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.