like this:

```
A: 4 3 1 5 2 <-unsorted list
B:

A: 4 3 99999 5 2
B: 1

A: 4 3 99999 5 99999
B: 1 2
 ...
```

It's a loop around "find the smallest" and is called a selection sort. Here's the code for that crude idea above:

```
// A is unsorted, slowly sort it into B:
List<int> B = new List<int>();
```

```
for(int i=0; i<A.Count; i++) {
  // find smallest in A (code copied from old chapter):
  int smallIndex=0; // A[0] is smallest, so far
  for(int j=1; j<A.Count; j++) {
    if(A[j]<A[smallIndex]) smallIndex=j;
  }
  // now smallIndex is on the smallest thing in A, use it and cross it out:
  B.Add(A[smallIndex]);
  A[smallIndex]=99999; // cheap way to cross it out
}
```

These are tricky. If you write something like this, put some `print`'s in the middle so you can see where it's messing up (the first try always messes up).

There's an even cleverer, more confusing way to do it with just one list. We find the smallest and *switch* it into the first position. Then find the smallest out of the rest and switch that into the second spot.

This is a real selection sort (you can probably look it up for more explanation):

```
for(int i=0; i<A.Count-1; i++) {
  // skip everything before i -- it's sorted,
  // find smallest thing from i to end:
  int smallIndex=i;
  for(int j=i+1; j<A.Count; j++) {
    if(A[j]<A[smallIndex]) smallIndex=j;
  }
  // now swap it into A[i]:
  int temp = A[i];
  A[i]=A[smallIndex];
  A[smallIndex]=temp;
}
```

If you liked this, insertion and bubble sort are the other two you could look up. They're both nested loops that make no sense until you read the plan.

Chapter 29

Struct in struct

This section is another with no new rules – just useful tricks. Now that we have structs, and lists, and string indexes, we can combine them to make big data structures.

Usually these are pretty natural - maybe you have several people who can own several dogs each. With practice, that's easy to put into the computer and use.

Big data structures are also good practice writing double and triple loops.

29.1 Lists of structs

Making a list of `Cow` structs looks like this:

```
// review of Cow:
[System.Serializable]
public struct Cow { public string name; public int age; }

public List<Cow> Barn; // list of Cows
```

As we know, in the Inspector you have to type a non-zero number for the size of that list. Popping it open will show lots of `Cow`'s, which you can also pop open.

More interesting is how to search around the Barn in code. The trick is the same as for things like `c1.spotCol.r` for the old cow's with colored spots. Go left-to-right, using the options for that type.

Here's code using `Barn`:

```
int n=Barn.Count; // # of cows, same as normal
n=Barn[0].age; // age of first cow
Barn[1].age=4; // make 2nd cow be 4 years old
Barn[1].name="Lu-Lou Cow";
```

Barn is a list, so we can use Count, or can pick out one item using []. Barn[0] is a Cow, so we can use dot-age or dot-name.

To compare, some errors:

```
Barn.age; // error - have to say which cow
Barn[0].Count; // error - cows don't have a count
```

We can also copy entire cows in and out of the barn. This is really the same boring stuff we did with lists of ints, or normal structs:

```
Cow cc; cc.name="Kow"; cc.age=5;
Barn[2] = cc; // simple var-assign, with a whole cow
Cow c3=Barn[3]; // c3 is a copy of that cow
Barn[4]=Barn[5]; // copy entire cows between list boxes
```

We can create the entire Barn list ourselves, adding each Cow. This makes six 1-year olds named cowy:

```
Barn = new List<Cow>(); // size 0
for(int i=0; i<6; i++) {
  Cow c; c.name="cowy"; c.age=1; // a normal cow
  Barn.Add(c); // <-standard Add
}
```

A list of classes works the same, except they also need to be new'd. This makes a list of six Dog's (Dog is a class):

```
List<Dog> Kennel= new List<Dog>(); // size 0
for(int i=0; i<6; i++) {
  Kennel.Add(new Dog());
}
```

That's a little sneaky. new Dog() creates a Dog and returns a pointer to it. Normally we assign that to a variable, but adding it to the list is just as good, since that's the final place for it.

We could use two steps instead, with a middeman variable:

```
  Dog d=new Dog(); // 2-step version. d is a temp
  Kennel.Add(d);
```

Here's a bugged version. See if you can spot the problem. Hint: how many Dog's are there?:

```
Kennel = new List<Dog>();
Dog d=new Dog(); // dog is a class, so each new creates one
for(int i=0; i<6; i++) {
  Kennel.Add(d); // <-standard Add
}
```

This creates 6 pointers, all aimed at the same Dog.

A neat Unity trick is making a list of `GameObject`'s. This lets us drag many Cubes into the Inspector:

```
public List<GameObject> Blocks; // size in Inspector, drag in Cubes
```

We'd change them the usual way. `Blocks[0].transform.position=pos;` would move the first block. Or this would turn every other block red:

```
Color cc = new Color(1, 0, 0); // red
for(int i=0; i<Blocks.Count; i+=2) {
  Blocks[i].GetComponent<Renderer>().material.color=cc;
}
```

We can also fill that list ourself, by adding instantiated objects. They'll all be in the same spot, but we can move them later:

```
public GameObject ballPrefab; // drag in a Cube prefab
List<GameObject> Blocks;

void Start() {
  Blocks = new List<GameObject>(); // size 0, so far

  for(int i=0; i<10; i++) {
    GameObject gg=Instantiate(ballPrefab);
    Blocks.Add(gg);
  }
}
```

This solves a problem from before. We were able to create dozens of balls, but we had no place to save pointers to them all. A list is the perfect place for that.

The shortcut `Blocks.Add(Instantiate(ballPrefab));` would also work. I prefer the extra step of declaring `gg`. We'll probably want to use it to color or place each fresh ball.

29.2 Structs with Lists in them

Structs with lists inside them also aren't special, but look interesting. As usual, the list needs to be `new`'d. Here's a `Goat` struct which has a list of everything the goat eats:

```
struct Goat {
  public string name;
  public int age;
  public List<string> eats; // currently null
}
```

Some sample code playing around with `Goat g;`, showing how to use `eats`. This goat will eat tin cans and old shoes:

```
g.name = "Billy";
g.eats.Add("cans"); // error - null reference exception. eats not created yet
g.eats = new List<string>();
g.eats.Add("tin cans");
g.eats.Add("old shoes");
int n=g.eats.Count; // 2. g eats two things
```

Then some errors:

```
g.Add("grapes"); // error. g is not a list
g[0]="raisins"; // same error. g is not a list
int n=g.Count; // ditto
```

We could also create an eat-list first, then assign it all at once. Here `BillyFood` is like a temp, passing the list of foods along to `g2`:

```
List<string> BillyFood = new List<string>();
BillyFood.Add("crates"); BillyFood.Add("barrels");
Goat g2;
g2.eats = BillyFood; // g2 now eats crates and barrels
```

`new`'s can pile up on us. Suppose `Goat` was a class. We'd need to create and and to create it's list:

```
Goat g = new Goat(); // new for class
g.eats=new List<string>(); // still need to new the eats list
```

29.3 Lists of strings tricks

A string can use indexes and has a length. So an list of strings is sort of like a 2-dimensional list. A warm-up review:

```
public List<string> W;  // ["cow","banana","anteater", aardvark"]

int n=W.Count; // 4 words in the list
n=W[0].Length; // 3 letters in cow
```

Now the new-looking part, using two `[]`'s in a row:

```
char ch=W[0][2]; // w (3rd letter of cow)
ch=W[2][0]; // a (1st letter of aardvark)
```

The cool thing is it's not a new rule. `W[0]` is `"cow"`, which is a string. So of course adding `[2]` after grabs the `'w'` character.

Now we can write this cool nested loop to count the total number of `a`'s in every word in the list:

```
int aCount=0;
for(int i=0; i<W.Count; i++) { // each word
  for(int j=0; j<W[i].Length; j++) { // each letter in word
    if( W[i][j]=='a' ) aCount++;
  }
}
```

As usual, it's easy to mix up the `i`'s and `j`'s and get wrong counts and out-of-range crashes. `W[word][letter]` might be better variable names if you don't use `i` and `j` for every nested loop.

A cute trick with arrays of strings is to use them to make a map. This usually isn't the best way, but it's fun and a nice example.

For a 3x4 map, we can make a list with 3 strings, all length 4. We'll make the rule that o is an open space, and X has a block in it:

```
List<string> M=new List<string>();
M.Add("xoox");
M.Add("xxxo");
M.Add("xooo"};
```

We can use a nested loop to make that picture. Cubes go on the X's. This is a copy of the nested grid loop from last chapter, except we check the map before deciding to make a block:

```
for(int i=0; i<M.Count; i++) { // 3 rows
  for(int j=0; j<M[i].Length; j++) { // each letter in word: x or o
    if( M[i][j]=='x' )
      newBlockAt(i,j); // from previous chapter
  }
}
```

It doesn't look as nice as it could since `newBlockAt` leaves a little space between. In this case it would be better to have them touching, like one long solid wall.

To get a real feel for working on a grid, we can count how many walls are next to spot (x,y). We take a pretend step in each of the 4 directions:

```
int nearWalls=0;
if(M[y][x-1]=='x') nearWalls++; // left
if(M[y][x+1]=='x') nearWalls++; // right
if(M[y+1][x]=='x') nearWalls++; // down (b/c of the picture)
if(M[y-1][x]=='x') nearWalls++; // up
```

It crashes if we're on an edge (fixed with `if`'s). Math like this is common for walking around in a grid: `M[y][x-1]` is the space to the left of us, and so on.

29.4 2D lists

A real 2D list is just a list of lists. The cool thing is there are no new rules - we can make them out of what we have now.

For example, `List<List<string>>` is a list of lists of words. The inside part is `List<string>`, a normal list of strings. Then `List< ... >` goes around it. So it's a list, containing lists of strings (if you can figure out the TV show about a serial killer who kills serial killers, you can figure out this).

To get started, suppose we have some lists of words for a Mad Lib:

```
public List<string> Subjects; // ["Bob", "The cat", "My face"];
public List<string> Verbs; // ["runs", "sits", "cries", "inspects", "tolerates"];
public List<string> Adjectives; // ["cowardly", "heavy duty"];
public List<string> DirectObjects;
  // ["clouds", "fish", "tears", "truth", "gold", "needles"];
```

We eventually want to use these to randomly make 4-part sentences, choosing one from each, like "The cat inspects heavy duty truth".

We'll put those four lists into a big list:

```
List<List<string>> AllParts=new List<List<string>>(); // a list of lists
AllParts.Add(Subjects);
AllParts.Add(Verbs);
AllParts.Add(Adjectives);
AllParts.Add(DirectObjects);
```

Now `AllParts` has four string lists in it. `AllParts[1]` is the verb list, and `AllParts[1][0]` is the first verb, `"runs"`.

A picture would look like:

```
AllParts
[0] -> Bob, The cat, My face
[1] -> runs, sits, cries, inspects, tolerates
[2] -> cowardly, heavy duty
[3] -> clouds, fish, tears, truth, gold, needles
```

Playing with it to get a feel, we can do things like this:

```
AllParts[2].Add("slow"); // add a new adjective
AllParts[0][1]="The dog"; // replace cat with dog
AllParts.Count; // 4. Four lists of words
AllParts[3].Count; // 6. clouds, fish ... is 6 things
AllParts.Add(new List<string>()); // add 5th sentence part
AllParts[4].Add("?"); AllParts[4].Add("!!!"); // possible sentence enders
```

We can make a random sentence using a loop to pick from each part:

```
string madLib="";
for(int i=0;i<AllParts.Count;i++) {
  if(i!=0) madLib+=" "; // space in front of every word except the first
  int randPos=Random.Range(0, AllParts[i].Count);
  madLib+=AllParts[i][randPos];
}
```

Notice how the last line uses the usual `[][]`. Down the side to the sentence slot we're on now, then across to the random word for that slot.

There's one final super-cool thing a list of lists lets us do. Each of the inside lists counts as a normal list, and we can do normal list things with it. For example, we could write a function to pick a random word from a normal list:

```
string getRandWord(List<string> W) { return W[Random.Range(0, W.Count)]; }
```

This is a totally normal list function, that knows nothing about lists of lists and won't work with them. But we can still use it in our 2D list sentence maker:

```
  for(int i=0;i<AllParts.Count;i++) {
    ...
    madLibs+=getRandWord(AllParts[i]); // <- calling normal list function
  }
```

This works since `AllParts[i]` counts as an ordinary list of strings. `getRandWord` is happy to take it as an input and pick out a random work from that one list.

A 2D integer list of lists feels more griddy, but it's made mostly the same way. This makes a 4x4 grid:

```
List<List<int>> G=new List<List<int>>(); // 2D list
for(int i=0;i<4;i++) G.Add( new List<int>() );
// now we have 4 empty lists. Grow them all by four 0's:
for(int i=0;i<G.Count;i++) // each column
  for(int j=0;j<4;j++) G[i].Add(0); // add four 0's, up this column
```

Probably the most confusing line is `G.Add(new List<int>());`. The inside of it creates a fresh int-list. Normally we assign that to a variable, but we're allowed to skip the middle-man and jam it straight into the list.

We can think of this as a real grid:

```
[3]  0   0   0   0
[2]  0   0   0   0
[1]  0   0   0   0
[0]  0   0   0   0
G-> [0] [1] [2] [3]
```

I put `G` across the bottom, with the lists going up, so we could use `G[x][y]` in the more normal way. For example, `G[3][0]` is the lower right.

These next two single loops put 7's up the right side, and 8's across the middle (the last 8 overwrites a 7). Notice which slots have the variables:

```
for(int y=0;y<3;y++) G[3][y]=7; // up last column
for(int x=0;x<3;x++) G[x][1]=8; // across 2nd row

// result:
[3]  0   0   0   7
[2]  0   0   0   7
[1]  8   8   8   8
[0]  0   0   0   7
G-> [0] [1] [2] [3]
```

We can even use a normal `List<int>` function on each column, since each column is a list (but not on a row.) This sets everything in an ordinary list to a value:

```
void setVal(List<int> A, int val) {
  for(int i=0;i<A.Count;i++) A[i]=val;
}
```

Now `setVal(G[3],9);` replaces those 7's with 9's. Of course, it only works for columns – each column is one list. There's no way to send a row to `setVal` since each row is 1 box from each of the column lists.

29.5 Larger structures

We can use these rules to make structs with lists of structs, and larger. It's not that complicated, since you make them based on the data you already have.

For example, making a list of goats (which have lists of what they eat):

```
// repeat of Goat class:
class Goat {
  public string name;
  public List<string> eats;
}

// make goat list:
List<Goat> GG=new List<Goat>();
for(int i=0; i<8; i++) { // make 8 empty goats
  Goat g=new Goat();
  g[i].name = "goat #"+(i+1);
  g[i].eats = new List<string>();
```

```
  G[i].eats.Add("goat chow");
  GG.Add(g);
}
```

Notice we have to `new` the goat-list, `new` each goat in the list, and `new` each `eats` list for each goat.

A partial picture:

```
GG 0 -> | name: goat#1
        | eats -->  ["goat chow"]

   1 -> | name: goat#2
        | eats -->  ["goat chow"]

   2 -> | name: goat#2
        | eats -->  ["goat chow"]
```

A few sample lines that do and don't work:

```
int n = GG.Count; // 8 goats
n = GG[0].Count; // ERROR -- goats don't have a length
n = GG[0].eats.Count; // 1 food
string s=GG[0].eats[0]; // "goat chow"
n = GG[0].eats[0].Length; // 9 letters in "goat chow"
GG[2].eats.Add("shoes"); // legal. also eat shoes
GG[2].Add("cake"); // ERROR - goats aren't lists
```

We can run some interesting loops through this. This nested loop counts how many goats like a certain food:

```
int howManyLikeThis(List<Goat> G, string food) {
  int foodCount=0;
  for(int i=0; i<GG.Count; i++) { // check each goat
    // check all foods this goat eats:
    for(int j=0; j<G[i].eats.Count; j++) {
      if(GG[i].eats[j] == food) {
        foodCount++;
        break; // don't double-count this goat; quit eats loop
      }
    }
  }
  return count;
}
```

It checks the obvious way: scan every goat, look through what that goat eats, count it and skip to the next if you get a food match.

Here's a triple loop to count the total Z's in all food items (if five goats like "pizza", one of them twice, this would count as 12 z's):

```
int zCount=0;
for(int i=0; i<GG.Count; i++) { // each goat
  // each food:
  for(int j=0; j<GG[i].eats.Count; j++) {
    // each letter:
    for(int k=0; k<GG[i].eats[j].Length; k++) {
      if(GG[i].eats[j][k]=='z') zCount++;
    }
  }
}
```

`GG[i].eats[j][k]` has four total look-ups, but it's fine since we really need that many: which goat, what it eats, which food, which letter.

29.6 Giant game board example

Previously we used a nested loop to create a 2D grid of Cubes. But we couldn't save pointers to them, which meant we couldn't find and change them later. Now we can, which means we can almost make a game.

Our game will eventually allow you to toggle spaces (selected or un-selected) letting you make crude pixel shapes. First a simple class to hold data for each space:

```
class BoardSquare {
  public GameObject cube; // pointer to Cube at this space
  public bool selected; // selected squares change color
}
```

Then we'll make a simple 2D list, to hold them, for when we make the cubes, later:

```
// 2D grid of boardSquares:
List<List<BoardSquare>> Board; // want 8 wide, 6 high grid

void Start() {
  Board = new List<List<BoardSquare>>(); // main 2D list
  for(int i=0;i<8;i++) {
    Board.Add(new List<BoardSquare>()); // add empty column
    // grow column to 6 high:
    for(int j=0;j<6;j++) Board[i].Add(new BoardSquare());
  }
```

Now `Board.Count` is 8 (going across the bottom.) `Board[0].Count` is 6 (going up.) `Board[7][5].selected=true;` selects the upper-right corner.

Making the blocks and hooking them up is nothing special. We'll use `Board[x][y].cube=gg;` as we create each real Cube, then math to get the proper arrangement:

```
public GameObject cubePrefab; // drag in

  // the rest of Start:
  for(int x=0; x<8; x++) {
    for(int y=0; y<6; y++) {
      Vector3 pos; pos.z=0;
      pos.x=-4.0f+x*1.2f; // x goes across, from left
      pos.y=-3.0f+y*1.2f; // y goes up, from bottom
      Board[i][j].cube = Instantiate(cubePrefab, pos); // create and place shortcut
      Board[i][j].selected=false;
    }
  }
```

That gives us something new. We have 6 by 8 Cubes on the screen, and we now have a way to find and change them all. `Board[x][y].cube` is that Cube.

The "game" will be to move around and toggle the current square on/off. We'll be able to make a crude picture, which is still pretty cool. I'll break it into using the keys to move, and using space to toggle the color of the cube we're on. Update will call each function:

```
int row=0, col=0; // where we are on the board now
void Update() {
  moveCheck();
  selectCheck();
}
```

`selectCheck` will toggle the color of the current cube. Without movement, we can never leave square (0,0), but that's good enough to test. We can tap space and watch the corner cube change color:

```
void selectCheck() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    BoardSquare bs=Board[col][row]; // <- using row/col to look up current space
    bs.selected = !bs.selected; // flip T/F
    // now set to correct on/off color:
    Color cc=Color.blue;
    if(bs.selected) cc=Color.red;
    bs.cube.GetComponent<Renderer>().material.color = cc;
  }
}
```

The best part of this is the top line with `bs=Board[col][row]`. It picks out the current spot, based on row and col, exactly how we want in a 2D grid. Once we have that, we use `bs.selected` and `bs.cube` to look at the parts.

Without using `bs` as a shortcut, the last line would start with: `Board[col][row].cube.GetComponent`. That's long, but reading it left-to-right should look fine.

Movement is the arrow keys. left/right change the column, up/down change the row. I decided to have off-edge wrap around, which requires 4 ugly `if`'s. The current cube is a tad smaller, to show that we're on it:

```
void moveCheck() {
  int oldRow=row, oldCol=col; // save old row/column

  if(Input.GetKeyDown(KeyCode.RightArrow)) {
    col++; if(col>=8) col=0; } // wrap-around
  if(Input.GetKeyDown(KeyCode.LeftArrow)) {
    col--; if(col<0) col=7; }
  if(Input.GetKeyDown(KeyCode.UpArrow)) {
    row++; if(row>=6) row=0; }
  if(Input.GetKeyDown(KeyCode.DownArrow)) {
    row--; if(row<0) row=5; }

  // if anything changed, show new spot:
  if(oldRow!=row || oldCol!=col) {
    //  Reset old square to normal size:
    Board[oldCol][oldRow].cube.transform.localScale = new Vector3(1,1,1);
    //  Make new square smaller:
    Board[col][row].cube.transform.localScale = new Vector3(0.8f,0.8f,0.8f);
  }
}
```

Doing something special to the current item is always a pain. That's what the second half of the code is doing. We need to check whether we moved, reset to old square's size (which is why we needed to save it in old row and col), and finally change the new square.

Be sure and look at `Board[col][row].cube.transform.localScale`. Long, but pretty.

This type of set-up for a board is common. Each square might need to know the terrain type, buildings in it, who captured it, and so on. A class or struct is good for that. A link to the visible part (`cube`, here) is just one more field.

29.7 List of indexes

This is an old trick, not very common anymore, but it's a fun exercise. Suppose you want to make a list of *some* of the things in a list. If you wanted items 1,4 and 6. You'd make a list `[1,4,6]`, which is an array of indexes. Here it is in code:

```
public List<string> Ani; // ["ant","bear","cow","deer","eel","ferret","goat","hawk"]
public List<int> InZoo; // [1,4,6] // stands for bear,eel,goat

void Start() {
  // print animal names in the zoo:
  for(int i=0;i<InZoo.Count;i++)
    print( Ani[InZoo[i]]); // bear eel goat
}
```

We've never seen anything like `Ani[InZoo[i]]` before. It's a nested look-up, doing the thing above. `InZoo[i]` goes through 1, 4, 6. So `Ani[InZoo[i]]` goes through items 1, 4 and 6: bear, eel, goat.

Here's a longer version of the same trick. Suppose we want to print the animals in a random order. A cheap way is to shuffle a list with 0-7, then read the animals in that order:

```
List<int> RandIndexes=new List<int>();
for(int i=0; i<Ani.Count; i++) RandIndexes.Add(i); // 0-7
shuffle(RandIndexes); // old function. ex: 2,6,0,1,5,7,4,3

for(int i=0;i<RandIndexes;i++)
  print(Ani[RandIndexes[i]]);
```

The last line is the same nested look-up as before.

This is really a version of pointer thinking. We can't have pointers to strings, but we can use the indexes like pointers. It's not common, but it's a nice way to practice.

29.8 Internal list pointers

If we have a list of objects, we can pick out some of them with another list of pointers. For example `AllDogs` is a list of every dog. `SledTeam` won't have any new dogs – it will only point to things in `AllDogs`:

```
public List<Dog> AllDogs; // created and filled in Inspector
List<Dog> SledTeam; // will point to some things in AllDogs

void Start() {
  SledTeam=new List<Dog>();
  for(int i=0;i<4;i++) SledTeam.Add(null); // 4 empty Dogs pointers

  // start with first 4 dogs, but can change:
  for(int i=0;i<4;i++) SledTeam[i]=AllDogs[i];
}
```

You can't tell by the definition. but in our minds `SledTeam` isn't holding Dogs. It will never have any Dogs of its own. It's purpose is to pick 4 dogs out of the main Dog list.

This would print the names of all dogs on our team:

```
  for(int i=0;i<SledTeam.Count;i++) {
    if(SledTeam[i]==null) print("(missing)");
    else print(SledTeam[i].name);
```

Here, checking `SledTeam[i]==null;` makes perfect sense. `null` is a perfectly good value – we haven't picked a Dog for that position yet. We're also imagining `SledDog[i].name`as reaching over into `AllDogs`:

```
AllDogs:  0  1  2  3  4  5  6  7  8  (9 real dogs)
             ^  ^     ^  ^

              \/     /  /
              /\    /  /
SledTeam:    o  o  o  o  (4 arrows)
```

There's another version of this trick where a class has pointers to itself inside of it. For example, all of our Rabbits can have another Rabbit best friend:

```
[System.Serializable]
public class Rabbit {
  public string name;
  public int age;

  [System.NonSerialized]
  public Rabbit bestFriend;
  // not part of me -- an arrow to another rabbit
}

public List<Rabbit> AllRabbits; // set up in Inspector
```

Clearly, we can't have an actual rabbit in a rabbit, since that would have another rabbit inside of it, going on forever. But it's fine to have `bestFriend` be an arrow. We'll never use `new` on it. Its purpose is to point out some other pre-made rabbit (or be `null` if we have no best friend).

A neat thing is that Unity can't tell. It assumes gameObject variables pointer to gameObjects it created. But it assumes a rabbit is a rabbit and tries to make one for `bestFriend`, triggering infinite rabbits. A special safely check kicks in after it makes 7 levels of nested rabbits and gives an error.

`[System.NonSerialized]` is there to explain to Unity that `bestFriend` is like a gameObject pointer – it's only an arrow. It says not to create it. All of this can get complicated quickly, but deep down it's the Real vs. Arrow issue that all reference types have.

This code makes every pair of rabbits be mutual best friends:

```
for(int i=1;i<AllRabbits.Count;i+=2) { // 1,3,5,7 ...
  Rabbit r1=AllRabbits[i-1], r2=AllRabbits[i];
  r1.bestFriend=r2; // Ex: Bunny[0] and [1] mutually like each other
  r2.bestFriend=r1;
}
```

Or we could randomly assign best friends, with some anti-social rabbits whose best friends are `null` (1 in 6 chance):

```
for(int i=0;i<AllRabbits.Count;i++) {
  Rabbit fr=null; // no best friend, so far
  if(Random.Range(1,6+1)!=1) { // 1 in 6 for no friend
    int friendNum=Random.Range(0,AllRabbits.Count);
    if(friendNum!=i) fr=AllRabbits[friendNum];
    // can't be friends with yourself
  }
  AllRabbits[i].bestFriend=fr;
  // note: assigning null is legal, and fine
}
```

Basically, `bestFriend` could aim at any other rabbit, or `null`, and can double-up. That's not as fair as pairs of best-friend rabbits, or maybe it is, if you're a rabbit.

Now that we've set best friend arrows, we can use them. This would count how many rabbits have a best friend named Thumper:

```
int count=0;
for(int i=0;i<AllRabbits.Count;i++) {
  Rabbit bf=AllRabbits[i].bestFriend;
  if(bf!=null) { // we may not have a best friend
    if(bf.name=="Thumper") count++;
  }
}
```

Finding the most popular rabbit needs a nested loop. We'll check every rabbit, counting how many other rabbits like it, remembering the highest total:

```
int bestIndex=-1; // most popular rabbit, so far (Price is Right strategy)
int bestVotes=-999; // votes for that rabbit
for(int i=0; i<AllRabbits.Count; i++) {
  Rabbit currentBunny=AllRabbits[i]; // shortcut to this bunny
  int votes=0; // how many other bunnies like this one:
  for(int j=0; j<AllRabbits.Count; j++) {
    if(AllRabbits[j].bestFriend==currentBunny) votes++;
  }
  if(votes>bestVotes) { bestIndex=i; bestVotes=votes; }
```

```
}
if(bestIndex<0) print("no rabbits like any other rabbits");
else print("Most popular is "+AllRabbits[bestIndex]);
```

It's a nested loop because each rabbit knows who it likes, but not who likes it. We need another loop for that.

Chapter 30

Member functions

When we make a class, we often write some helpful functions using it. For the `Dog` class we had `setDog(d1,"Spike",2)` and `showDog(d1);`. In our minds, those were `Dog` functions.

Member functions is a trick to make them feel more like Dog functions. We can have `d1.set("Spike",2);` and `d1.show();`, and it feels like `d1` is setting itself, or running its personal `show` function.

This is really the other half of how to make a class or struct. After we add fields, we also add functions. Both are used with a dot.

Member functions don't actually do anything new. We're merely moving around where we define some functions and changing the way we call them. This might feel like a lot of rules for not much gain. But it's a nice way to organize things, all of the built-in classes and structs use them, and we'll need them for some Object-Oriented tricks later on.

30.1 Example and motivation

Here's a rewrite of the old setupDog function as a member function:

```
class Dog {
  public string name; public int age;

  // this is a member function:
  public void setup(string nm, int howOld) {
    name=nm; age=howOld; // <- name, age? By themselves?
  }
}
```

The basic rule is that we can use `name` and `age` directly. We learned you can never do that – you have to say which Dog it is first. But member functions have that built into the call. `pet2.setup("Spot",5)` remembers `pet2` called

it. Clearly `name` is the name of `pet2`. If we run it again with `d1.setup("",0)`, we're now clearly using `d1`'s fields.

It's a very clever trick. You have to call them using Dog-dot – that's a rule. So a member function is guaranteed to be running for one Dog. It gets to use that Dog's fields in a simple, fast way.

There's nothing special showing how `setup` is a member function. Functions written inside of a class are automatically member functions. As we can see, they use parameters the normal way. They can return things. They have only that one extra rule: always running "for" some Dog.

It helps to see why we're doing all this work. Some of the advantages of having member functions:

- Putting `d1`-dot in front makes it easier to see this is a `Dog` function when we're reading it.
- It also makes it easier to see how `("Spot",2)` is the real input. `d1.setup("Spot",2);` makes it look more like an assignment statement, which it mostly is.
- When we're writing code, it's easier to find the `Dog` functions. We can type `d1`-dot and search the pop-up: the same pop-up that shows the fields also shows the member functions.
- We can use shorter function names. As a regular function, `setupDog` seemed good. But as a member function, `setup` is fine, since we know we're inside `Dog`.
- The "using this dog" `name`/`age` shortcut can make the inside of member functions a little nicer.

These little things add up. Once you know how, instead of writing a helpful function, you write a helpful member function.

Here are the member function rules listed out:

- Member functions are written inside the class. The extra word `public` must be in front, but otherwise they look normal.
- They can only be run by a variable of that type, using variable-dot-function. Like `d1.setup("Spot",2)`. Trying to run them by themself is an error. `setup("Gabby",6);` is no good.
- The variable that runs them is an automatic input. It doesn't have a name inside the function. Instead you use its member variables "naked." In a dog member function, just `name` and `age` means the name and age of the dog that called you.

- Member functions, even in structs, are allowed to change the parts of what called them. For example `frog1.doFrogStuff();` could be changing variables inside of `frog1`.
- The order you write them doesn't matter. Member functions can come before, after or in-between member variables.
- Forgetting `public` in front isn't an error, but you won't be able to use the function (it won't appear in the pop-up after `d1`-dot, and more importantly, it gives a *function not found* error if you type it anyway).

30.2 More simple examples

Dog examples

Here are some more `Dog` member functions, all with no inputs:

```
class Dog {
  public string name;
  public int age;

  public string desc() { return "Name: "+name+" "+age+" years old"; }

  public void reset() { name=""; age=0; }

  public bool isAPuppy() { return age<=2; }
}
```

We know they really all have 1 input – the dog that calls them.

`desc()` is the same simple "help print me" function we've written for structs before. We'd call it like `string w = dog2.desc();`. As usual, it uses `name` and `age` and knows they mean `dog2`'s name and age. Notice how it returns a value the same way as a normal function.

`reset()` is showing the rule of how you're also allowed to change the fields. `pet1.reset();` would blank out the inside of `pet1`.

`isAPuppy()` only reads `age`, which is fine. We've seen the function-in-an-if trick with regular functions. Now we can use `if(d2.isAPuppy())`.

Member functions with inputs use them in the normal way. This checks whether a dog is within an age range:

```
  public bool inAgeRange(int low, int high) {
    return age>=low && age<=high; // uses age of the calling Dog
  }
```

Now `bool adult = d2.inAgeRange(4,9))` is legal.

You're allowed to have a member function that doesn't use any fields, but you'd never do it. It would be pointlessly confusing. This terrible function picks a random Dog age:

```
class Dog {
  ... // Dog stuff

  public int randomDogAge() { return Random.Range(1,15+1); }
}
```

We be required to call it like `int n=d1.randomDogAge();`. But it doesn't use `d1` for anything, or change it. It runs the exact same no matter what Dog we give it. Why did we require you to run it from a Dog – that's just confusing. It would be better as a normal function.

The most complex member functions take another Dog as input. The standard example is copying an input Dog into ourself. `d1.copyFrom(d2)` makes `d1` a copy of `d2`:

```
  // inside of Dog:
  void copyFrom(Dog dd) {
    name = dd.name; // myName = otherDogsName
    age = dd.age; // myAge = otherDogsAge
  }
}
```

Inside, `dd.name` uses the input Dog, the same as any other function, while `name` is our name.

Here's a simple class for a 2D point with 2 member functions:

```
class Point2 {
  public float x,y;

  public void set(float xx, float yy) { x=xx; y=yy; }

  public float fromCenter() { return Mathf.Sqrt(x*x+y*y); }
}
```

`p1.set(1, 7.2f);` runs the first one. It's allowed to change the inside of `p1`. The input `xx`'s and field `x`'s can be confusing – people try all sorts of naming rules to tell inputs from fields.

`float dist=p1.fromCenter();` runs the second. Nothing special – it reads `p1`'s x and y, and returns the answer (that's the actual formula for distance).

Mover example

Way back in the struct chapter we made something to hold the four variables we need to slowly move and wrap-around. This was it:

```
struct Mover {
  public float val; // the moving value
  public float spd; // amount to change val, each update
  public float min, max; // the limits for val
}
```

That was a nice way to group them. But the code hand-moved each variable, like `m1.val+=m1.spd;`. That would be way better as a function. Now that we know member functions, it would be even better as one of those.

If you remember, `val` goes up or down by `spd`, wrapping around when it hits min or max:

```
struct Mover {
  // the same 4 variables

  public bool doMove() { // returns true if it wrapped around
    bool wrapped=false;
    val+=spd;
    if(val>max) { val=min; wrapped=true; }
    if(val<min) { val=max; wrapped=true; }
    return wrapped;
  }
}
```

`doMove` is nothing special. The math is the same as previously. But it's written next to the variables, inside the function, making it a little easier to find. `mover1.doMove();` runs it. Of course, its purpose is to change `mover1.val`.

We usually write some more helpful member functions. Setting min and max at the same time feels natural, so we should write that:

```
  public void setRange(int low, int high) { // helpful setting function
    if(low>high) low=high; // we may as well check this
    min=low; max=high; // <-setting our min and max
  }
}
```

Here's our final code to fade from red to black, 20 times in a row:

```
Mover redMove;
int laps=0;

void Start() {
```

```
  redMove.setRange(0.2f, 1.0f);
  redMove.val=1.0f; // start at highest #
  redMove.spd=-0.02f; // small since total range is only 0.8
}

void Update() {
  if(laps<20) {
    if(redMove.doMove()) laps++;
    transform.getComponent<Renderer>().material.color=
      new Color(redMove.val,0,0);
}
```

It's supposed to feel as if we're asking the `redMove` variable to run it's own `doMove` function.

BoardSquare

A few chapters ago we made a simple class to hold data for one square in a game board. That particular one allowed the user to toggle between two colors (I called that selecting a space) and it showed which space you were "on" by making it a little smaller.

We can make those member functions:

```
class BoardSquare {
  public GameObject gg; // same as before
  public bool selected; // select to flip the color

  public toggleSelect() { // flips in on/off
    selected=!selected;
    Color cc=Color.blue;
    if(selected) cc=Color.red;
    gg.GetComponent<Renderer>().material.color = cc;
  }

  public highlight(bool isOn) { // show the current square
    float sz=1.0f; if(isOn) sz=0.8f; // highlight makes us a little smaller
    Vector3 sz = new Vector3(sz, sz, sz);
    gg.transform.localScale = sz;
  }
}
```

We can now change squares like this:

```
Board[oldCol][oldRow].highlight(false); // undo old
Board[col][row].highlight(true); // select the new one
```

It looks somewhat nice – a space in the board is turning its highlight on or off. The pop-up trick also works: `Board[col][row].` (a dot) causes the member functions, including highlight, to pop-up.

When someone presses space we'd use `Board[x][y].toggleSelect();`. That handles the coloring.

In both of these examples, 95% of the work was "hey, that should be a function". The last 5% was seeing that it may as well be a member function.

30.3 Calling your own member functions

Member functions can call other member functions. They use the same shortcut rule as variables – no dot, simply write the name. Here Dog's have a `d1.superDesc()` which uses their `desc()` function:

```
class Dog {
  public string name;  public int age;
  public string desc() { return "Name: "+name+", "+age+" years old"; }

  // new superDesc:
  public string superDesc() {
    string dw = desc(); // <- call "my" desc() function
    if(name=="dog" && age==0) dw="Basic Puppy";
    return = "<<<< "+ dw +" >>>>>";
  }
}
```

We call `desc()` and it means the `desc()` of the current Dog. In other words, `d1.superDesc()` knows it's also running `desc()` for `d1`.

Here's a completely fake function that uses two Dogs – the calling Dog and an extra input Dog. It shows how the rules work even for funny stuff:

```
class Dog {
  // ...
  public string doubleSuperDog(Dog other) {
    string w1 = "dog#1 "+superDesc(); // runs my superDesc
    string w2 = " dog#2 "+other.superDesc(); // runs its superDesc
    return w1 + w2;
  }
}
```

The first call to `superDesc()` knows to use ours, which knows to use our `desc()`. The second call, `other.superDesc()`, runs on the other dog, which runs `desc()` on the other dog. The computer remembers which dog you're on, even when you flip back-and-forth.

30.4 Built-in member functions

Most of the pre-made structs and classes have member functions. Now that we know how they work, we can enter variable-dot and look for them.

30.4.1 Unity member functions

`Vector3` has a `Set` member function – we can write `Vector3 pos; pos.Set(-7,2,0);`. It's merely a shortcut for hand-setting them. It looks like this:

```
struct Vector3 {
  public float x, y, z;

  public void Set(float new_x, float new_y, float new_z) {
    x=new_x; y=new_y; z=new_z;
  }
}
```

There's nothing special about `new_x`, that's just the name they picked.

Oddly, `Color` doesn't have a `Set` function or anything like it. They didn't write one.

I used `transform.Translate(1,0,0)` several chapters ago without explaining the dot. Now we know it's a member function. `transform` is a variable (you get it for free in Unity if your script is on a gameObject). When you type `transform`-dot, you can see field `position` and member function `Translate`.

If you have `GameObject g;`, typing `g`-dot shows a big list, including one named `SetActive(bool);`. That tells us we can use `g.SetActive(false);` to hide it (and again with true to un-hide).

Put another way, we knew there had to be a way to temporarily de-activate a Cube. There is, and it's a member function.

30.4.2 String member functions

I lied when I wrote that `string` was a basic variable type. You probably figured out that `string` is way more complicated than something like `int`, `float` or `bool`. `string` is really a built-in C# class with a bunch of special rules to make it act like a basic type.

But since it's really a class, it has member functions. As usual, we can type `w`-dot to find them. Lots of them are really fun:

- `w.Contains("a")` is true when `w` has an `'a'` in it. We know this is just an index loop. We've written it already, but it's still a nice shortcut.

- `w.StartsWith("abc");` and `w.EndsWith("abc")` are also simple loops, which we're already written.
- `string w2 = w.SubStr(2,4);` gives a sub-string – start from index 2 and take 4 letters (so positions 2, 3, 4 and 5). `w.Substr(2,2)` on `"catfish"` gives you `"tf"`.
- `string w2 = w.Insert(3,"ABC");` adds `"ABC"` just before index 3 (`"chicken"` would become `"chiABCcken"`) As usual, it returns a changed copy. You'd need to use `w=w.Insert(3,"ABC");` to change yourself.
- `string w3 = w.Remove(2,4);` gets rid of 4 characters starting at 2 (it returns the result.)

  A funny thing about this, and the ones before it, are that we know member functions can change themselves, but these don't. We prefer returning the changed version and not changing the original.
- `w.ToLower()` returns an all-lower case version of `w`. It's used in a clever trick for case-insensitive compares. `if(w.ToLower()=="done")` is true for `"done"`, or `"DONE"` or `"Done"` ....

Those are all simple loops, mostly things we're written before. But it's still nice to have them "in" the computer as easy-to-find member functions. Also, maybe, they used some tricks to run them a little faster than our versions.

30.4.3 List member functions

Back in the `List` chapter, we used `L.Add(6);` to grow `L` by one box. That's obviously a member function.

There are a few more:

- `L.Clear();` resets the list to size 0.
- `L.RemoveAt(0);` gets rid of the first item (it slides everything else down, using a loop, then subtracts 1 from the list size).

  It works for any index. `L.RemoveAt(L.Count-1);` gets rid of the last item, which doesn't need a slide-loop at all.
- `L.IndexOf("cat");` searches for that word and returns the index, or `-1` if it's not there. We've written this before. `L.Contains("cat");` is the same but worse – it only returns true or false.
- `L.GetRange(2,4));` is like substring. It returns a list of 4 items from `L` starting at index 2.

There are more than that, but that should give the idea.

30.5 `this`

Inside of a member function, you can use `this` to mean "me". For example you could write `this.name` instead of just `name`.

Times when you need it are rare and strange. Suppose there's a real global function taking a dog as input, which our member function needs to run on itself:

```
class Dog {
  public void dogFunc() {
    // oh no! I need to run dogUsingRealGlobalFunc with me as input:
    int n=dogUsingRealGlobalFunc(this);
    ...
  }
}

int dogUsingRealGlobalFunc(Dog d) { ... }
```

There's no good reason to have a global function like that – it would be a member function. But on big projects funny stuff can happen. Sometimes you're stuck using not-quite-right things.

You'll sometimes see a style where every member variable has `this` in front:

```
class thisUsingFunctions {
  public int a, b;
  // using extra this's for fun:
  string desc() { return this.a+" "+this.b; }

  void setup(int a, int b) { this.a=a; this.b=b; }
}
```

In the second function, using `this` let us re-use the names in the inputs. `this.a=a;` copies the input `a` into the member variable `a`.

30.6 Constructors

We've already seen functions like `makeDog("Rex",8)` which create and return a Dog. But languages like to make those official. We've seen it: `p1=new Vector3(1,0,9);` or `cc=new Color(0,0,1);`. Those are officially called *constructor*'s.

An example of a Dog constructor. This allows us to write `Dog d1 = new Dog("K-9",12);`:

```
class Dog {
  public string name;  public int age;
```

```
  public Dog(string nm, int howOld) { name=nm; age=howOld; }
}
```

The body of the function is normal, even boring. But the part in front looks strange. Special rules for constructors:

- The name of the function is the name of the class.
- Don't write the return type. Leave it blank. The return type is automatically an item of that class.
- Don't create the object or return it. The system does both of those things for you. Your only job is to set the fields.
- You can't run them using variable-dot. They only run using a `new`. In fact, every time you use `new`, it needs to find a matching constructor.
- Overloading is allowed. You can have multiple constructors with different inputs.

Here are several for a Cow struct. The let us call `new Cow()` or `new Cow("Lu-lu",7,1475)` or `new Cow("Alice")` and do various odd things:

```
struct Cow {
  public string name;
  public int age;
  public float wt;

  // with no inputs, create a standard cow:
  public Cow() { name="Bessy"; age=2; wt=1500; }

  // or provide all 3 inputs:
  public Cow(string nm, int years, float pounds) {
    name=nm; age=years; wt=pounds;
  }

  // this one is just silly, but legal:
  public Cow(string nm) {
    name = "Madam " + nm;
    age = Random.Range(1,10+1);
    if(nm.Length>5) wt=1000;
    else wt=1500;
  }
}
```

All three of them simply set the 3 Cow member variables. The first one replaces the old `new Cow()`. Our basic cow is now Bessy, 2, and 1500. The

second is the standard one that copies the inputs straight into the fields. The third shows that we can do anything a normal function would do, as long as we set all 3 fields.

Constructors are member functions in a strange way. The first thing they do is create a Dog or Cow. Then they run as member functions on that. When we use `name=nm;`, we're assigning to the thing that we're about to return.

Here's one more set of constructors for the old `FullName` class:

```
struct FullName {
  public string first, last;

  public FullName() { first=last=""; } // the do-nothing one

  public FullName(string fName, string lName) {
    if(lName=="") lName="(no last name)";
    first=fName; last=lName;
  }
```

We can use `f1=new FullName();` as usual, or `f1=new FullName("Cher","");`, who gets (no last name) for a last name.

Constructors are allowed to call other functions. It's common to have a shared `setup` function:

```
struct Cow {
  ...
  public Cow(string nm, int years, float pounds) { setup(nm, years, pounds); }
  public Cow(int years, float pounds) { setup("generic cow", years, pounds); }
  public Cow() { setup("",0,0); }

  public void setup(string w, int ag, float lbs) { name=w; age=ag; wt=lbs; }
}
```

Every constructor figures out what it wants to values to be, and calls `setup` as a quick way to assign them.

Horribly confusing `C#` constructor rules

`C#` has two very strange rules for constructors. One is about class/struct and we've seen it:

For a class, `new Dog("Spot",4)` allocates a real Dog, on the heap. Then it gets filled in by your code and a pointer is returned (automatically, without you writing anything). But for a struct, `new Cow("Bessy",4,1500)` creates a temporary Cow. It can be copied into a Cow variable.

It's the same funny rule as before.

The other strange rule – when you write a class, you get a free constructor. When you write Cow, you automatically get `new Cow()` that makes empty strings and zero's.

But if you write a single constructor of your own, the free one goes away. If you wanted it, you have to re-write it yourself. That's why `FullName` had it. There's no special reason for this rule – other languages do it differently.

30.7 Scope

This is probably obvious: the names of member functions are only for inside the class. It's the same rule as for member variables. It's fine to use the same member function names in different classes, or anywhere else outside the class.

This has three `doStuff`'s, which are fine since they're in different scopes:

```
struct A {
  public void doStuff() { print("A stuff"); }
}

struct B {
  public void doStuff(); { print("B stuff"); }
}

void doStuff() { print("script stuff");}
```

There's never going to be any confusion between `a1.doStuff()`, `b1.doStuff();` and just regular `doStuff();`.

30.8 Style: member vs. non-member

Member functions are just regular functions that we thought looked nicer using the variable-dot notation. They look best when there's one special item. `d1.copyFrom(d2)` is fine since `d1` is changing itself, using `d2`.

But compares look nicer as normal functions, since neither is special. Below shows checking for 2 equal dogs written both ways:

```
class Dog {
  public bool isEqual(Dog d) { return name==d.name && age==d.age; }
}

// as a non-member:
bool isEqual(Dog d1, Dog d2) { return d1.name==d2.name && d1.age==d2.age; }
```

The first one is `if(d1.isEqual(d2))`. The second is `if(isEqual(dog1, dog2))`.

`Vector3.Distance(v1, v2);` is another one like this. It could have been `v1.Distance(v2);`. But since neither point is more special than the other, the first way seems nicer.

30.9 Special struct in script rule

This is the last tricky rule. Consider a Cow defined inside one of our scripts:

```
public class cowUser : Monobehaviour {
  public int minCowWt;

  struct Cow {
    public string name;
    public float wt;

    public void set(string nm, string cowWt) {
      // this is not allowed to use minCowWt
    }
  }
```

There are 2 ways to look at `Cow`. The most obvious way is that every `Cow` must be created by a `cowUser`. A `Cow` would then automatically know about who made it and be able to use those variables.

That's called an *inner class*. Some languages use it, like Java, but not C#.

The other way to look at it is that `Cow` just happens to be written inside. But it's really a completely independent struct. Any other script could declare one. A `Cow` can't depend on belonging to a particular `cowUser`.

That's the way C# does it.

Chapter 31

Access modifiers

This chapter finally explains why we have to add `public` in front of all of our struct variables. If you remember, without the `public`, you can't use them, which seems like a crazy rule.

The actual rule is that everything is either `public` or `private`, and if you don't say, they're automatically `private`. But that doesn't explain why we have `private` in the first place.

This chapter is about how `private` variables work, why we'd ever want to use them, and a few similar tricks.

31.1 How `private` variables work

`private` really means that people *outside* of the class aren't allowed to use it. Member functions in the class are allowed to see all the fields. You can think of `private` as employees-only.

Here's an example class with two `private` variables:

```
class NumHider {
  private int n;
  int m; // m is also private, since we didn't write anything in front

  public void Set(int v1, int v2) {
    n=v1; m=v2; // legal. private doesn't apply to us
  }

  public getN() { return n; }
  public getM() { return m; }
}
```

This is not a good class, but it shows how it's possible to indirectly use private variables through member functions:

```
NumHider nh=new NumHider();
nh.n=5; nh.m=3; // ERRORS - private
nh.Set(5,3); // legal, the function sets m and n
if(nh.n<10) // ERROR
if(nh.getN()<10) // legal
```

We can never touch `n` and `m`, but we can change and read them through functions. Later we'll have an example where that's useful.

We can also have `private` member functions. We can't call them, but they aren't useless since other functions can. Here `fixNeg` is private, but `Set` uses it to fix the inputs:

```
class NumHider {
  public void Set(int v1, int v2) {
    n=v1; m=v2;
    fixNeg(); // <- we can call this, user can't
  }

  private void fixNeg() { if(n<0) n=0; if(m<0) m=0; }
}
```

The pop-up hides `private`'s from us, which makes them less annoying. Typing `nh`-dot will only show Set, getN and getM.

31.2 Classes as new types

For our classes so far, we started out knowing the variables. For example, `FullName` started with us wanting one string for first name and another for last name. Turning it into a class was just a nice way to group them. The member functions are merely helpful shortcuts. There's nothing wrong with that, but `private` wasn't made for those.

There's a different way to use classes. Sometimes we start with something we want to make. The variables are just a way to make it happen, and we really don't care about them. We only care about using the member functions.

`private` was invented for this idea – we can use it to say "don't think about the variables – the functions do what you need."

Here's an example. Suppose we want a random number roller that can remember the range, and never rolls the same number twice in a row. The interface (the public functions) can be like this:

```
class RollerNoRpt { // not done: names of public funcs only
  public void SetRange(int min, int max);
  public int Next(); // roll number, with no repeat
}
```

Just those functions are all we need for a useful class. We can say: `RollerNoRpt r1; r1.SetRange(5,10);`, and then use `int n=r1.Next();` to get 5-10's with no repeats.

Now that we like those functions as the interface, we have to write them, along with variables to make them work. We'll save the range, pre-adding 1 to the max, to account for how Random works. We'll have a variable saving the previous number:

```
class RollerNoRpt {
  private int min, maxp1; // if range is 1-6, we save (1,7)
  private int prevNum; // to avoid same num twice

  public void SetRange(int low, int high) {
    if(low>high) { int tmp=low; low=high; high=tmp; }
    min=low; maxp1=high+1;
    prevNum=low-1; // at the start, there is no previous roll
  }
```

So far, by hiding the variables we've made it impossible to get them backwards: If you call `SetRange(7,2)` by mistake, it fixes it.

`Next()` will use the "roll until it's not a repeat" loop from before, and the private helper function `simpleRoll`:

```
  public int Next() {
    if(min+1===maxp1) return min; // if range is 4-4, answer is always 4
    int nn = simpleRoll();
    while(nn==prevNum) nn=simpleRoll();
    prevNum=nn;
    return nn;
  }

  // used only by Next:
  private int simpleRoll() { return Random.Range(min,maxp1); }
}
```

That proves that we can do it, and it's some fun coding. But the important thing is how it's completely hidden. Users type `r1`-dot, see `SetRange` and `Next`, and that's all they need.

Another example, a little simpler: we want a class to check whether an x,y is inside a rectangular area (maybe for a 2D game).

We'll give options to set an area starting from either the lower-left corner, or from the center. For example `r1.SetFromCent(6,6,8,4)` puts the rectangle at (6,6), 8 wide and 4 high. Or `r1.SetFromLL(0,0,5,5)` puts the lower-left corner at (0,0), 5 wide and high. Some situations will prefer one or the other.

No matter how we set it, `r1.isIn(p1)` checks whether `p1` is inside. We don't care how it checks, or what variables it uses, as long as it works.

The plan is to store the real positions of all four corners. If we use SetFromCent with x=50 and 8 wide, we'll save 46 and 54 for x min and max:

```
class Rectangle {
  float xLeft, xRight, yLow, yHigh; // private

  public void SetFromCent(float xc, float yc, float wide, float high) {
    float halfWide=wide/2, halfHigh=high/2; // go 1/2-way in
    xLeft=xc-halfWide; xRight=xc+halfWide; // each direction
    yLow=yc-halfHigh; yHigh=yc+halfHigh;
  }

  public void SetFromLL(float xLLc, float yLLc, float wide, float high) {
    xLeft=xLLc; yLow=yLLc;  // copy these
    xRight=xLeft+wide; yHigh=yLow+high; // compute these
  }

  public bool isIn(float x, float y) {
    return x>=xLeft && x<=xRight && y>=yLow && y<=yHigh;
  }
}
```

That's all fun code. I especially like how `isIn` is merely two in-between compares this way. But the main thing is how it's hidden. Users only care about `SetFromCent`, `SetFromLL`, and `isIn`. They can't even see the variables.

This next example takes that idea a little further, making a 0-255 color class which purposely hides how Unity Colors use 0-1 (paint programs all use 0-255 for color levels. It's the range all artists know).

Here's the outline (it's a little fakey, but it's short and shows the idea):

```
class Color255 { // not done: names of public functions only
  public void Set(int red, int green, int blue) // use 0-255

  public void applyTo(Transform t) // apply the color to this transform
}
```

We can make bright orange with `c1.set(255,128,32);`. Then use `c1.applyTo(cat.transform);` to paint the cat orange. We never see a 0-1 value.

To store them, we'll use a real Unity `Color` variable, and a private helper conversion function:

```
class Color255 {
  private Color c; // a real 0-1 Unity color

  // this expects 0-255 values (which is why they are ints)
  public void set(int red, int green, int blue) {
    c.r=toF(red); c.g=toF(green); c.b=toF(blue);
    c.a=1; // 1=not transparent
  }

  private float toF(int n) { return n/255.0f; } // convert 0-255 into 0-1

  public void applyTo(Transform t) {
    t.GetComponent<Renderer>().material.color=c;
  }
}
```

`toF` is a typical `private` helper function. It's used by `Set` to convert each 0-255 into the real 0-1. We don't want the user to see it, since the entire point of this class is hiding how color is 0-1.

The class only has one variable, which is fine for classes like this.

31.3 Interface/Implementation idea

There are some concepts and terms that go with "use a class to make an idea." They aren't really technical terms, but people use them a lot, so it's nice to hear them, and some can be helpful.

We think of a class as divided into *Interface* and *Implementation*. Interface is the normal English meaning – the part you interact with. For a class, it's the `public` functions. Implementation is how it actually works.

Anyone using the class is a *client*. Clients use the Interface, and don't care about the Implementation.

This is really the same trick we've been using with functions – we only need to know the inputs and output, not exactly how it works. For a class, the inputs and outputs are what you can do with all the public functions.

Another word for the idea is *Information Hiding*. A slightly newer term is *Encapsulation* – the private implementation is encapsulated away from you.

Sometimes we call a class like this an *Object*. That's where the term Object Oriented Programming comes from.

Object is another way of saying we're absolutely not thinking of it as pile of variables.

Here's a list of some regular times people like to use `private` to break into Interface vs. Implementation:

- The class was written by an expert. You don't know how it works inside, and don't want to know. You're glad all that stuff is hidden with `private`.
- You wrote the class, but by tomorrow you'll forget what the variables stand for. If you don't make them private you'll set them directly instead of using the function ("I wrote this class. I know how things work!") and screw it up.
- You've got Interns and Jr. Programmers who don't understand that some variables have tricky rules, or go together with other variables. They've never seen the Interface/Implementation idea. Making only the Interface `public` automatically makes them do it the right way, without having to explain anything extra to them.
- We might want to rewrite it. If we think of a better way to roll dice we can rip out the private variables, and rewrite the guts using new, better ones. As long as `SetRange` and `Next` work the same, it's safe.

Sometimes it's the case that we know exactly what's inside of a class, but we still want to force ourselves to use the interface. Suppose we have a score that should also be displayed in a textBox. We could make a simple class to group them:

```
class Score {
  public int s;
  public GameObject label; // set this to previously created textBox
}
```

It's the user's job to remember to change the label when they change `s`. But at the very least we can make a helpful function for that:

```
class Score {
  public int s;
  public GameObject label;

  public void updateLabel() { // useful label-setting function
    label.GetComponent<Text>().text="Score: "+s;
  }
}
```

Users can add to the score with: `s1.s++;`, then run `s1.updateLabel();` to display it. But someone, somwwhere, will forget to run `updateLabel()`. The score will change, but we won't see it. To avoid that we need one command to do both things at once, which you have to use:

```
class Score {
  private int s; // <- users can't write s1.s++; any more
  private GameObject label; // <- they won't need to use this

  private void updateLabel() { // <- changed to private
    label.GetComponent<Text>().text="Score: "+s1;
  }

  // use these to change the score and auto-fix the label:
  public void ChangeBy(int amt) { s+=amt; updateLabel(); }
  public void Set(int value) { s=value; updateLabel(); }

  public int score() { return s; } // need this to read the score
}
```

To be nice, it's two commands. `s1.Set(5);` seems fine, at first, but `s1.ChangeBy(1);` is an easy way to add 1. The key is that both update the label. You can't get them out-of-synch.

Making `s` private caused a problem: we can't read the score any more. That's what `s1.score()` is for. We'll have to write ugly things like `if(s1.score()>21)`. But always having the score and label change together will be worth it.

The best way to assign the label is a bonus trick. We need to give it a textBox once, at the start, and won't change it again. A neat way to allow that is by putting it in the only constructor:

```
class Score {
  private int s; // <- users can't write s1.s+; any more
  private GameObject label;

  public Score(GameObject theLabel) {
    label=theLabel; // save the textBox link
    s=0; // we may as well do other stuff to make
    updateLabel(); // it look nice
  }
```

It's a neat trick since we need to use `new` anyway, and it's impossible to forget to supply the label:

```
public GameObject scoreTextBox; // dragged into Inspector slot
Score s1; // the class we just wrote
```

```
void Start() {
  s1 = new Score(scoreTextBox);
  // s1 is created, and has its label
```

This is one typical use of `private`. It's a way to say "I know that you know how to use the variables, and it would probably be fine. But, to be safe, please, please always call the functions instead".

31.3.1 Rewriting classes

A fun way to see the Interface/Implementation idea is to rewrite how a class works inside, without changing what it does.

The `Rectangle` class could be rewritten to secretly store the center and width/height (the old version stored the positions of the corners). This isn't super-exciting:

```
class Rectangle {
  private Vector2 center; // saving the center
  private Vector2 halfSize; // how far it goes in both directions x/y

  public void SetFromCent(float xc, float yc, float wide, float high) {
    center=new Vector2(xc, yc); // save the center they gave us,
    halfSize=new Vector2(wide/2, high/2); // and 1/2 the size. Easy
  }

  public void SetFromLL(float xLLc, float yLLc, float wide, float high) {
    halfSize=new Vector2(wide/2, high/2); // use the size and LLcorner
    center=new Vector2(xLLc, yLLc)+halfSize; // to find the center
  }

  public bool isIn(float x, float y) { // a lot more math
    return x>=center.x-halfSize.x && x<=center.x+halfSize.x &&
        y>=center.y-halfSize.y && y<=center.y+halfSize.y;
  }
}
```

I think the old way is better. But the point is that both ways – completely different variables – look and act exactly the same to users. That's pretty cool.

The random roller can have a much better rewrite. We probably want it to hit every number before repeating: for 1-6 it could be 4,3,6,1,5,2, then go again.

We can use the shuffle trick to do that. It starts with a hidden list of the scrambled numbers, walks through them as we ask with `Next()`, and rescrambles when we run out:

```
class RandNoRpt {
```

```
  private List<int> N; // all numbers we can roll, mixed up
  private int cur; // index of next number in N

  public void SetRange(int low, int high ) {
    if(low>high) { int tmp=low; low=high; high=tmp; }
    int rangeSize = high-low+1;
    // fill N with every number we can roll:
    N = new List<int>();
    for(int i=0;i<rangeSize;i++) N.Add(low+i);
    _shuffle();
  }

  private _shuffle() { // shuffles the list N
    cur=0; // restart at first item in new shuffle
    // this is copied from List chapter:
    for(int i=0; i<N.Count; i++) {
      int ii = Random.Range(0,N.Count);
      int tmp=N[i]; N[i]=N[ii]; N[ii]=tmp;
    }
  }

  int Next() {
    if(cur>=N.Count) { // past end, reset:
      int lastNum=N[N.Count-1]; // this can't be first in new one
      _shuffle();
      if(N[0]==lastNum) { int tmp=N[0]; N[0]=N[1]; N[1]=tmp; }
    }
    int ans=N[cur]; // shuffle or not, return next # in the list
    cur++;
    return ans;
  }
}
```

The pop-up still shows only `SetRange` and `Next()`. Users know nothing about a list and shuffling, which is great. Those are details they don't need to think about.

31.3.2 Accessors

Hiding a variable to make you use functions is so common that we have a name for the functions: we call them *accessors* or a "getter/setter" pair.

The Score class is a typical example. Everyone knows the class has an integer score variable. We're not trying to keep you from thinking about it. We merely want you use use `s1.Set(4)` to change it. A not-so-good side-effect is having to use `s1.score()` to read it. `Set(n)` and `score()` are its accessors – one to

"get" it, the other to Set it.

It's not great, but we can indirectly use the two functions to do anything `=` could do before:

```
// using the getter/setters:
s1.ChangeBy(1);
if(s1.score()>10) s1.Set(10);

// how we think of it:
s1.s++;
if(s1.s>10) s1.s=10;
```

We almost always use them to add extra rules about changing a variable – such as also updating the label. A common example is not allowing a variable to be negative. We put a check for that in the Set function:

```
class nUser {
  private int n;
  public int getN() { return n; }
  public void setN(int newVal) {
    if(newVal<0) newVal=0; // copies input into n,
    n=newVal;  // but fixes negative values
  }
}
```

It's a little awkward. `n1.setN(5)` replaces `n1.n=5`. Not too bad. But `n1.n--;` turns into `n1.SetN(n1.getN()-1);`. Ick, but it ensures `n` can't be negative.

There's one more trick getter/setter pairs allow. They can create fake variables. For example, this class stores the distance in feet, but can pretend it uses inches:

```
class FakeInchClass {
  public float feet;

  public float inches() { return feet*12; }
  public void setInches(float newInches) { feet=newInches/12; }
}
```

If you prefer to use this class with inches, you can. Use `n1.setInches(18)` and `n1.inches()` gives you back 18. For real it sets `feet` to 1.5, but so what? And if some other wise-guy sets `n1.feet=5` its fine – `n1.inches()` reads back 60. It works great.

We can do something similar with Rectangle. We already have a function that sets it using the lower-left corner. We can rename it and add one to compute the lower-left corner:

```
class Rectangle {
  public Vector2 center; // let people set this directly
  private Vector2 halfSize;

  public Vector2 getLowerLeft() { return center-halfSize; }
  public void setLowerLeft(int x, int y) { center=new Vector2(x,y)+halfSize; }
```

Combined, they allow us to examine and position Rectangles as if lower-left was the actual variable.

31.3.3 Special get/set

The getter/setter idea is so popular that C# has a super-special shortcut. The idea is this: getters and setters are substitutes for `x=n1.n;` (getter) and `n1.n=3;` (setter). What if we change `=`'s so they automatically run the appropriate get or set?

You still have to write both functions, in the class. But you don't have to use that messy syntax to call them. Even something tricky like `n1.n++;` will automatically run your getter for `n`, add 1, then run the setter.

In this legal C# code, `get`, `set` and `value` are magic words, specially for this trick. `n` can't be negative:

```
class NcantBeNegative {
  private int nn;

  public int n { // the fake variable is named n
    get { return nn; } // called for int x = n1.n;
    set { if(value<0) nn=0; else nn=value; } // called for n1.n=4;
  }
}
```

The funny syntax (it looks like a function named `n`, except with no parens for the input) is the special rule. `get` and `set` aren't real function names. `int x=n1.n;` magically calls `get` and `n1.n=3` magically calls `set` with `value` at 3.

The details aren't all that important. This trick is a style thing – if everyone else uses it, so should you. Hopefully it helps show the getter/setter concept.

It's also a fun example of language design. Older languages thought of this trick and rejected it. `n1.n=4;` being a secrete function call seemed too confusing. C# decided, why not?

Many different languages are that way – the features they do or don't have are opinions of the designers.

Chapter 32

Scripts as Classes

Now that we know how classes really work, we can see what's actually going on with Unity scripts. And now that we know about pointers, we can play with pointers to scripts.

These are examples of using Unity, but also the way regular programming works.

32.1 How scripts really work

This is probably obvious by now – scripts are classes. That's the reason they start with `public class testA`. Unity scripts have some added tricks, but in every way they are for-real classes.

Some of what this means:

- All of the script functions we wrote are really member functions. Even Start and Update.
- All of the global script variables are really member variables. When we have `void Start() { ticks=0; }` we're really setting a member variable in a member function.
- We didn't write `public` in front of everything, which means most things are private. That doesn't matter with just one script, since private/public is about what people *outside* of you can see.
- If we make some script functions `public`, other people can call them.
- Start and Update are private – they don't have `public` in front. So how is Unity able to automatically run them? It's using a trick we haven't seen yet.
- Like all classes, they don't exist until someone runs `new`. That's why our very first printing script needed to be dragged onto the camera. The

system runs `new textA()` automatically. Also like all classes, they can be `new`'d multiple times. That's why we can have several copies, or can accidentally double-drag 2 onto the same Cube.

- Two copies of a script have two different sets of variables. It's the same rule as `Dog d1` and `d2` having different names and ages.
- As a check, you can try the "`this`" trick: in a script, type `this`-dot. Your "globals" (really your member variables) and functions will appear in the pop-up, like any other class.

To be complete, the `: Monobehaviour` in a script definition is a standard programming rule called Inheritance, which we'll see much later. It's the way Unity knows a class can go onto a Cube, and might have a special Start and Update. It tells the system that a class is a "script".

That's also how our scripts for free get variables `transform`, `gameObject`, and `GetComponent`. They come with Monobehaviour.

32.2 Pointers to scripts

Since scripts are classes, we can have pointers to them. Suppose we have a script named `testA` on a Cube. Any other Cube can declare: `public testA ss;`, drag in the first Cube, and have a direct link to its script.

Even cooler than that, if all we have is a link to a gameObject, we can use GetComponent to find its script. The idea is: `GetComponent<Renderer>()` finds the Renderer. Renderer is a class. Our scripts are classes. So `c1.GetComponent<testA>()` to find the script on `c1`.

32.2.1 Direct script pointers

Here's a simple example of one main script using pointers to scripts on two other Cubes.

This first class looks like a hybrid between a unity script and a normal class. It goes on gameObjects, but it doesn't have a Start or Update. That's legal, but means it will never do anything. But it has `public` member functions. This class waits around for people to call its functions:

```
class cubeScriptC : MonoBehaviour {
  public void turnRed() { changeCol( new Color(1,0,0) ); }
  public void turnYellow() { changeCol( new Color(1,1,0) ); }

  public void teleport() {
    Vector3 pos;
    pos.z=0
    pos.x=Random.Range(-7.0f, 7.0f);
```

```
    pos.y=Random.Range(-4.0f, 4.0f);
    transform.position=pos;
  }

  // private helper function:
  void changeCol(Color c) { GetComponent<Renderer>().material.color=c; }
}
```

If we drag this onto a Cube, it's as if we gave it 3 new commands: `turnRed()`, `turnYellow()` and `teleport()`.

This next script would go on some other object. It can reach out to 2 Cubes and run those member functions:

```
class testC : MonoBehaviour {
   public cubeScriptC c1, c2; // <- links to the 1st script

  void Start() {
    c1.turnRed(); // turn 1 red
    c2.turnYellow(); // turn the other yellow
  }

  void Update() {
    if(Random.Range(0,200)==1) c2.teleport();
  }
}
```

`public cubeScriptC c1, c2;` is a normal variable declaration, for a class we made. It's the same as `Dog d1, d2;`. The magic is how Unity allows us to drag a Cube into them. If it has `cubeScriptC` on it, Unity hooks up a pointer.

In Start, `c1.turnRed();` is a normal member function call. It runs `turnRed()` on `c1`.

As a check, we can type `c1`-dot. It pops up member functions `turnRed`, `turnYellow`, and `teleport`, just like a real class.

32.2.2 GetComponent to find scripts

We can do the same trick with cubes we create using Instantiate. If it has a script, we can find it with `newThing.GetComponent<scriptName>()`.

Assume we have a Cube prefab, with `cubeScriptC` on it. This script creates 5, grabs the script on each, and uses it to set up them up:

```
class spawnTestA : MonoBehaviour {
  GameObject prefabC; // drag in a prefab with cubeScriptC

  void Start() {
```

```
    for(int i=0;i<5;i++) {
      GameObect gg=Instantiate(prefabC); // standard instantiate
      cubeScriptC cc=gg.GetComponent<cubeScriptC>(); // <- get the script
      cc.teleport();
      cc.turnYellow();
    }
  }
}
```

32.2.3 Multi-script ball-dropping game

This section will make a little game about dropping balls on targets, using three scripts. Here's the plan:

- There will be three classes: `player`, `flyingBall` and `target`. In Unity terms, we'll write three scripts.
- The player will be a cube that we can steer along the top, dropping balls (the whole thing will be from a front view, so the balls will just fall.) Same as always, the player-Cube will have the only copy of our "main" script, `player`.
- We'll make 4 target Cubes. Just regular Cubes with the `target` class on them. Position them anywhere interesting, with z=0 (so the balls we drop can hit them.) We'll be trying to destroy them.
- The ball should be a prefab with a Rigidbody added (so it falls) and the `flyingBall` script on it. The player creates one, with Instantiate, for each ball drop.
- When the player creates a fresh ball to drop, it will find the `flyingBall` class on it, reach over, and set some member variables. When a ball hits something, it will try to find the `target` script on what it hit, and call a member function to make the target react.

The player is mostly old code. This puts us near the top and makes the arrow keys move us back-and-forth. Update calls `handleShoot` when we press the space key, which I'll write later:

```
class Player : MonoBehaviour {
  Vector3 pos;
  float nextDropTime=0; // game seconds. can't shoot if < this

  // start in upper-center:
  void Start() { pos = new Vector3(0,4.5f,0); transform.position = pos; }

  void Update() {
    if(Time.time>nextDropTime && Input.GetKeyDown(KeyCode.Space)) {
```

```
      nextDropTime=Time.time+0.5f; // 1/2 second until next shot
      handleShoot();
    }
    handleMove();
  }

  void handleMove() {
    int mv=0;
    if(Input.GetKey(KeyCode.LeftArrow)) mv=-1;
    if(Input.GetKey(KeyCode.RightArrow)) mv=+1;
    if(mv!=0) {
      pos.x=Mathf.Clamp(pos.x+0.1f*mv , -7, 7);
      transform.position = pos;
    }
}
```

The only thing interesting so far is how `nextDropTime` uses the `Time.time` trick to force a half-second delay between shooting.

The `handleShoot` function uses `Instantiate` to create a ball, then grabs the script and sets 2 public variables. For simplicity, I'm placing the new ball below the player-Cube and letting it drop. The player's shooting function:

```
  public GameObject ballPF; // drag in prefab ball with flyingBall script

  void handleShoot() {
    GameObject ss = Instantiate(ballPF);
    Vector3 pos = transform.position; pos.y-=1.0f; // just below us
    ss.transform.position=pos; // place bullet below us

    // get a pointer to our bullet's flyingBall variable/script:
    flyingBall bs = ss.GetComponent<flyingBall>();
    bs.hurtAmt=Random.Range(2,5+1); // set two variables
    bs.endTime = Time.time+4.0f; // on the new ball
  }
}
```

On to the `flyingBall` script. We want the balls to live for a few seconds, then wink out of existence. When the player set `bs.endTime = Time.time+4.0f;`, that ball is being given 4 seconds to live. Update on the balls checks that time:

```
class flyingBall : MonoBehaviour {
  public float endTime; // set when made
  public int hurtAmt; // set when made. How much we hurt the target

  void Update() {
    if(Time.time>endTime) Destroy(gameObject);
```

```
  }
// not done
```

Each ball has its own copy, with its own `endTime`. We can safely have multiple balls with multiple wink-out times.

The rest of the plan was for the balls to hurt targets when they hit one. We'll use `OnCollisionEnter` for that. We used it way, way back, so you may not remember it. It runs when we hit something:

```
  void OnCollisionEnter(Collision cc) { // in flyingBall
    // try to get pointer to the target script on what we hit:
    target trg = cc.gameObject.GetComponent<target>();
    if(trg==null) return; // what did we hit? Not a target

    // use target member function (not written yet):
    trg.getHit(hurtAmt);
    Destroy(gameObject); // make us vanish
  }
}
```

Those first two lines are a common trick. Only the target Cubes have the `target` script on them. So, we can check for a target by trying to get that script. Not finding it isn't an error – as usual it simply returns `null`.

If we hit a target, we reach over to it and run its `getHurt` member function (which we'll write, below). Recall `hurtAmt` was originally set when the player dropped the ball. It's worked its way to finally lowering a target's health.

The `target` script mostly sits around waiting for someone to call its `getHit` function. It should wiggle a little, and possibly die:

```
class target : MonoBehaviour {
  private int health=10; // hits subtract, until <=0
  private bool isDying=false;
  private int wiggle=0; // # of frame to wiggle (set after a hit)

  public void getHit(int amount) { // <-balls call this
    if(isDying) return; // can't kill it twice
    health-=amount;
    wiggle+=15+amount*5; // update uses this
    if(health<=0) beginDie();
  }
// not done
```

`beginDie` is a typical private function, to make `getHit` shorter:

```
  private void beginDie() {
```

```
    isDying=true; // Update looks at this
    GetComponent<Renderer>().material.color = Color.Red;
  }
```

The rest is handling wiggling and shrinking when we're dead. The style should be familiar – an Update calling a function for each task:

```
  void Update() {
    //if(Input.GetKeyDown("a")) wiggle+=10; // testing
    //if(Input.GetKeyDown("x")) getHit(5); // testing

    // getHit will set these, this is where we use them:
    if(wiggle>0) handleWiggle();
    if(isDying) handleDie();
  }

  private void handleWiggle() {
      // move a little bit, stay in bounds, reduce wiggle counter:
      Vector3 pos = transform.position;
      pos.x += Random.Range(-0.03f, 0.03f);
      pos.x = Mathf.Clamp(pos.x, -7, 7);
      pos.y += Random.Range(-0.03f, 0.03f);
      pos.y = Mathf.Clamp(pos.x, -3, 4);
      transform.position = pos;
      wiggle--;
  }

  private void handleDie() {
    // get a little smaller, die at size 0:
    float sz = transform.localScale.x - 0.02f;
    if(sz<=0) { Destroy(gameObject); return; }
    Vector3 sc = new Vector3(sz, sz, sz);
    transform.localScale = sc;
  }
}
```

Altogether, we've got the player class talking to the ball class, and the ball class talking to the target class. Only one time each, but the ability to find another class and set a variable or run a function is what makes this work.

Testing the end of a sequence can often be a challenge – testing how the targets wiggle and die without having to play the game each time. That's what the commented-out lines are for. They totally cheat, skipping past moving and dropping and colliding. Pressing A lets us see a wiggle, and X tests a hit and possible death.

I'm 99% sure that most game cheats and hidden power-ups were put in for testing, and left in by mistake.

32.2.4 Hand-running fake updates

That dropping balls example is a typical loose set-up. Everyone does their thing, talking at various times, with no one in charge. That's great when it works, but sometimes you need one master program to run things. That's a very traditional approach, which we can set-up in Unity.

Suppose we want 8 balls which occasionally change color. We could write one script for that, with an Update. Each ball runs itself. Or, we could get rid of the Update. Each ball waits for instructions from a master program. It's the same idea as the turn-red-turn-yellow example, but more systematic.

The class for the balls is merely useful color-changing, waiting for someone to call them:

```
class colorBall : MonoBehaviour {
  float nextChange; // time for next color change
  Color baseCol; // color will flicker around this

  public bool isReady() { return Time.time>=nextChange; }
  public void setNextTime() { nextChange = Time.time + Random.Range(0.2f, 0.6f); }

  public void setStartCol() {
    baseCol.r = Random.Range(0.0f, 1.0f);
    baseCol.g = Random.Range(0.0f, 1.0f);
    baseCol.b = Random.Range(0.0f, 1.0f);
  }

  public void nextCol() {
    Color cc=baseCol;
    // give slight tweak:
    baseCol.r += Random.Range(-0.2f, 0.2f);
    baseCol.g += Random.Range(-0.2f, 0.2f);
    baseCol.b += Random.Range(-0.2f, 0.2f);
    GetComponent<Renderer>().material.color=cc;
  }
}
```

It uses some Unity built-ins (checking the time, set color), but mostly looks like a normal non-Unity class.

The main program creates 8 balls from a prefab (which has the `colorBall` script in it), saves links, and hand-runs the functions:

```
class player : MonoBehaviour {
  public GameObject ballPF; // drag prefab ball with colorBall here
```

```
  List<colorBall> C; // will point to everyone's scripts

  void Start() {
    C=new List<colorBall>();
    for(int i=0; i<8; i++) {
      GameObject gg = Instantiate(ballPF);
      Vector3 pos; pos.z=0; // put at random location
      pos.x=Random.Range(-6.0f, 6.0f);
      pos.y=Random.Range(-3.0f, 3.0f);
      gg.transform.position = pos;
      colorBall cc=gg.GetComponent<colorBall>(); // get script
      cc.setStartCol(); // calls that ball's setup function
      C.Add(cc); // save the link in our list
    }
  }

  void Update() {
    for(int i=0;i<C.Count;i++) {
      if(C[i].isReady()) {
        C[i].setNextTime();
        C[i].nextCol();
      }
    }
  }
}
```

Start is nothing special – we've made and placed 8 things before. Update is the interesting part. Notice how it looks like using the interface of a normal class – we're using only functions, and they handle the details.

I feel like `C[i].nextCol();` would have been a complicated line a few chapters ago. Hopefully by now it's fine – find `C[i]` and run its `nextCol()` function.

The advantage of this approach is being able to more easily control events. Suppose we can pause, and also the player can freeze one ball

```
  void Update() {
    for(int i=0; i<C.Count; i++) {
      if(isPaused) continue; // paused means no color changes
      if(i==currentPlayerBall) continue; // don't change ball player is on
      if(C[i].isReady()) {
        C[i].setNextTime();
        C[i].nextCol();
      }
    }
  }
}
```

The other advantage is seeing everything that happens. It's all in the master Update function. If we skip one particular Cube sometimes, the code for it is right there.

We use these tricks in all sorts of programs. Classes that sit there waiting for calls are great. The `target` script has an Update with wiggle&die, but that's merely animation – it's a sit-and-wait class. "Fire-and-forget" objects are also useful in lots of situations, as long as they are mostly independent. The `flyingBall` class needed us to give starting values, but was fine running itself after that.

After seeing a few examples of each, you can usually figure out the idea behind various arrangements of classes, and how they should work together.

Chapter 33

Nested classes, static

This section is just two funny rules about classes jammed together. The enum/class section isn't really new. It's more funny stuff that using `private` can do. And remember, you never need to use private. But `static` is a new rule, used to make real globals.

33.1 Nested `enum`'s, classes and `public`

In a class, `public` and `private` also apply to `enum`'s or any other classes you write. So far it didn't matter, since we only used them from inside of that class. But outsiders can only use the ones you make `public`.

An example, this makes one public and private of each:

```
public class Player : MonoBehaviour {

  public enum statusT { resting, walking, running };
  private enum hatT { none, cowboy, bowler, baseball, bonnet };

  public class Dog { public string name; public int age; }
  class Lion { public string noise; public float tailLength; }
  // Note: Lion is private, since we didn't write anything
```

Inside of `Player`, we can use all 4 of these (obviously, since we've been doing it for a while). Outside, we can't see `hatT` or `Lion`, since they're `private`.

We'd make them `private` for roughly the usual reason: they're "helper" classes and enums. No one outside of the class should be using them, or should even see them in the pop-up.

To use the `public` ones outside of player, we're required to add player-dot in front. Here's an example of a new class `Kitten` using them:

```
public class Kitten : MonoBehaviour {
```

```
  Player.Dog d1; // the Dog inside of Player
  Player.statusT s1; // same
  //Player.Lion L1; // ERROR -- Lion is private in Player

  void Start() {
    d1 = new Player.Dog(); // <- also here
    s1 = Player.statusT.resting; // <- yow, and here

    d1.age=5; // whew! this is normal
  }
```

The reason should make sense. We could have one `Dog` in a file by itself, and another inside of `Player`, and another inside of `testA`. Three different Dog classes. Their names would be `Dog`, and `Player.Dog`, and `testA.Dog`. It's the usual namespace situation.

And we like it that way. If `testA` needs its own personal `Dog` class, it should be able to have one.

Obviously, you can skip the first part from inside the class. Inside of `Player` we can use `Dog` as a shortcut for the real name, `Player.Dog`.

You can't have a `public` variable of a private type. It's easy to do by accident:

```
public class Player : MonoBehaviour {
  // Lion class is private:
  class Lion { public string noise; public float tailLength; }
  public Lion L1; // ERROR
```

You get a really cool error: *Inconsistent accessibility: field type 'Player.Lion' is less accessible than field 'Player.L1'.*

The problem is obvious, but takes some thought. Anyone outside the class trying to use `L1` will need to look up what `Lion`'s are, which is private, so they won't find it. They can't possible use `L1` if they aren't allowed to read its type.

When you get this error, it almost always means that you wanted to make the class public, and simply forgot to.

You might remember the special way to put a class in the Inspector from way back:

```
// dog that can be in the Inspector:
[System.Serializable]
public Dog { public string name; public int age; }

public Dog d1; // has an Inspector slot
```

Now we know why we needed the extra `public` in front of the class. We can't make `d1` public without having the class itself be public.

And finally, as usual, if you ever have any problems caused by private classes or private enums, make everything public. You never actually need to make something private.

33.2 Static / namespaces

`static` is a special word that allows us to make true global variables and functions. Write something inside of a class and add `static` in front. Now it's a true global, using that class as a namespace.

33.2.1 static functions

The trick is most common with functions, First here's an example making a global function, inside of `Dog`:

```
class Dog() {

  // normal function:
  public static bool isEqual(Dog d1, Dog d2) {
    return d1.age==d2.age && d1.name==d2.name;
  }
}
```

Because of the `static`, `isEqual` is a normal function. You can't call it with a Dog in front, and wouldn't want to. But anyone, anywhere could call it like `if( Dog.isEqual(pet1, pet2) )`.
The `Dog` in front is just a namespace, telling us how to find `isEqual`.

This is a pretty typical example. We wanted to write a regular function to compare two Dogs. Then we thought it would be nice to group it with the dogs, so we put it inside with `static`. An advantage is we can now write `Dog`-dot and find it.

I think the biggest confusion about `static` is that it goes in front like public/private, but it's completely different. Public/private don't change how something works, just who can use it. But `static` is a change in how it works.

Our old function `Vector3.Distance(v1, v2)` uses this trick. It's a normal function telling you how far apart the two input points are. It's put inside the `Vector3` namespace to make it easier to find:

```
public struct Vector {
  public float x, y, z;
```

```
  // static in front means it's not a member function:
  public static float Distance(Vector A, Vector B) {
    // standard 3D pythagorean theorem:
    float dx=B.x-A.x, dy=B.y-A.y; dz=B.z-A.z;
    return Mathf.Sqrt(dx*dx+dy*dy+dz*dz);
  }
}
```

The `static` is telling us we don't use `v1.Distance(v2, v3)`. It's not a member function. Notice how the inside is all `A.x` and `B.y`. There's no direct use of "our" member variables, since we're a normal function.

Here's an example writing `desc` both ways:

```
public class Dog {
  // member function:
  public string desc1() { return name+" "+age; }

  // non-member function:
  public static string desc2(Dog d) { return d.name+" "+d.age; }
}
```

Notice the first has no dog input, since a dog calls it: `d1.desc1();`. The second has an input since it's a normal function: `Dog.desc2(d1);`.

33.2.2 Fake classes for namespaces

We've seen the idea of a namespace as a folder for global functions. C# fakes this using the static rule and a do-nothing class. Here, `Rnd` is a folder for some dice-rolling functions:

```
using UnityEngine;
using System.Collections;

public class Rnd { // this is not a class
  public static bool pct(int chance) { return Random.Range(1,101)<=chance; }

  public static int roll_xDy(int numDice, int diceSides) {
    int sum=0;
    for(int i=0;i<numDice;i++) sum+=Random.Range(1,diceSides+1);
    return sum;
  }
}
```

We now have global functions `pct` and `roll_xDy`, who's full names are `Rnd.pct` and `Rnd.roll_xDy`. Technically `Rnd` is a class, but not really. We're abusing the rules to use it as a folder.

It's an idiom. For people who haven't seen the trick, a class with no member variables is horribly confusing. But long-time users don't even see the word class anymore. This is the C# way of making a namespace.

Unity has several of these. `Mathf` holds functions like `Mathf.Sqrt(8)` (square root) and trig functions and so on. Mathf is clearly a folder – a namespace. But the actual way it's written is a struct with no fields and all static functions.

33.2.3 static variables

`static` in front of a variable stops it from being a field, and turns it into a regular global. This makes two globals:

```
class gameStats { // this is not a class
  public static int currentLevel;
  public static int livesLeft;
}
```

Anyone, anywhere can use `gameStats.livesLeft--;` or `if(gameStats.currentLevel==5)`. As before, `gameStats` isn't really a class.

To compare, if they didn't have `static`, we'd start with no variables. And declaring `gs1`, `gs2`, and `gs3`, would create 3 sets. Adding `static` means we start with them, and can never make any more. They're regular declared variables.

Unity has global variables using this trick. The global for the time in seconds since it started, is `Time.time`. It turns out `time` is a static float, and `Time` is a class abused to be a namespace.

There's rarely a reason to mix `static`'s into a real class, but you can do it. Suppose I want a variable holding the total number of Dog's ever made. I may as well keep it in the Dog class, as a static:

```
class Dog {
  public string name; // these are the fields
  public int age; // in each Dog

  public static int count; // global variable
}
```

This gives us one global, named `Dog.count`. Each Dog we declare has name and age, as usual.

It's common enough to put global functions and variables in the same fake class. For example:

```
class gameStats { // this is not a class
  public static int currentLevel;
```

```
  public static int livesLeft;

  public void reset() { currentLevel=1; livesLeft=3; }
}
```

`gameStats.reset()` is a global function, changing global variables.

33.2.4 File of classes

I've been defining classes, and everything else, inside of scripts since that was the only place we had.

If multiple scripts in a program wanted to use a class, it's fine to put it in a file by itself. We'd create this file the normal way, and anyone could declare things from it:

```
using UnityEngine;
using System.Collections;

public class Dog { public string name; public int age; }

public class FullName {
  public string first, last;
  public string desc() { ... }
  ...
  }

  public enum catBreed {persian, siamese, tabby, AmerShorthair};
}
```

Notice how pretty this is. It's our normal definitions, without any of the extra stuff from a script.

Also notice how these are "naked". They're not wrapped in a namespace or anything. `Dog`'s entire name, for anyone, is just `Dog`. Any script could declare `Dog d;` or use `catBreed cb1=catBreed.tabby;`.

We could throw a namespace around them. `class Pet { ... }` around the whole thing. Then we'd have `Pet.Dog d1` and so on. The advantage is the usual scope help: with these hidden inside of `Pet`, another part of the program can re-use the names.

Chapter 34

Reference parms

This section is about a new rule where a function can change its inputs for real. We can partly do that now with pointers – `fixName(dog1);` can adjust `dog1` – but this lets us do it with ints, strings, floats, and structs.

We try not to use this rule if there's any better way. But it's good for the times you need it, and it's a basic computer science concept. The normal way, which copies inputs, is named **call-by-value**. This new way is **call-by-reference**.

34.1 `ref` rules

This rounds a `float` to the nearest value, changing it in place:

```
void roundMe(ref float n) { // <- ref is new rule
  int nearest = (int)(n+0.5f); // standard rounding trick
  n=nearest;
}
```

Notice how all it does is change `n` into the correct value. We've seen functions like this before. It works because `n` is a solid link to the variable you gave it:

```
float q=3.6f;
roundMe(ref q); // q is 4
```

Here are the rules for call-by-reference (what you type, mixed in with how it works):

- Write `ref` in the function definition, before the parameter type.
- You also have to write `ref` in front when you call it. That's to remind people that it's a reference parameter.

- The `ref` applies to one thing, but you can use it more than once. Exs: `void A(ref int n, ref float ff)` (both) or `void B(int q, ref string w);` (only `w`).

- Inside the function, the `ref` parameter isn't a new variable box with a copy of the input. It uses the box you gave it. It's like a super-pointer.

- You have to call it with a variable, which must be the correct type. This is a natural consequence of sharing a box. `ref float n` takes a box holding a float.

  For examples, `roundMe(ref 12.3f)` is an error – not a box. `roundMe(ref x+1.0f);` is an error – also not a box. `int n=4; roundMe(ref n)` is an errors – not a float box. But `roundMe(ref c1.wt);` and `roundMe(ref A[2]);` are fine.

- Changing a reference parameter changes the original variable.

In `float q=3.6f; roundMe(ref q);` I like to think that we're giving it `q`. Without the `ref`, we're merely giving it 3.6. With `ref` we're handing over the entire box.

34.2 More `ref` examples

Here's another version of the old Clamp function which changes the input into the correct range, using call-by-ref:

```
void clampMe(ref int num, int low, int high) {
  if(num<low) num=low;
  else if(num>high) num=high;
}
```

Notice how all it does is change `num`. Running it looks like this:

```
int x=12;
clampMe(ref x, 1, 10); // x is lowered to 10, for real

int z=8;
clampMe(ref z, 50, 60); // z is raised to 50
clampMe(ref z, 1, 100); // no change. z still 50
```

Here's moveTowards rewrite. It pushes the input float towards the target, but not past:

```
void moveMeBy(ref float num, float target, float mvAmt) {
  // get there in one step?
  if(Mathf.Abs(target-num)<=mvAmt) { num=target; return; }
  // go towards target, in whichever direction:
```

```
  float dir = Mathf.Sign(target-num); // +1 or -1
  num += mvAmt*dir;
}
```

`moveMeBy(ref n, 10, 0.1f);` pushes `n` 0.1 closer to 10, stopping on it.

Those three are not useful functions. At first, `roundMe(ref x);` seems better than the hacky `x=round(x);`. But a return value let's us do more:

```
float x2=round(x); // using a normal round, which returns the answer
z=round(x+y);
if(round(x)>round(y)) ...
```

A swap is an actually useful call-by-ref function. We've seen the 3-step swap dance. Now we can put it into a function:

```
void swap(ref int a, ref int b) {
  int tmp=a; a=b; b=tmp; // standard 3-part swap dance
}
```

Notice how both have the `ref` in front, since both need to change. A sample use:

```
if(low>high) // switch to put them in order:
  swap(ref low, ref high);
```

Another real one I sometimes use is for computing a min and max. If it needs to, it adjust one of them to contain the new value:

```
void growRange(ref float min, ref float max, float newNum) {
  if(newNum<min) min=newNum;
  if(newNum>max) max=newNum;
}
```

It doesn't seem like much, but it can help find the min and max of a list:

```
int smallest=A[0], largest=A[0];
for(int i=1; i<A.Count; i++)
  // grow to fit A[i] in the range:
  growRange(ref smallest, ref largest, A[i]);
```

If saves me from writing 2 `if`'s, has a descriptive name, and there's no way to write it without using call-by-ref.

34.3 Output parameters

We can use the call-by-ref trick to have a function return 2 things. We'll give the function 2 empty boxes, asking it to put the answers in them.

This function attempts to break a string into 2 words if it has a comma. We can't return 2 things, but we can fake it with 2 "output" parameters:

```
void commaSplit(string w, ref string w1, ref string w2) {
  int commPos = w.IndexOf(',');
  // if no comma, we only have 1 word:
  if(commaPos<0) { w1=w; w2=""; return; }
  w1=w.Substring(0,commaPos); // before comma
  w2=w.Substring(commaPos+1); // after comma
}
```

In our minds, this returns `w1` and `w2`. A sample run:

```
string animal="", noise="";
commaSplit("milk cow,moo-moo", ref animal, ref noise);
// animal is "milk cow", noise is "moo-moo"
```

`commaSplit` feels as if it has 1 input, and 2 slots for the outputs.

Suppose we often want to convert floats like 5.32 into a 5 and a 0.32. We could write a "return-by-reference" function for that. In our minds it has 1 real input, and 2 outputs:

```
void wholeAndFraction(float num, ref int whole, ref float fraction) {
  whole = (int)(num); // ex: (int)5.8 is 5
  fraction = num-whole; // ex: 5.8-5 is 0.8
}
```

We call it by making 2 spots for the results and calling with the as the last 2 inputs:

```
int n1; float f1;
wholeAndFraction(9.34f, out n1, out f1);
// n1 is 9, f1 is 0.34
```

34.3.1 `out` shortcut

As we've seen, there are 2 main ways to use call-by-reference. The first is when we give it a box to possibly adjust. The function will read from it, and might change it or might not. The second is giving blank boxes for the answers. The function has no reason to read from them, and will definitely fill them all with the answers.

C# decided to make an extra rule for that second way: `out` for "output parameter". It's the same as `ref` except with more error messages (you can't read from them, and you have to assign to them).

Here's `commaSplit` rewritten with `out`:

```
void commaSplit(string w, out string w1, out string w2) { // <-out
  int commPos = w.IndexOf(',');
  // if no comma, we only have 1 word:
  if(commaPos<0) { w1=w; w2=""; return; }
  w1=w.Substring(0,commaPos); // before comma
  w2=w.Substring(commaPos+1); // after comma
}
```

It's the exact same, except for the word `out`.

It's another fun example of language design. On the one hand, `out` is a tiny change from `ref`, and you don't need it, and it's another special word to remember. On the other hand, why not build the idea of output parameters into the language?

It turns out that C# is the only language to that chose to add it. In fact, Java decided the whole call-by-reference thing was so rare and hacky that Java doesn't have even `ref`.

34.4 mixing real return and `ref` parms

We often have functions that either give an answer, or they don't work. Often they can return -1 or null, but not always. Sometimes we like to have them return a `bool` saying it they worked, and give back the real answer in a reference parm.

For example, `int.TryParse` is a C# built-in that attempts to convert `w` into an int, or else returns false:

```
string w; // could be "12", or could be "cat"
int num;
bool gotNum = int.TryParse(w, out num);

if(gotNum) print("number is "+num);
else print("failed. w wasn't a number");
```

For `"37"` it will return true, and `num` will be 37. For `"3cat"` it returns false and `num`'s value won't matter.

The more common way to use it looks very strange at first:

```
if(int.TryParse(w, out n)) print("number is "+n);
else print("can't read it");
```

Normally, functions inside of an `if` shouldn't do anything except return true or false. But in this case, it seems fine to have it also fill `n`.

We can rewrite `commaSplit` to use this style. It returns false if there isn't a comma:

```
void commaSplit(string w, ref string w1, ref string w2) {
  int commPos = w.IndexOf(',');
  // if no comma, it didn't work:
  if(commaPos<0) { w1=w; w2=""; return false; }
  w1=w.Substring(0,commaPos);
  w2=w.Substring(commaPos+1);
  return true; // got both words
}
```

The new way to use it:

```
string animal="", noise="";

if(commaSplit(W[0], ref animal, ref noise)) {
  print("The "+animal+" says "+noise);
}
else print("invalid animal/noise pair");
```

34.5 Pointers and References

We can already change classes from inside of a function, so it seems like we wouldn't also need call-by-reference for them. But sometimes we do.

This function to swap two goats (assume Goat is a class) does nothing:

```
void swap(Goat a, Goat b) { Goat tmp=a; a=b; b=temp; } // broken swap
```

Here's a picture after we call `swap(g1, g2);`:

```
// start function call:
  main                     swap
g1 -> [ name: A  age: 4]  <- a
g2 -> [ name: B  age: 7]  <- b
```

If we change `a.age`, we're changing the real age. But making `a` point to `b` is simply moving a local variable. We can't change where `g1` points, only the contents of what it points to.

Adding `ref` in front of `a` and `b` fixes the problem:

```
// working goat swap
void swap(ref Goat a, ref Goat b) { Goat tmp=a; a=b; b=temp; }

swap(ref g1, ref g2);
```

Now changing `a` is also changing `g1`.

This next one is simpler, but somewhat made-up. Suppose we want a function to blank out a goat, which works for `null` goats. This version doesn't work:

```
void blankGoat(Goat g) { // non-working
  if(g==null) g=new Goat(); // <-original goat is still null
  g.name="test goat"; g.age=0; // <- this part is fine
}
```

It's the same problem. We can't make the original goat pointer aim at a different Goat. `ref` fixes it:

```
void blankGoat(ref Goat g) {
  if(g==null) g=new Goat(); // <-changes calling goat to point here
  g.name="test goat"; g.age=0;
}

blankGoat(ref gg1); // works, even if gg1 is null
```

34.6 Raycast example

`Raycast` is probably the most complicated function in Unity – it uses an output parameter with a `bool` return, and has a bunch of overloads and default parameters. It makes a nice example of lots of things, including call-by-reference.

What it actually does is pretend to shine a laser pointer in the 3D world. It tells us what it hit, if anything. We use it to shoot a game laser, or check where a mouse is aimed, or check whether we have room to move in a certain direction.

With all the combinations of overloads and default parameters, there are 16 versions! Four are simplifications of this monster:

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hitInfo,
   float maxDistance = Mathf.Infinity, int layerMask = DefaultRaycastLayers,
QueryTriggerInteraction queryTriggerInteraction = QueryTriggerInteraction.UseGlobal)
```

The last 3 use default parameters. That means we can ignore them, leave them out, and the system fills them with the standard values. That leaves us with this:

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hitInfo)
```

The first two inputs are the imaginary ray we're shooting: where it starts in 3D, and the arrow to follow coming out of the start.

The third is the answer. `out` means it's an output parameter, which means we merely need to give it a blank `RaycastHit` struct. That's a specially made

struct, whose only purpose is to hold results of a raycast. As usual, `hitInfo` is an unimportant variable name.

So here's a legal use of raycast. It shoots from the camera, straight forward:

```
Vector3 pos = new Vector3(0,0,-10); // where camera is
Vector3 dir = new Vector3(0,0,1); // forward along Z

RaycastHit RH; // will hold output
Physics.Raycast(pos, dir, out RH)); // now RH has info
print("We hit " + RH.transform.name);
```

But a raycast could also hit nothing. That's why it returns a `bool`. False means it missed. We use it like this:

```
if(Physics.Raycast(pos, dir, out RH)) {
  print("We hit " + RH.transform.name);
}
else { print("We missed everything"); }
```

Let's play around just a little more. The 3 parameters we skipped are distance, layerMask, and triggerInteraction. We have to fill them in from left-to-right, which means we can use distance.

This shoots a ray for 20 units (if something is 21 units away, it won't reach and returns false):

```
// shoot a ray only 20 units forward:
if(Physics.Raycast(pos, dir, out RH, 20)) {
  print("hit "+RH.transform.name);
}
```

Another version uses a `Ray` as the first input. Here's the struct `Ray`. It's incredibly simple, for a struct:

```
struct Ray {
  public Vector3 origin;
  public Vector3 direction;
}
```

All it does is group together a starting position and a direction arrow. Here's a raycast using a Ray:

```
Ray rr;
rr.origin = new Vector3(0,0,-10); // using the same positions
rr.direction = new Vector3(0,0,1); // and arrow as earlier

RaycastHit RH;
if(Physics.Raycast(rr, out RH)) { ... }
```

Finally, here's a semi-useful script to test raycasts. The player moves along the bottom of the screen, shooting a raycast straight up. Whatever it hits turns red, then white when we move away:

```
Vector3 upDir = new Vector3(0,1,0); // This won't change
Transform oldHit = null; // the thing we were hitting last frame

void Update() {
  movePlayer();

  RaycastHit hitInfo; // results go in here
  Transform newHit; // what's above us this frame (can be nothing)
  if(Physics.Raycast(transform.position, upDir, out hitInfo)) {
    newHit = hitInfo.transform;
  }
  else {
    newHit=null;
  }
  // do something if we hit something different:
  if(newHit != oldHit) {
    if(oldHit!=null) setColor(oldHit, false); // reset old to white
    if(newHit!=null) setColor(newHit, true); // color new thing red
    oldHit=newHit;
  }
}

void setCol(Transform tr, bool isOn) { // on=red, off=white
  Color cc = Color.white;
  if(isOn) cc=Color.red;
  tr.GetComponent<Renderer>().material.color = cc;
}

float xPos=0;
void movePlayer() {
  if(Input.GetKey("a")) xPos-=0.1f;
  else if(Input.GetKey("s")) xPos+=0.1f;
  xPos = Mathf.Clamp(xPos,-7,7);
  transform.position=new Vector3(xPos,-3.5f,0);
}
```

34.7 Reference vs. reference

You may have noticed that we now have two different uses of the word reference: call-by-reference, and reference types. That's an unfortunate accident, especially since they have similar meanings, but different.

Reference types are actual variables, with their own boxes. Which are pointers. Call-by-reference variables aren't their own variables. They're alternate names for the original box.

It's common to say `Dog d1;` is a reference. Inside a function where `w1` was passed-by-reference we'd also call `w1` a reference. Yikes!

Then take this function:

```
void A(Dog d1, ref Dog d2, ref int x) { ... }
```

`d1` is a reference variable passed by value. `d2` is a reference passed by reference. `x` is a value-type passed by reference. Yikes, yikes!

Chapter 35

Arrays

Arrays are crude versions of `List`'s. They work the same, with one huge drawback: they have a permanent size which can't change. That's pretty bad. You want Lists for almost everything.

We learn about arrays because they're the real way computers store sequences. They're built into many languages. Lists are actually made using arrays. Many built-in functions use arrays because they're simple, often good enough, and you can turn them into Lists if you need.

35.1 Basic array rules

Array index loops use `N[i]` and `N.Length`. This sets everything in an array to 0:

```
for(int i=0; i<Nums.Length; i++)
  Nums[i]=0;
```

Besides `Length` instead of `Count`, this is identical to how a List does it.

Like `List`'s, arrays are a 2-part type. `[]` means some sort of array, with the type in front. `int[]` means "array of ints". Others:

```
int[] A; // array of ints
float[] B, C; // 2 arrays of floats
bool[] LightIsOn; // array of bools
Cow[] Barn1; // array of Cows
```

Don't confuse the square brackets with indexing. This is a special use for declaring. The inside is always empty. `int[]` is the only way to do it.

`C#` arrays are reference types. They start as `null` and need to be created with `new`. You're required to give them the permanent size, in square brackets:

```
int[] A=new int[5]; // A is a length 5 int-array

float[] B=new float[5], C=new float[20];
// B is a length 5 float array, C is length 20

string[] W;
W = new string[12]; // length 12 string array
```

This is another completely new rule and a different use for `[5]`. Again, don't confuse it with an index.

All of the rules together, this makes a size 10 array full of the word cat:

```
string[] Animals=new string[10]; // size 10 empty array
for(int i=0; i<Animals.Length; i++)
  Animals[i]="cat";
```

The rule where arrays can never change size goes together with the rule where you have to give a starting size in the `new`. Without being able to add items, the only way an array can ever get a size is during creation.

It has to do with how memory works. Memory is like a line of boxes and things tend to be created from one end. After you `new` a size 10 array, the next things you create will come immediately after. There's no room to grow most arrays. Even if there were, the memory allocator isn't really set up to grow an existing chunk.

35.2 Arrays and functions

Functions take arrays as inputs and outputs in the usual way. As before, `int[]` means it takes an array of int's. The input array can be any size. For example, this counts how many of a certain number are in an array:

```
int countNumsIn(int[] A, int countMe) {
  int count=0;
  for(int i=0; i<A.Length; i++) if(A[i]==countMe) count++;
  return count;
}
```

It looks the same as the List version, except for `int[]` instead of `List<int>` and `Length` instead of `Count`.

Because they're pointers, you can use a function to change the contents of an array. This makes every number be even. The array we give it will be changed:

```
void forceEvenValues(int[] N) {
  for(int i=0; i<N.Length; i++)
    if(N[i]%2!=0) N[i]++; // make it even by adding 1
}
```

Returning an array uses the same syntax: `string[]` in front means it returns an array of strings. This creates and returns an array of the size you want, filled with the word you give it:

```
string[] makeSameWordArray(int len, string wd) {
  string[] Result=new string[len]; // make array, final size
  for(int i=0; i<Result.Length; i++)
    Result[i]=wd;
  return Result;
}
```

We could use it like `string[] A=makeSameWordArray(5,"cat");`.

35.2.1 Array work-arounds

An advantage of `L.Add(3)` with Lists is not needing to know the final size in advance. Since arrays need to know, making an array often has 2 steps. First we count how long it needs to be, then we create it and fill it up.

This returns an array with all of the words longer than 3 letters:

```
string[] longWords(string[] A) {
  // Step 1. Find # of words in answer:
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(A[i].Length>3) count++; // length of string

  // Step 2. Make the answer:
  string[] R=new string[count]; // using the count we computed
  int ri=0; // index into R
  for(int i=0;i<A.Length;i++) {
    if(A[i].Length>3) {
      R[ri]=A[i];
      ri++;
    }
  }
}
```

Notice how we needed an extra index for the answer in the second loop. Suppose it turns out the final list should have items 2, 5, 6 and 9. We need `ri` so we can copy those into 0, 1, 2, and 3 of the result. It was a lot easier being able to use `Result.Add(A[i]);`.

Adding 2 arrays end-to-end isn't as bad. We know the final size is the sum of the sizes, but we need math to add the second to the end:

```
string[] combineEtoE(string[] A, string[] B) {
  // we need to pre-make the result array to the final size:
```

```
  string[] R=new string[A.Length+B.Length];
  for(int i=0; i<A.Length;i++) C[i]=A[i]; // copy A
  for(int i=0; i<B.Length; i++) C[i+A.Length]=B[i]; // slide-copy B
  return C;
}
```

If we had a List, the second loop would be simply `C.Add(B[i]);`.

Arrays can change size, sort of. We can create new empty array, 1 box larger; copy everything over; then aim at the new array. This adds `"zebra"` to the end of string array `W`:

```
string[] W2=new string[W.Length+1]; // 1 larger than W
// copy everything over:
for(int i=0; i<W.Length; i++) W2[i]=W[i];
W2[W2.Length-1]="zebra"; // last, new, spot is "zebra"
W=W2; // aim at new array
```

The old array is abandoned and garbage-collected. That works, but if we add a lot it's slow and makes a ton of garbage.

35.3 Max size / current size trick

We often fake an array that can grow by making it the maximum size and marking how much we really use with an extra int:

```
string[] W = new string[20]; // 20 is maximum size
int wsz=0; // current size is 0

W[wsz]="dog"; wsz++; // adding dog
W[wsz]="horse"; wsz++; // adding horse
```

The end result: `wsz` is 2. `W` counts as size 2. And `W[0]` and `W[1]` are our animals.

We usually put them in a class:

```
class GrowArray {
  public string[] W;
  public int sz=0; // current "size" (the amount being used)

  public void reset(int maxSize) { W=new string[maxSize]; sz=0; }

  public void Add(string ww) {
    if(sz>=W.Length) { print("no room"); return; }
    W[sz]=ww; sz++;
  }
}
```

It's the same trick, but using a class covers it up and makes it easier to use.

When we search the list, we have to remember to use `i<sz` instead of `i<W.Length`.

This is actually how the List class works. It has a hidden array. But instead of quitting when the array is full, it creates a new array twice as big and copies.

35.4 creation shortcuts

Because arrays are built-in, they have an easy shortcut for making them pre-filled. {2,8,6} makes a size 3 array with those numbers. It can only be used when the array is declared:

```
string[] Ani = {"cow", "hen", "pig"}; // size 3 array of strings
int[] N = {1, 7, 12, -5, 7}; // size 5 array of ints
```

The system counts the items, creates an array that size, and fills it with those values. As usual, there's nothing special about starting values. You can change them later.

To use the shortcut anywhere else, you need extra in front. Add `new int[]`:

```
Ani = new string[]{"worm", "slug", "snail","bug"};
N = new int[]{3,4,7,4,9.12};
```

As with most array syntax, it's not part of a pattern or larger rule. It works that way because that's the rule we made.

You can even use it inside a function call, often for testing:

```
testMe(new string[]{"ant","bug"});
testMe(new string[]{"x","y","z"});
```

35.5 List / array conversion

Lists have a constructor taking an array as input:

```
int[] A={4,8,12,7}; // sample array
List<int> L=new List(A); // new List, same items as A
```

We often use it with the array-creating shortcut. It's long, but shorter than hand-adding everything:

```
L=new List<int>(new int[]{1,4,6,8,11});
```

Lists have a function to return an array of what they are:

```
int[] N = L.toArray();
```

It makes a copy, so we can change the contents of `N` without changing `L`.

A common use for this is a function that takes an array as input, when we have it in a list:

```
List<float> LF; // we use this list
oldArrayUsingFunction(LF.toArray());
```

35.6 OverlapSphere Unity example

Unity's `OverlapSphere` function returns an array. You call it with an imaginary ball, in the 3D world. It tells you every Cube, Ball and so on partly inside of it. This finds everything within 2 spaces of (0,5,0):

```
Vector3 center=new Vector3(0,5,0);
Collider[] Hits; // an array of colliders, currently null
Hits=Physics.OverlapSphere(center, 2); // returns an array

for(int i=0; i<Hits.Length; i++) {
  string w=Hits[i].transform.name;
  print("hit " + w);
}
```

It returns an array of Colliders (they go on gameObjects). It's an array instead of a List since all you're going to do is look through it. An array is fine for that.

Every time you run it, a new array is created and thrown away when you're done. That's not a big deal. But OverlapSphere has a version which re-uses your array. You make one as big as you think you need and hand it over. OverlapSphere fills it and returns how many there were:

```
Collider[] H=new Collider[10]; // 10 should be plenty

int hits=Physics.OverlapSphereNonAlloc(center, 2, H);

for(int i=0; i<hits; i++) { // <- loop up to hits, not H.Length
```

If you overlapped 3 things, they would be in 0-2 of `H`, the return value would be 3, and the rest of `H` would be old junk values.

35.7 2D arrays

If you remember, 2D lists are just lists of lists – no extra rules needed. Arrays have that method, sort of, plus they have a special perfect-rectangle version.

These are mostly good as examples of the crazy rules people jam into computer languages.

35.7.1 Ragged `[][]` arrays

An array of arrays is mostly like a list of lists, except for one funny rule. Write two pairs of `[]`'s in the definition:

```
int[][] A; // A is an array of arrays
```

This would make `A` be 3 by 4:

```
A=new int[3][]; // <-backwards, but this is the rule
for(int i=0; i<A.Length; i++) // A.Length is 3
  A[i]=new int[4]; // each slot is another size-4 array
```

Like with lists, `A[0]` is the first row. It's a single length four array: `A[0].Length` is four. `A[0][0]` is one int in the grid.

We call these ragged arrays since each row could be a different length. We made them all length 4, but they didn't have to be.

35.7.2 Real 2D `[,]` arrays

C# also includes an older special syntax to make perfectly rectangular 2D arrays. A "real" 2D 3 by 4 array would be created like this:

```
int[,] M; // M is a 2D array
M = new int[3,4]; // creates it 3x4, all at once
M[0,0]=7; // use both inside one set of []'s
```

It comes with a special version of the insta-array shortcut:

```
int[,] K = {{1,2,3},{4,5,6}}; // K is 2x3

int[,] J = {{4,7,9}, // using white space to
            {1,1,1}, // make it look like a grid
            {2,6,3}}; // this make a 3x3
```

The rule for finding the lengths is fun, since it has 2 lengths. `M.Length` tells you the *total* number of items – 12 – which isn't very helpful.

`M.GetLength(i)` gets the size of each dimension, starting at 0. `M.Rank` tells you how many dimensions it has (1 is a normal array, 2 is 2D, and so on).

This uses them in a double-loop to change everything to 7's in any sized 2D array:

```
for(int i=0;i<M.GetLength(0); i++)
  for(int j=0; j<M.GetLength(1); j++)
    M[i,j]=7;
```

Chapter 36

Efficiency

In many of my examples, I make a big deal about how to make the program look nice. I don't say much about how to make it more efficient – in other words, making it run faster.

As you write code, there are a lot of little "this way or that way" decisions. For a bunch of different reasons, if you make all these little decisions based on what seems like it would run faster, it doesn't work and turns the code into a mess.

There are things to do to avoid super-slow code, but most of them are when you make the plan – not when you're writing each line.

Some notes:

36.1 Most speed-ups don't work

A lot of the things that seem like they'd speed up a program just don't work. They do nothing, or make it run *slower*. Here's a list of why some things don't work:

The compiler rearranges things anyway

Writing an equation on one line or on several lines doesn't matter. The computer breaks it into step-by-step anyway. Suppose you have this:

```
x = a*b + c*d;
```

The compiler changes it into:

```
int temp1 = a*b;
int temp2 = c*d;
x = temp1 + temp2;
```

By the same rule, declaring lots of extra temporary local variables doesn't hurt you. The computer was going to fake-declare them anyway.

`if` expressions are the same way. For real, the computer can only have one thing in an `if`:

```
// you type this:
if(n>=1 && n<=10)

// compiler turns it into:
if(n>=1)
  if(n<=10)
```

Or's are broken up using a different trick (it's hard to show, but it works).

The compiler might even flip the test. If you write `if(n>0) A else B` the compiler might turn it into `if(n<=0) B else A`.

So go ahead and write `if`'s and conditions how-ever it looks nicest. If there was a faster way you could have re-arranged them, you may as well let the compiler find it.

The compiler knows the speed-ups

Over the years, we've found a lot of mini-tricks to rearrange code to speed things up. And we've programmed them, and put them into compilers. In other words, if you can look up a speed-up trick, and it works, it's probably already in the compiler.

The computer will pre-do math with constants. For example `n=7*24;` is fine to get hours in a week. The compiler turns it into 168. When the program runs, the line is just `n=168;`.

In general, leave math in whatever form is easiest to read. I like to write `Random.Range(1,6+1);` to remind me it's going to 6, with the 1-less rule. It's not slower, since the compiler makes it a 7 before the program ever runs.

The compiler can re-arrange to factor out common terms. For example:

```
int n = x+b*3-1;
int m = (y+b*3-1)/2;
```

You might notice both use `b*3-1`. We could compute that once and re-use it:

```
// "improved" version:
int q = b*3-1;
int n=x+q;
int m = (y+q)/2;
```

But a good compiler does this for us. Over 50 years very smart people have written papers about simplifying equations, which made their way into compilers.

If the code would look better, go ahead and precompute. Otherwise let the compiler handle it.

Speed-up logic is hard

This is an old example of a non-working attempt at a speed up. Suppose resetting a game round resets `health=100;`. That's a waste if our health was already 100. We'll improve things by checking first: `if(health!=100) health=100;`

But `if`'s take time, too. The old way took 1 step. This way takes at least 1 step, usually 2.

But suppose resetting health to 100 also resets the health bar. Would an `if` here save time?:

```
if(health!=100) {
  health=100;
  redrawHealthBar(); // takes 30 steps
}
// else the health bar is already displaying 100 health
```

If still takes one extra step if the player is damaged. If that's the usual case, this way is still slower than without the `if`'s.

36.2 Premature Optimization

Let's suppose we can write each part of the program in two ways: runs faster, or easier to read – one or the other. Another way to say the same thing, suppose we can rewrite any part to be a little faster, but more confusing.

When would we decide to do it which way?:

- If we need to test it and are pretty sure there will be some things that need fixing, easy to read is better.
- If we're going to expand it later (for example, a match game using colors and we probably want to add matching shapes) then easy-to-read is better.
- Often part of our code is temporary or experimental – we're 70% sure it will eventually be replaced by something better. There's not much point making it faster.
- If we get a lot of requests from customers for changes in this part, easy to read is better.

- If this part is done, tested and you're confident it won't need to be changed, running faster is an option.

An example, here's my simple "keys move left and right" code:

```
if(Input.getKey(KeyCode.A) pos.x-=0.1f;
if(Input.getKey(KeyCode.D) pos.x+=0.1f;
pos.x=Mathf.Clamp(pos.x, -7, 7); // runs even if we don't move
transform.position=pos; // ditto
```

When we don't move, this code uselessly makes sure `pos.x` is between -7 and 7, and uselessly resets our position. We could speed it up by only running the last two lines after we moved:

```
float oldX=pos.x;
if(Input.getKey(KeyCode.A) pos.x-=0.1f;
if(Input.getKey(KeyCode.D) pos.x+=0.1f;
if(pos.x!=oldX) {
  pos.x=Mathf.Clamp(pos.x, -7, 7);
  transform.position=pos;
}
```

This runs slower! When we're moving, this is two lines *more* than the old code. It runs a little faster when we're not moving – but who wants a game that speeds up when you stop?

But I can keep trying. Here's my next attempt which really is faster:

```
if(Input.getKey(KeyCode.A) {
  pos.x-=0.1f; if(pos.x<-7) pos.x=-7;
  transform.position=pos;
}
else if(Input.getKey(KeyCode.D) {
  pos.x+=0.1f; if(pos.x>7) pos.x=7;
  transform.position=pos;
}
```

This always runs faster, but I think it's harder to read (two lines setting position, extra 7's).

But wait. We weren't done making the game. We want to slide sideways for a few frames after you let go of a key. Then we want wind and bumping to also move the player. Those are going to be total rewrites. Time spent speeding up the version above was just wasted.

36.3 Not all code sections are equal

Suppose you have one player and thirty enemies (or anything using a script with Update). For speed, the enemy code is 30 times more important. In other words, if you think of 30 little ways to speed up the player's code, you could have thought of just one way to speed up the enemy code.

Suppose you have a hundred lines in a script, then a nested loop around two lines. Maybe it checks every square in a 100 by 100 grid:

```
do stuff #1
do stuff line #2
...
...
...
do fortieth thing
...
do fiftieth thing
...
do one hundredth thing

for(int i=0;i<100;i++)
  for(int j=0;j<100;j++) {
      loop line #1
      loop line #2
   }
```

The inside of the loop runs 10,000 times. Compared to the hundred lines before it, the loop is 99% of the total time.

Making those hundred lines ten times as fast would be less than a 1% speed-up. Improving the lines in the loop is all that matters.

You can also have a huge loop even if you don't have that many items. Suppose we have an array of 100 foods and compare every one to every other. That's another ten thousand times loop.

Suppose the inside has a function call that also loops over the array. Everything in that function loop runs `10,000*100 =` a million times. From just a size 100 array.

Accidentally writing a triple-nested loop isn't all that uncommon. A lot of making code run faster isn't trying to speed up good code – it's looking for places like unnecessary triple loops.

But even more, there are lots of places where you don't care about speed at all: the Start / Pause / GameOver screens. Scrolling text. The code to fix player names longer than 20 characters. Calculating your "rating" after you finish a level (there's a delay and a little spinning animation, anyway).

36.4 Is it already fast enough?

There are places where obviously we care about speed, but once we get it fast enough, there's no point getting any faster.

Almost any menu-based web program only has to run faster than human reaction speed, which isn't very fast at all. Humans can't click buttons more than ten times a second. If you can respond in 1/10th of a second (which is forever to a computer), any faster won't matter.

For a game, screens update at 30 or 60 frames/second. We have at least 1/60th of a second to do all the work for the next frame display. If we're making a matching game, that's plenty of time. "Speeding it up" will have no visible effect.

36.5 Will it cause more errors?

"Faster" code tends to be more difficult to read and more complex, which means it tends to have more, harder to fix, errors. Beyond that, some speed-up tricks add new possible errors.

A fun one is how the caching trick can add a new out-of-synch error. A typical example of caching is a link to some other Cube, with a shortcut link to it's material:

```
public GameObject otherCube; // this is pre-set
Material otherCMat; // we compute shortcut to Material of otherCube

void Start() {
  otherCMat = otherCube.GetComponent<Renderer>().material;
}

void Update() {
  otherCMat.color=c2; // using the shortcut
  // otherCube.GetComponent<Renderer>().material.color=c2; // the long way
```

The new problem is that we can aim `otherCube` at something else, while forgetting to recompute `otherCMat`. To an unsuspecting tester, that error makes no sense – it shouldn't even be possible. Which means it can be extra-difficult to fix.

36.6 Things that work

In general, focus on high-use parts of the code - for example things that run for 50 items, every frame.

A main way of speeding things up is getting rid of accidental nested loops. They can be hidden, like a loop calling a function which runs another loop. A common Unity example is `GameObject.Find`. That's a loop through everything in the game – maybe a few hundred things and not too bad. But if you forget, you might put in inside another loop and it blows up into tens of thousands of steps.

`List`'s are a good way to have accidental nested loops. I mentioned they're really arrays and doing anything not at the back involves a slide-loop. If you don't know this, you might add to the front, `L.Insert(0,w);`, in a loop. Adding 100 items that way is 5,000 steps, compared to `L.Add(w)` at only 100 steps. Remove is the same way. Removing the last item is 1 step, removing the first is a slide-loop to shift everything down.

Other tricks involve looking at your plan.

For example, code handling monsters often checks line-of-sight to the player, and `Raycast` is doing lots of math. Instead of running it every frame – 60 times/second – we can run it twice a second. That's a 30x speed-up. It will cause monsters to sometimes have up to a 1/2-second reaction delay. That's fine. In fact, it makes them feel more realistic.

Path-finding (computing a path through a maze or building) is crazy slow. You can often run that every few seconds, or only when something changes; and have some monsters just follow others.

Many actions on lists are much faster if the list is sorted. We can often keep lists sorted as we add items. This takes lots of practice and some theory

LinkedLists are the other way of storing items in a row. They're very fast to insert and remove from any position, just as fast to loop through end-to-end as arrays, but much slower than arrays if you need to jump around with look-ups. Choosing to use an array or a linked-list can make a big difference, but knowing which takes lots of practice.

Using less space will speed up a program. Memory is divided into high speed, medium speed and regular. The more of your program which fits into the high and medium speed, the faster it runs.

This is partly why Unity uses floats instead of doubles – less space. But it's still a matter of finding the places where it matters. If you have one especially huge array, cutting the size in half might help; but don't waste much time making small arrays smaller.

The last things we can do are making little tweaks to the ultra high-use code parts. If you work like a dog and are lucky, you might get a 2% speed-up.

`v1.magnitude` (length of a vector) vs. `v1.sqrMagnitude` is an example. Finding the distance ends with a square root. The second command leaves it out, to be a bit faster. You could change `if(v1.magnitude<3` to `if(v1.sqrMagnitude<9)` for a very small speed-up (maybe).

Chapter 37

Debugging

I tried to put bits about testing a program and finding errors all through this. But we never wrote any really big programs, and I wanted to sum up, plus mention some common debugging tricks.

37.1 Test as you go

The main trick to debugging code is to write a little, and then test. Put another way, it seems like it would be easier to just not make any mistakes, but that never happens.

The quickest way to get a working program is to write a little, test, write a little more, and so on.

Some of why that's true:

Having two bugs at once is horrible, especially if they overlap. You could fix one and not even know. Testing as you go is much more likely to find one error at a time.

You might have a bad plan. Early testing tends to find it before you write lots of stuff that needs to be torn out later when you finally see the problem.

Sometimes part of your programs are basically copies. It's better to find bugs in the original before making 3 copies of the same error.

You'll have to test everything anyway. At some point you'll have an error where you can't even imagine what's causing it. This doesn't seem possible: if something is in the wrong spot, the error is obviously in the equation. But there are times when you've checked the obvious places, and they're fine. Something that can't be causing the error, is causing the error. The only way to find it is to test every part.

Put another way, after you write a simple function that can't possibly be wrong, there's 100% chance of it being a suspect later on. You will definitely at some point think "well, maybe the problem is here – I never tested it".

37.2 How/what to test

If there are any commands I'm not sure about, I usually write a small, separate program to test them.

For example, Unity can use "layers" to make a Raycast ignore certain objects. If that sounds great for what you need, make a small test with fresh Cubes. Assume there's a least one thing you don't understand, because there always is, and try to find it.

New language features – anything big – usually take an entire practice project to learn. If you start using a new feature in a real project, it pretty much automatically turns into a practice project. In other words. you can write a useful program, or you can write a program to learn something new. But not both.

You don't have to start writing the first part of the program first. You can write functions, then test them with fake function calls in `Start`.

Typical tests are "sanity checks" where you run it with really easy inputs where you know the answer. Another is an edge-case: try numbers just before and after cut-offs. If something checks for out-of-bounds, try things that are barely out, barely in, and use every boundary.

Most bugs are in things that you don't test, not even once, since there's no way it can be wrong.

A test-loop can try lots of numbers, with an `if` to check that the results are at least close. Sometimes you have to be creative – if a function checks a 10x10 board, you'll have to fake one up. That seems like a lot of work, but you'll have to do it anyway when you get the first impossible to track down bug.

You can test a main part of the program by faking up functions. If something is suppose to read user input, you can have it temporarily give a random, legal number. If a function is suppose to decide where to walk next, it can temporarily pick 20 feet ahead, or a random spot.

37.3 Tracking down bugs

No matter how much you test, there's always some bug that pops up out of nowhere. Here's are some tricks to track it down:

- When you find a bug, keep testing to make sure what it really is. Just run normally and try more things.
  If an enemy doesn't die when it should, do they never die, or sometimes

not die? Is it every enemy, or just that one? Is it only when you use a certain weapon? If you wait and shoot it some more, does it die then?

Usually more testing will give clues about where the problem really is, or isn't.

- Don't assume it's what you just did. Maybe your game crashed after you added balloons. But it really always had a bug when the score goes past 50, and balloons just happened to be the first thing to do that.
- Use prints to narrow things down. Somewhere in the program, some number isn't what it should be. Enough prints can find it and track down who's doing it.
- It's very common to be sure the error is in one place, and be completely wrong. At some point, do a quick check of all the things you know can't possibly be causing the problem. If computed value is wrong in a textBox, put just `"cat"` in there. If you don't see `"cat"`, there's something wrong with the box.
- Sometimes, after trying to figure out some old, horribly awful section you wrote, that keeps having problems, you decide to just rewrite it. Only do it if you think this section is so bad that it's going to suck up more and more time. Lots of people go nuts on rewrites. If a section looks terrible, but works, it's fine.

Some specific tricks:

Add "is this even running?" print statements. If your function to align text isn't working, it might be because it never got called. If your function to make 10 Cubes is mistakenly making 20, it may be working just fine, but was called twice. Run and then check to see it ran everyplace it should, and nowhere it shouldn't:

```
print("A"); // we got to point A
if(n>=0 && active==true) {
  print("B"); // we entered the if statement
```

You can see the *stack trace* from a print statement. It will list all functions that were called to get to this one. Suppose blocks are removed for no reason. Add a print in the `removeBlock` function – it will show you exactly who's calling it when they shouldn't be.

To really check a function, print all of the values. After running, check to see which ones are wrong:

```
int doStuff(int n) {
  print("starting doStuff, n="+n);
  ...
```

```
  print("ending doStuff, ans="+ans);
  return ans;
}
```

Sometimes a function is called too much to read everything it might print. You can narrow it down with `if`'s. This only prints for a Player input:

```
int doStuff(int n, GameObject gg) {
  if(gg.transform.name=="Player") // don't print all the time
    print("doStuff on Player, n="+n);
```

Occasionally you have a problem caused by reading old data. I like to sometimes change-up a print statement to check. If the first line in Start changes to `"In Start version II"` and the output is `"in Start"`, you know something is very wrong.

You may have been reading old output this entire time. Or the system may not have been reading your changes. You may need to find a Recompile-All option, or a way to delete old files to force it to recompile.

Sometimes you can't find a nullReferenceException because of a chain. In the code below, the null could be in `c1` or `favoriteDog` or in `gg`, or there could be no renderer:

```
float r = c1.favoriteDog.gg.GetComponent<Renderer>().material.color.r;
```

To track it down, it can be broken into single steps, with prints in-between:

```
if(c1==null) print("no c1");
Dog d=c1.favoriteDog;
if(d==null) print("no dog!");
else print("got dog "+d.name);
GameObject gg = d.gg;
if(gg==null) print("no gameObject");
// and so on ...
```

That seems like a lot of typing, but it's better than sitting around thinking "`favoriteDog` must be `null`, right? I mean, all dogs have a `gg` and a renderer, don't they?"

Sometimes it helps to temporarilty simplify the program. If the customer data isn't being displayed correctly, it probably wasn't caused when you computed their order. But it could have been. Comment that out. Simplify the program a little bit at a time by commenting out parts. With luck, removing one thing will cause the error to vanish – you found the part causing the bug.

If the problem is with a feature you're not very experienced using, test it in a different mini-program. For example, if this is only the second time you've used

Unity's OnCollisionStay, make a very simple demo of how your real program uses it. There may have been a few things you misunderstood.

Some specific Unity tips:

- You can Pause while running. This allows you to select items and examine values.
- While Paused, you can move items, resize, change colors. You can even add new items. This can be helpful to find the scope of a bug.
- When spawning items in a loop, change the name, such as `"cat"+i`. This might help you realize the bug is always the first one made, or the last, or something else to do with the item #.
- You can make fake Inspector variables to store locals. Suppose you want to watch your velocity:

  ```
  public Vector3 vvv; // a copy of our velocity that we can see

  void Update() {
    vvv = GetComponent<Rigidbody>().velocity;
  ```

Be very careful about hacks. For example, you can't fix the bug when `x<0`. Instead you add an extra `if` statement in Update to manually fix it. Those things usually turn into super hard-to-find bugs later. If you have to – your demo is tonight – put a big comment `HACK FOR BUG #258`

Be sure to remove all of your test lines. I once spent 15 minutes trying to figure out why it always jumps to stage 4. It was an old testing line that jumped to stage 4. Write them down, or make then super-easy to see, or reload your backed-up file.

When you fix a bug, take some time to remember exactly what fixed it. Maybe write down problem/solution. It's very common to see a similar bug and all you can remember is "I tried this, looked here for a long time, but the problem was . . . arrg! I forgot!".

Don't make casual fixes. When you're looking through a function for a bug, it's irresistible to do a little clean up and quickly fix things that are "obviously" wrong. That seems crazy – who would do that? But when you see `i<L.Count-2` in a loop, it's so, so hard not to think "minus 2? How did that get there? I can save so much time if I fix it now".