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. There are options to include various extra Assets – we won't need any of them, but they won't hurt. You can always get them later.

If you haven't used it 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 between 1 and 2 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 one line from the Console. The simplest way to pop open the Console window is to double-click that line (only if something is there.)

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 MonoDevelope 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. If the *Light* Inspector area is open, the little triangle can pop it shut.

Then 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

One thing that can go wrong is you get multiple copies running. For example, you accidentally drag it twice onto the Light. Or drag it onto the Light and the Camera.

The system will run two copies of your program at the same time. It won't necessarily cause a problem, but it will double-print everything, confusing you.

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 changing the name twice. You have to enter the final name immediately after you create it. It you take the name it gives you, that's fine. But if you rename it later things will break. Or renaming it again will break things. For now, the easiest way to fix "Can't add script, names don't match" is to delete the script from Project (right click) and remake it. If you write something inside that you needed, cut&paste works fine in the script editor.

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 Monodevelope 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.
- It's case-sensitive. 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.)

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:

- Lines from each run might pile up in the Console, making it hard to see what's new. 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
- Monodevelope (where we typed the program) has lots of buttons on top, including `Run`. We don't need to use any of those. They're for stand-alone programs. Unity is just borrowing the part of Monodevelope that helps us write code.
- 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 apce 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, ...)

That way the end-user gets the smallest possible program, which runs faster as well.

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. We know the open-curly could have been on the line after Start. Just `void Start()` isn't an error so far. But the next line doesn't have one either. 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. And a double-click takes us one line below the problem.

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 all of the extras.
- 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. Also 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.