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 change the order we 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" ); }
```

This runs the line `print("moo");` over and over. Actually, since `n` is always less than 0 it runs that line forever. It completely freezes all of Unity.

More than most things, loops need us to have a plan. There are only a few simple computer rules for how they work. The real work is all the tricks we know to make loops do useful stuff.

This non-useless loop runs the print 10 times, printing `n=1` through `n=10`:

```
int n=1;
while(n<=10) {
  print("n="+n);
  n++;
}
```

The rules are probably obvious (I'll list them all later): put a true/false test in parens, the same as an `if`. The body goes in curly-braces.

The new part is after running the body: a loop 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

I want to put this part near the front, since crashing Unity is a pain.

It's easy to mess up the part of a loop that makes it stop and have it run forever, which freezes the program. We call that an **infinite loop**. Our programs using Update technically ran forever, since you had to press Stop. But this is the first time they can get stuck.

In most environments you can press the Stop button. But in Unity3D the entire window freezes (not merely the Game window – the entire window running Unity.)

You have to force-quit Unity (depending on the computer, right-click and force-quit, or go to the task manager.) The rest of your computer will be fine, so you can always search the internet for "how to make a program quit."

That's a pain, but won't cause any harm, but you lose anything unsaved.

You won't lose scripts (since you already saved them before running.) But you will lose things like making a Cube. If you're made any non-script changes, it's not a bad idea to use `Save->Scene` and `Save->Project` before running any risky loops.

22.2 Number-sequence loops

A common way to "drive" a loop is having it move a number up (or down) to hit some sequence. This one prints even numbers from 2 to 20:

```
int n=2;
while(n<=20) {
  print(n);
  n+=2;
}
```

Walking through this: 2 is less than 20, so it runs the body. We print 2, change `n` to 4 and jump back. 4 is less than 20, so we print 4, change `n` to 6 and jump back.

The last time we run, the body prints 20, changes `n` to 22 and jumps back. `22<=20` is false, so we quit. Like an `if`, we jump past the body.

Lines can be in any order, but printing, then changing is the best way. It makes the start and stop easy to see. `n` starts at 2 and the first thing we print is 2. It checks `(n<=20)` and the last number we print is 20. Sometimes we do more than just printing, but we almost always want "go to the next `n`" at the end.

To get a feel, here's a worse loop which also prints 2 to 20 by 2's:

```
// 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 at 0 in order for the first printed number to be 2. Likewise the last thing this does is run when `n` is 18, add 2, print 20, then jump back. 20 is past 18 so it jumps out. Ouch.

Another idea, which works out to be bad, is to write the stopping number. We know the loop counts 2, 4, 6 .... We could have it stop when it hits the first number past 20, which is 22:

```
// another not-so-good 2-20 by 2's:
int n=2;
while(n!=22) { // <-change to !=22
  print(n);
  n+=2;
}
```

Beside showing the final number and not requiring arithmetic, `n<=20` has two more advantages. It shows `n` is increasing, and it also works if we skip past. If we started `n` at 1, the `(n<=20)` version prints 1,3 up to 19 (21 is past 20, so it quits.) The `(n!=22)` version is an infinite loop starting from 1 (it skips from 21 to 23.)

That's what I meant about the tricks to make loops do what we want. "Change at the end" and "use a less-than" aren't rules, but loops are hard and those make them less so.

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;
}
```

Hopefully you can see the spots for start, stop and how to move. We could use a function to make a generic print-increasing loop:

```
void printUpSeq(int start, int end, int stepBy) {
  int i=start;
  while(i<=end) {
    print(i);
```

```
    i+=stepBy;
  }
}
```

`printUpSeq(50,100,10);` would print the 50,60,... 100 above. Using `printUpSeq(0,20,1)` would print just 0 to 20.

Then try `printUpSeq(50,100,200);`. It counts 50 to 100 by 200's. The first step past 50 is 250, which is too big, so it quits after printing only 50. That's kind of neat it gives the correct answer on even oddball input like that.

Counting backwards is the same pattern as counting up. We have to flip the test to use `>=`. This counts from 40 down to 10 by 1's:

```
int i=40;
while(i>=10) { // NOTE: 40>=10, stops at 9
  print(i);
  i--;
}
```

The same as before, we like how `(i>=10)` helps show that `n` is going down, towards 10.

Messing around, suppose we forgot to flip, and left it as `i<=10`. The first time would check `(40<=10)`. That's false so the loop would jump out, never printing anything. That's legal – loops can run 0 times. But it's the wrong answer.

Suppose instead we forgot to change `i++` to `i--;`. The loop would count 40, 41, 42 .... Testing `(i>=10)` would always be true. We'd have an infinite loop counting up from 40 forever.

To sum up: these things are *sensitive* and getting them wrong can give unpredictable results.

For a different summing up, this function can count up or down, checking to be sure it won't run forever. It's all the previous tricks, combined:

```
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: step is negative, so i+=step is going down
  }
}
```

We can move the number using other math. This doubles it. It prints 2, 4, 8, stopping past two digits. Notice it's the same pattern: where to start, when to stop, move the number at the end:

```
int i=1;
while(i<=99) {
  print(n);
  i*=2;
}
```

This is where the less-than comes in handy. I'm not sure what the exact last value will be, but I don't need to know. `i<=99` says to quit when you see a three digit number (it so happens this print 64, then quits on 128.)

Here's the same loop going backwards – dividing by 2. We have to think a little here – integer divide rounds down. If we start with 100 we'll get 50, 25, 12, 6, 3, 1, 0. Dividing anything enough will always hit 1, then 0. So we can stop when we hit 0:

```
int i=150;
while(i>0) { // quit when we drop to 0
  print(i);
  i /= 2; // <- rare "divide me by this" shortcut
}
```

22.3 While loop rules

As usual, 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, which usually causes it to spin and freeze the program.

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 never an `else`. With the way loops spin around you wouldn't use one, and it's simpler to add an extra `if`. You might be thinking I'm not telling you about while-else since it's complicated, but there really, really never is a while-else.

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;` the rules say to jump back to the start. But `i==4` is now false, so it jumps right back to the end again. When you run a loop in your head, it's safe to check at the end, either quitting or jumping back. That works out the same as the real rule. But when you have a hard one, remember the actual rule is "jumps back after running the body".

`if`'s and `while`'s only test at the start. For example this `if` prints moo, even though it becomes false half-way through:

```
i=3;
if(i<10) {
  i=28; // more than 10, but does not quit. We already checked
  print("moo");
}
```

Since `while` loops repeat, they seem as if they're always checking. But they won't check during the body, only one time before each spin.

You can't use a loop for motion over time. Compared to running Update over and over, loops run instantaneously. If you write a loop that gradually changes your position from -7 to 7 in a hundred small steps, it's the same thing as having those changes in a row, which is the same as setting position to 7.

Loops are only good for computing things (in other words, the same as `if`'s or anything else.)

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
```

No one ever does this next thing, but another plan for doing something 10 times is 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. Also, since we don't see it as much, it seems at first like it might run 9 or 11 times.

22.5 Loops with floats

For real int's are more common, but you can use a float to drive a loop. Everything works the same. This will print from 1.5 up to 10 (as usual, probably not 10 – the largest one until 10) going by 0.6:

```
float n=1.5f;
while(n<10.001f) {
   print(n);
   n += 0.6f;
}
```

If you skipped the chapter on special float stuff, skip this part also: I used 10.001 to account for rounding. I wasn't sure if adding 0.6's would hit 10 exactly, but if it did, rounding might make it be 10.000001 and incorrectly not print something that counts as 10. So this is a typical acount-for-not-exact trick.

But other that me sneaking that in, this is a nothing-special number sequence loop.

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 reroll – sort of like the "doubles go again" rule in Monopoly. Usually you won't roll doubles, like a 1 and a 6 and your roll is a 7. But if you roll a pair of 6's, another, another, then finally a 3 and a 1, your roll counts as 40.

This go-until-done loop does it. 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 to 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) { // <- && inside!
  nn=Random.Range(1, 100+1);
}
```

This loop probably rolls, say, 87 the first time, then quits. But it could roll 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:

```
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;
}
```

It does hurt my head a little. For real I'd test this to be sure. We can use a count-to-100-by-1's loop:

```
int num=1;
while(num<=100) {
  print("num=" + num + " pow=" + pow2ThisOrLess(num) );
  num++;
}
// sample output (1 line of many):
// num=10 pow=8
```

Another mathy-one, suppose I need a square grid with at least 60 spaces. I'll try 5 by 5 (25 spaces), 6 by 6 . . . until finally 8 by 8 is the first one big enough.

The plan for the program is the same: counting `n` up from 1, by 1's, stopping when n by n has enough space:

```
int getSizeSquareGridThisOrMoreSpaces(int num) {
  int n=1;
  while(n*n<num) n++;
  return n;
}
```

The condition looks really strange, but it's what we figured out. When we're on 5, it sees `5*5` is too small, moves to 6, sees `6*6` is still too small. Finally for 8 it sees `(8*8<num)` is finally false, so quits on 8.

Sometimes the short loops are really sneaky.

Back to the dice examples, we can make a minor adjustment to roll a fresh number every Update with no repeats. Instead of avoiding 5, we'll avoid the last number we rolled. To make it easier to check, I'll roll just 1-4:

```
int oldRoll=-1; // previous roll

void Update() {
  int roll=oldRoll; // fool loop into running the 1st time
  while(roll==oldRoll) roll=Random.Range(1,5);
  // the loop is done. roll is different from the old roll
  oldRoll=roll;

  print(roll);
}
```

Notice how it starts the old roll at -1. That's a cute way of saying nothing is off-limits the first time (it rolls 1-4 then rerolls if it got -1, so never.)

Using a global to save the old value is common, and only a little tricky. Notice here how we don't overwrite `oldRoll` at first, since we need to know it to avoid it. Once we finally have a good one, then we copy it into `oldRoll` for the next update (the old value trick takes practice - I wanted you to see it at least once.)

This will just spray 1-4 in the console. If you let it run for a while and scroll, you'll see it doesn't repeat. But the basic idea is still the same as the old "1-6, but not 5."

Sometimes the condition gets complicated. A neat trick is using a bool and `while(done==false)`. Here's "1-6 but not 5" rewritten using a `done` variable:

```
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 what it does, but look how pretty it is. The loop clearly says it runs while it's not done. Then at the end we have a really obvious step where we decide it we're done now. At the start, setting `done=false` is a clean way to make it run the first time.

Here's a last fun but long example of roll-until-happy. Every few seconds I'd like the Cube to jump anywhere else on the screen, but at least 3 away from where it is now. The plan will be to roll a random spot, then re-roll it until we get one far enough away. This rolls the spot:

```
int x=-99, y=-99; // current position, which we avoid
// off-edge values mean we can be anywhere the first time

// new random x,y at least 3 away from old x,y:
void setNewPos() {
  bool done=false;
  int x2=0, y2=0; // the new values we're going to roll
  while(done==false) {
    x2=Random.Range(-7.0f, 7.0f);
    y2=Random.Range(-5.0f, 5.0f);

    // set DONE by checking if we're far enough away
    // get the positive x and y distance from the old position:
    int dx=x2-x; if(dx<0) dx*=-1;
    int dy=y2-y; if(dy<0) dy*=-1;
    if(dx+dy>=3) done=true;
    // else done stays false, and we keep trying
  }
  // we got values we like, copy into globals:
  x=x2; y=y2;
}
```

Figuring out if we like the new position took 3 lines. I wouldn't care to cram all that into the while-parens. This is just simple bool use, but it fits really nicely into this sort of loop.

To be complete, Update would use `setNewPos` with a delay counter:

```
int delay=0; // first one happens right away

void Update() {
  delay--;
  if(delay<=0) {
    delay=50;
    newPos(); // sets new x&y
    transform.position = new Vector3(x, y, 0);
```

```
  }
}
```

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++;
  }
}
```

It's cool since checking `i!=5` makes it seem like we don't know if we're coming from above or below, which we don't know, so we like it there.

And a note: loops like these are especially easy to mess up and make infinite.

Checking for prime number is a very computery loop. I like it since it's another example of programming things the way we'd do them ourselves.

If you remember, 1 isn't prime, 2 and 3 are, just because. Otherwise you're prime if nothing divides you. We can rule out all the even numbers, then try to divide by 3, 5, 7 ... up to half the number. If none of them went into it, the number was prime:

```
bool isPrime(int num) {
  // get the just because rules out of the way:
  if(num==1) return false;
  if(num==2 || num==3) return true;

  if(num%2==0) return false; // even number greater than 2

  int i=3; // start dividing by every odd number up to half:
  while(i<n/2) {
    if(num%i==0) return false; // a number goes into it
    i+=2; // next odd number
  }
```

```
  return true; // nothing went into it
}
```

The actual loop is a boring 1, 3, 5 . . . counter. Notice how it uses the early return trick: the function "always" returns true, but only if the loop finished with nothing going into your number.

A neat thing here is the special cases (1, 2, 3 and even #'s) are half the function. That's pretty common, to have lots of "brush clearing" before starting a nice, simple idea like that loop.

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. Some computers will crash the program (a red error that quits the loop. Yay!) Others will hit a special Infinity value and run forever. I've seen ones where `num` wraps around to super-negative, which also happens to quit the loop.

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; // start at 1 because of how the loop works
  // count how many rolls until we get a 6:
  while(Random.Range(1,7)!=6) rolls++;
  print(rolls);
}
```

The die roll isn'r even saved, since I don't care what it is unless it's a 6. Cramming the roll inside the `while` is sneaky, but semi-common and easy enough to read, after you've written a few hundred loops like this.

Another note: this is very easy to mess up. If I used `Random(1,6)`, forgetting the "can't roll highest" rule, it would never roll a 6, spinning 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*. We did this a little before, but didn't need it until now. An example:

```
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) {
    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)`.

The reason we like **for** loops is because they let us gather the loop motion at the top. 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.

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 do that. This one moves the ordinary `cow` variable from 5 to 8:

```
int cows;
for(cows=5; cows<=8; cows++ )
  print( "cow  "+ cows + " says moo" ); // 5,6,7,8
```

The same as a while-loop, it quits when we go past. So `cow` is 9 after this loop ends.

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 `nm=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++) {
  float pos=12.5f+0.8f*i; // <- this is one pretty formula
  print("do something using " + pos);
}
```

It's easy to see there are five, with no possible problems with float rounding cutting one off. And the formula looks fine written on a line like that. We start the count at 0 so the first equation make 12.5 with nothing added.

It's also easy to adjust to go down. We keep the 0 to 4 count and change to `12.5f-0.8f*i`. There's no worrying about `<` or `>`.

For fun, suppose we don't know how many we want. We want to go from 12.5 to at most 20 and don't want to do the math. The int plan still works:

```
bool done=false;
for(int i=0; !done; i++) {
  float pos=12.5f+0.8f*i;
  if(pos>20) done=true;
  else {
    print("do something using " + pos);
  }
}
```

This is a bit clunky, but the way it moves is interesting. It's counting up from 0 by one's, mixed with the stop-when-done trick. I won't blame you if you want to use a simple float loop instead.

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);
```

This runs exactly the same as a while loop for most numbers. If `n` is 5, this prints 5, 6, 7, 8, 9, the same as a regular loop. The only difference is when the `while` loop would run 0 times. For `n=20`, the prints 20 then quits (a `while` loop would just quit.)

You never need to use one of these – I think they so rarely give such a tiny improvement it's not worth cluttering your thoughts. But they're in many languages and it's sort of fun to see features we thought would be more useful than they are.

If you get a chance, look up a repeat-until loop.