Chapter 18

Function examples

18.1 Introduction

This chapter is all function examples and tricks, plus some new Unity functions that let us do neat things. But no new C# rules.

One style for writing short functions is to change the input into the answer. It saves you having to make an extra variable, and often looks fine. Here's the `clamp` function written in that style (recall if forces the first input to be between the other two.) `n` is the input and the output (after we fix it):

```
float clamp(float n, float min, float max) {
  if(n>max) n=max;
  else if(n<min) n=min;
  return n;
}
```

Another style is putting all the math on the return statement. Here's my old `square` function, and a rewrite of the `lerp` function (remember `lerp(3,9,0.15f);` computes 15% from 3 to 9):

```
float square(float n) { return n*n; }

float lerp(float begin, float end, float pct) {
  return begin+(begin-end)*pct;
}
```

Don't make things extra-short for no reason. But in this case, a 1-line function with the equation in the `return` is a nice way to let us know the function is just computing an equation.

Another use for this is value-returning front-ends. Here's a function to always clamp the input from 1 to 10, which uses the real clamp:

```
float clamp1to10(float n) { return clamp(n,1,10); }
```

Putting it on one line like that makes it easy to see that all this function does is let you run the real `clamp` with 1 and 10 already fllled in.

Another example, suppose we already have `bool closeEnough(a,b)` (true when they're within 0.01 of each other.) We can use that to make an almost zero function:

```
bool almostZero(float n) { return closeEnough(n,0); }
// ex:
if( almostZero(amount)) print("out of milk");
```

The same trick works on made-up non-mathy functions. Here's a story-telling function, then a shortcut with the name filled in:

```
// two-input story function:
string adventure1(string name, string animal) {
  string ans = "One day "+name+" took the "+animal+" for a walk on the beach.";
  return ans;
}
```

Suppose we use this a lot with the name `"you"`. We might write a front-end to `adventure1` with `you` built into it:

```
// short-cut using adventure1:
string youAdventure1(string ani) { return adventure1("you", ani); }
```

`youAdventure("cow");` tells a story about you and a cow.

We often mix the tricks. This `closeEnough` has the equation on one line, and uses function `abs` (absolute value) to do the work:

```
bool closeEnough(float a, float b) {
  return abs(b-a)<0.01f;
}
```

Using a less-than in an equation looks really strange. But it's a `bool` function, and the equation gives a true/false, so it all matches.

You're allowed to use a function call in the argument to another function call. For example, this uses `max` to find the largest of 3 numbers:

```
float max3(float a, float b, float c) {
  return max( max(a,b), c);
}
```

As usual, this runs "inside out." The inside `max` finds the largest out of `a` or `b`, then the outside `max` compares that to `c`.

Just so you know, overloading rules would let us reuse the name `max` for this, but I thought that looked too confusing.

18.2 Change me type functions

A lot of value returning functions look like they change you, but, of course, they can't. For example:

```
int cows=14;
clamp(cows,0,10);
print(cows); // 14 ??  why wasn't this clamped to 10?
```

The confusion is, in our minds, `clamp(n,0,10)` forces `n` between 0 and 10. But what it really does is compute that number and return it. Like every other function, we have to "catch" the result:

```
int cows=14, c2;
c2=clamp(cows,0,10);
print( cows + " " + c2 ); // 14 10
```

Often, you just want to say "make `cows` be between 0 and 10." To do that, use `cows=clamp(cows,0,10);`. That's a little like `n=n+1;`. It says "change `cows` into a 0-to-10 version of `cows`."

`n=abs(n);` is another version of that – it makes `n` be positive.

18.3 Dice functions

Functions using random feel different, so I'm putting them in their own section. None of these are all that special. Just examples of how, if you use `Random.Range` a lot in a certain way, you can write a function for it.

Rolling a 50% chance isn't that long, but if we use it a lot, we could turn it into a function. A funny thing is it takes no inputs. `heads()` is true half of the time:

```
bool heads() { return Random.Range(0.0f, 1.0f)>0.5f; }
```

Now we can say `if(heads())`.

Rolling a percent, like a 20% chance, is also short but awkward. Here's a common function to make that easier. It uses 1 to 100, so you can write `chance(20)` for a 20% chance:

```
bool chance(float pct100) { return Random.Range(0.0f, 100.0)<=pct100; }
```

We can use this like `if(chance(20))`. I put a 100 in the variable name to remind me the percent was out of 100, and not the usual 0-1.

Sometimes we want get a random plus/minus value, for example -0.1 to 0.1, or -2 to 2. We can make a shortcut function for that:

```
float randPM(float plusMinus) {
  return Random.Range(-plusMinus, plusMinus);
}
```

If we want about 50, we can use `50+randPM(2);`. Or if we want red to be about 0.4, we could use `0.4f+randPM(0.1f);`.

If we have a board game, we might roll two 6-sided dice a lot. The integer version of 1-6 is `Random.Range(1,7)` (it won't roll the highest, just because.) We can write a function to roll that twice and add:

```
float roll2d6() {
  return Random.Range(1,7) + Random.Range(1,7);
}
```

18.4 More style; types of functions

There are a few ways we do and don't like to write functions. These aren't rules – just things that seem to work better.

A slop-tastic but perfectly good function is about moving things out of the way. The `resetToLeft` from a few chapters ago is like that. It changes a handful of global variables and moves us, which is pretty sloppy, but it's still better than having those lines copied in four places.

A common form of this is a 1-use function who's only purpose is to break up something long. This sort of thing is common:

```
void Update() {
  doAllMoves();
  updateColorChange();
  if(difficulty>1) updateMoveMonsters();
  updateCheckWinLose();
}

void doAllMoves() { ... }
```

It's pretty much understood that `doAllMoves` will be a sad excuse for a function. It probably reads and sets a bunch of globals, may crash if not called from that exact spot in Update, and it's not making your program shorter since it's only called from one place.

But it's better than a super-long Update, so it's fine.

To make nicer functions we try not to read from global variables, and use inputs instead. If you remember my first color function, `ApplyColor();` had to read from `r`, `g`, `b` globals. We didn't have any other way. The later `setColor(r,g,b);` function is clearly better.

Suppose we sometimes, but not always want `setColor` to avoid making us too dark. One way to add that would be as a globalt:

```
bool colorCantGetDark=false; // change this to change how setColor works

void setColor(float r, float g, float b) {
  if(colorCantGetDark) { ... } // fix r,b,g if too dark
  // set color here
}
```

This seems nice since maybe we hand-set `colorCantGetDark` at the start of each level. Then we can call `setColor` the normal way.

But there's a 100% chance we'll use this somewhere and forget about the global setting. Or temporarily change it in some obscure place and forget to set it back. We'll be screaming at our cat about this stupid function that won't print dark colors.

Adding "not too dark" as a 4th input, or writing 2 versions of the function may not be quite as slick, but it's a lot safer.

Moving on, functions that return values almost never do anything else. For example `bool isOffEdge()` shouldn't also push us in bounds. It feels like a math-type function, so we expect it to give an answer and do nothing else.

Anything extra we call a *side effect*. That's a snarky way of saying the return value is the main thing, and you're not supposed to confuse us by also changing globals.

A confusing but common function type is when its main purpose is to do something, but it has a minor return value, usually saying if it worked. For example, a function that moves, maybe wraps-around, and returns true if it did:

```
bool moveAndWrap(float xMove) {
  // true means we had to wrap-around
  x+=xMove;
  if(x>7) { x=-7; return true;}
  if(x<-7) { x=7; return true; }
  return false;
}
```

We can use it like normal, just: `moveAndWrap(0.1f);`. Or we can also "catch" the answer:

```
bool didALap = moveAndWarp(0.1f);
if(didALap) { // do lap stuff
```

In our minds, we're calling `moveAndWrap(0.1f)` to move us, and the answer is a little bonus.

Even more confusing, we usually use it dirctly inside the `if`:

```
if(moveAndWrap(0.1f)) { // do lap stuff
```

This seems pretty nuts at first – we're moving ourselves inside of an `if`. But we still think of it as really a do-something function, merely wrapped in the `if` to check one extra thing about it.

18.5 Moving curve examples

Now that we know functions, we can do some new things with more of the Unity built-ins, like moving in a curve.

This works best in a top view, which we can set up pretty easily. The Cube will start at (0,0,0), with the Camera looking straight down at it.

Put the camera at position (0,10,0) and rotation (90,0,0). As usual, can just select MainCamera then enter those numbers in the Inspector. That puts it 10 units over 000, tilted to look down. Camera view (the Game window, or selecting the Camera) should now show just the built-in gray floor, with a centered white Cube.

The last part is making it so we can tell the front of the Cube from the sides and back. A cheap way is gluing a ball to the front. Create a Sphere. Without worrying where it is, go to Hierarchy and drag the Sphere onto the Cube (like you were putting it in a Folder.) If it works, Sphere should be under Cube, indented a little, and Cube should have a new toggle-triangle. In 3D programs, this is called **childing** – Sphere is now a child of Cube, and acts like it's glued to it.

The last step is to move Sphere to the right spot. Change the Inspector position for Sphere to (0,0,0.5). That should make it stick 1/2-way out of the front, so we can see which way the Cube is aimed. The exact shape you use and exact position won't matter. The rest of the example works no matter what.

Before we use them for real, let's play around with 2 Unity built-in move functions. The first is `transform.Rotate`. It has `void` for the return value, which means we call it by itself and it does something.

Here's a small program using `Rotate`:

```
void Update() {
  transform.Rotate(2,0,0);
}
```

Running should make the Cube spin end-over-end, like it's trying to roll to the top. Since we have an overhead view now, the top of the screen is like

North or Forward. If we change the call to `(0,2,0)`, the Cube should act like a spinner. Then if we use the third slot – (0,0,2) – the Cube should appear to be trying to roll sideways.

As a check, you can try (0,1,0), and see it spins 1/2 as fast. Or try (0,-2,0) and see that it spins counter-clockwise instead of clockwise.

The other function is `transform.Translate`, which moves you. Like `Rotate`, it takes three inputs and has a `void` return. The inputs are obviously the amount of movement along x, y or z. The really cool thing is it depends on which direction you're facing. Here's a small program testing **Translate**:

```
void Update() {
  transform.Rotate(0,2,0);
  transform.Translate(0,0,0.1f);
}
```

This spins while moving forward (the Cube's personal forward,) so makes a circle. Spinning faster or moving slower would make the circle tighter.

If you change to `Translate(0.1f,0,0)`, you'll see it spin while moving sideways, also making a circle (it's always moving right, facing inward.) Using `(-0.1f,0,0)` will spin the other way, moving left.

Moving up (`Translate(0,0.1f,0);`) will just move straight up, spinning, until it's above us (we're looking down.) So that's no good.

The difference in the numbers shows how you need to read what the parameters stand for. `Translate` assumes x, y and z are in regular distance units. Our screen is still about 14 wide (-7 to 7,) so `0.1f` every frame is pretty fast (across the screen in 140 moves.)

`Rotate` uses degrees, which is out of 360, so 0.2 is super-slow, but 2 isn't bad.

We need to fix one more problem: we don't know where we are anymore. So one last new rule: we know `transform.position` is where we are. Previously we shoved our `x` and `y` variables into it. With this new movement method we're going to read from it: `transform.position.x` and `transform.position.z` looks up our current position (x and z since it's a top view. y is always 0.)

Here's top-view out-of-bounds code, written as a function, using the new rule. We'd use it like `if(outOfBounds())`:

```
bool outOfBounds() {
  float x=transform.position.x;
  float z=transform.position.z;
  if(x<-7 || x>7 || z<-5 || z>5) return true;
  return false;
}
```

Now we have enough to try some clever stuff. This starts us with a random facing, moves in long curves, and bounces off the walls:

```
void Start() {
  // spin random 360 degrees once at start:
  float ySpinAny=Random.Range(0,360);
  transform.Rotate(0, ySpinAny, 0);
}
void Update() {
  // spin slowly and move forward (same code as before):
  transform.Rotate(0,1,0);
  transform.Translate(0,0,0.1f);

  if(outOfBounds()) // turn around when out-of-bounds:
    transform.Rotate(0,180,0);
}
```

This is nice, but gives us boring, predictable curves after watching it for a while.

We can change it up with the usual tricks: turn the curve into a variable; reroll it at random; and give the 180-degree flip some slight randomness:

```
float spin=1; // how fast we turn, as we move

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
}
void Update() {
  transform.Rotate(0,spin,0); // <- not always the same curve
  transform.Translate(0,0,0.1f);

  // 4% chance to change direction a little, making a curvey path:
  if(Random.Range(0.0f, 1.0f)<0.04f)
    spin=Random.Range(-1.0f, 1.0f);

  if(outOfBounds()) {
    float turnAround = Random.Range(180-30, 180+30); // 180 +/-30
    transform.Rotate(0,turnAround,0);
  }
}
```

This moves in semi-interesting curves, with fun bounces.

Since we don't force it in bounds, it has a bug: it can rarely go out-of-bounds too far and get stuck spinning every move, trying to get back in. We can't fix it yet, by forcing it to always be in bounds, but can later.

Turning and always going forward looks nice, but for fun, we can try other things with `Translate`. If we move sideways and forward, we'll be going diagonally. If we use a counter we can move left 10 times, right 10 times, left 10 times ... we'll zig-zag. I'm getting rid of the curve for this one:

```
float zigSpd=+0.05f; // sideways speed; alternate + and -
public int zigLen=14; // how long each zig lasts
int zigTimer; // counts up to zigLen as we move

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}
void Update() {
  // forward and left or right:
  transform.Translate(zigSpd,0,0.1f);

  zigTimer--;
  if(zigTimer<0) {
    zigSpd*=-1; // flip left/right
    zigTimer=zigLen; // restart the count
  }

  if(outOfBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

This works – it moves in short, straight lines making a zig-zag – but I just don't like the way it looks. I even played with `zigSpd` and `zigLen` a little, but I still don't like it. This happens. Real programmers try then throw away lots of code.

I think the real problem is that it should face the way it moves. My second try at zig-zagging is to always move forward, and snap change our facing: turn 30 degrees right, move some more, then 30 degrees back left . . . :

```
public int zigLen=35; // how long each zig lasts
public float zigDegs=30.0f; // degrees of each zig
int zigTimer; // counts up to zigLen


void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}
void Update() {
  transform.Translate(0,0,0.1f); // forward
```

```
  zigTimer--;
  if(zigTimer<0) {
    zigTimer=zigLen; // reset the count
    transform.Rotate(0,zigDegs,0); // do a zig
    zigDegs*=-1; // zig the other way, next time.
  }

  if(outOfBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

I like how the plan for this one is the same, but different. We still count down until the next zig. But now the direction change is a 1-time thing – just a rotate. The movement part is back to the old always-straight-ahead way.

18.6 Reading keys

We finally know enough rules to decipher a built-in keyboard reading function. This particular one is Unity's, but others are mostly the same.

Here's a small program that lets keys A and D move a Cube left and right. It would go on the Cube in the top-view scene from the last section (you should be able to adapt it to a front view, if you like):

```
float x=0;

void Update() {
  if( Input.GetKey(KeyCode.A) ){
    x-=0.2f;
    if(x<-7) x=-7; // stay in bounds
  }
  if( Input.GetKey(KeyCode.D) ){
    x+=0.2f;
    if(x>7) x=7; // stay in bounds
  }

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

You can probably figure out that `Input.GetKey(KeyCode.A)` is true when A is pressed. Here are some notes, mostly obvious, about it:

- The name of the function is `GetKey`. It's in the namespace `Input`. So, altogether: `Input.GetKey`.

- It returns a `bool` – true means that key is down. Like any `bool` function, you can use it inside of an `if`.
- The input is which key you're checking for. `KeyCode` is just an enumerated type, specially made to give names to all the keys. For example, `KeyCode.DownArrow`.
- There's also an overload which takes a `string` as input. For example, `Input.GetKey("a")`. I don't like that as much, since you have to remember upper/lower case, and the names of special keys (in the enumerated type, they just pop up after the dot.) Plus, we know it's just a front-end for the other one.
- There are three ways to check a key: just pressed, currently being held, or just let go. In Unity, these are `GetKeyDown`, `GetKey` and `GetKeyUp`. Most sorts of keys or virtual buttons have those three things. A little thinking will usually let you figure out which one you want.

  In this case, GetKey – while being held – is what we wanted. GetKeyDown would make us tap-tap-tap in order to move.

Here's a different version, which also moves back&forth, but using `GetKeyDown`. Tapping A or D starts you moving left or right. Tapping S stops you. I don't like this as much, but it's fun to program:

```
float x=0;
float spd=0; // set to 0, +0.2 or -0.2 by keypresses

void Update() {
  if( Input.GetKeyDown(KeyCode.A) ) spd=-0.2f;
  if( Input.GetKeyDown(KeyCode.D) ) spd=0.2f;
  if( Input.GetKeyDown(KeyCode.S) ) spd=0;

   x+=spd;
   if(x>7) { x=7; spd=0; }
   if(x<-7) { x=-7; spd=0; }

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

You might notice that the movement part is just a copy of old movement code. Previously, we had to use little tricks to change the speed. Now we can have key-presses change it.

We can steal some more old code. From way back, having the object speed up and slow down looked nice. We can have it so holding a key gradually speeds you up:

```
float x=0;
float spd=0; // changed by holding down a key
public float accel=0.005; // how much speed changes in one tick

void Update() {
  if( Input.GetKey(KeyCode.A) ) spd -= accel;
  else if( Input.GetKey(KeyCode.D) ) spd += accel;
  else {
    // if not holding a key, slow down, then stop:
    spd *= 0.98f; // slow down (0.98 is trial and error)
    if( spd<0.02f && spd>-0.02f) spd=0;
  }

   x+=spd;
   if(x>7) { x=7; spd=0; }
   if(x<-7) { x=-7; spd=0; }

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

The first two `if`s don't limit the speed. In this example, you hit an edge before you go too fast, so it's fine. But adding `if`s to limit it would be no problem.

The part where it slows down is an old movement trick. We want to make the speed go towards zero, whether it's positive or negative. Multiplying by 0.98 makes 1 and -1 both go towards zero. It's only 2% smaller, but it adds up fast. The last slowdown line says "when speed gets really small, just be 0."

We can also reuse the turn-and-move-forward code to use keys. I'll have A and D rotate, while W moves forward:

```
void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

  if( Input.GetKey(KeyCode.W) ) transform.Translate(0,0,0.1f);
}
```

We could change this version so the W key is acceleration:

```
float spd=0; // forwards speed. always positive

void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

 // speed up if W held, slow down if not:
```

```
  if( Input.GetKey(KeyCode.W) ) {
    spd+=0.01f;
    if(spd>0.14f) spd=0.14f;
  }
  else {
    spd-=0.005f;
    if(spd<0) spd=0;
  }

  // spd is often 0, which is fine. This won't do anything then:
  transform.Translate(0,0,spd);
}
```

Moving on, a fun thing is coding so the Cube crawls around the sides of the screen. It's fun since it seems difficult, but then you realize we can easily slide only along the bottom, or top (just change z from -5 to 5) or only along one side. The hard part is a design that puts all 4 slides together.

So this is an excuse to write some fun `if`s.

The basic plan is using a 0-3 `int` to remember which side we're on. This is a good place to make a small enumerated type (remember this is just a fancy way to say `Wall` means int, and can be 0 to 3):

```
enum Wall {up, right, down, left};
Wall wall = Wall.down; // sample use
```

`Wall wall` looks funny, but works: capital-W `Wall` is the name of the type, and lower-case `wall` is the variable name. People do that – if you can't think of a good name for a variable, just lower-case the first letter. It's sort of like `float f;`.

I'm going to handle each side in it's own `if`, using `wall` as a state-variable. So there are four `if`s, and each one knows how and when to go to the two walls it touches.

Movement is only 2 directions: clockwise or counter-clockwise; A key or D key held down. If we hold down A it should more left to right across the bottom, up the right wall, `left` along the top, and `down` the left wall. That's why the top and left walls move backwards:

```
float x=0, z=-5; // middle of bottom wall

void Update() {
  // get direction of move:
  int mv=0; // -1 0 or 1. Direction based on bottom wall
  if(Input.GetKey(KeyCode.A)) mv=-1; // A=clockwise
  else if(Input.GetKey(KeyCode.D)) mv=1; // D=counter-clockwise
```

```
  // handle move for each wall:
  if(mv!=0) {
    float mvAmt=0.1f*mv;

    if(wall==Wall.down) {
      x+=mvAmt; // note: + for D, - for A
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.up) {
      x-=mvAmt; // top moves moves backwards from bottom
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.right) {
      z+=mvAmt;
      if(z>5) { z=5; wall=Wall.up; }
      else if(z<-5) { z=-5; wall=Wall.down; }
    }
    else if(wall==Wall.left) {
      z-=mvAmt; // left wall: forwards=down
      if(z>5) { z=5; wall=Wall.up; }
      else if(z<-5) { z=-5; wall=Wall.down; }
    }
  }
  transform.position = new Vector3(x,0,z);
}
```

The enumerated type really was helpful here. While writing this, I messed up some of the numbers and directions. Seeing lines like `if(wall==Wall.right)` made it easy to see I was in the correct section (it's common, for me anyway, to test, find a problem, and then "fix it" in the wrong part of the code, making things even worse.) Little things to make the code easier to read really helps.

18.7 Making a real function library

So far, I've been suggesting that all of your useful functions should be pasted into each new script. That will work, and we did it back in the day, but there's a nicer way. Almost all languages and environments let you spread your program over several files. Common, similar functions are usually placed in a file by themselves.

Some environments need you to list or add all the files to your code Project. Unity3D automatically finds them for you. All you need to do is write a "function file," and you can use it right away.

The steps to putting functions by themselves are:

- Make a new C# script the usual way (`Project -> CreateC#script`; double-click to open in the editor.) The name doesn't matter.
- Delete everything past the first two `using` lines.
  All that stuff is Unity set-up for things that go on a gameObject. we're making something different.
- Pick what you want to call the namespace. I'm calling mine `rand`. This does not have to match the file name.
- Set up the namespace like below. You might notice it's a simplified version of what we had before:

  ```
  class rand {

  }
  ```

- Write your pure functions inside of it, with `public static` in front of each one (ex below.)
  By pure, I mean they can't use any globals (which seems pretty obvious, since how would they know what globals your other program has) and they also can't change position, color or size.
- Don't put it onto a gameObject (the system won't even let you.) Just having it be created is enough. The system will find it.
- In any script, use the functions through the namespace. Ex: `int n = rand.twoD6();`

Here's my final version:

```
using UnityEngine;
using System.Collections;

class rand {
  public static int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }

  public static bool heads() { return Random.Range(0.0f, 0.1f)>0.5f; }

  public static float tweak(float n, float plusMinus) {
    return n+Random.Range(-plusMinus, plusMinus);
  }
}
```

This makes functions `rand.twoD6`, `rand.heads` and `rand.tweak`. They all start with the namespace `rand` (which is just a name I picked.) A fun thing is, if you go to some other script and type `rand.` (rand, then a dot,) the pop-up will show you those three functions.

Any of our scripts can use `if(rand.heads())`.

This trick shows off the idea of functions. The file we made has no `Start` or `Update`. By itself, it will never run or do anything. Other people's `Start` or `Update` might use some of them, or all, or none. That's fine – they're just there if we want to use them.

In case you know a little about classes, this does **not** make a class. This is a total abuse of the word `class`, using the magic words `public static` to make a namespace with stand-alone functions. It's a common C# trick, and later we'll see `public` and `static` as perfectly good (but advanced) programming ideas.

18.8 Unity built-ins

Unity has already done the namespace-file trick for lots of common functions. `Mathf` groups lots of them. `Random` was made using this trick, to hold the two versions of `Range`. If you look up the official Unity `Mathf` scripting reference, the functions are listed under **static functions**. Now we know why it uses those words (we don't know exactly what a static function really is yet – we'll see that later.)

For example, `Mathf.Approximately(a,b)` is the close-enough function. Likewise, `Mathf.Abs`, `Mathf.Clamp` and `Mathf.Max` are in there, and do the same thing as my examples. For example, `float n=Mathf.Max(a,b);`