Chapter 11

Functions, part I

This section is an introduction to things called **functions**. You may have heard them called by different names – methods or procedures – they're all the same thing. They work about the same in all languages, and C# uses pretty much the same rules as everyone else.

The basic idea is you can write program lines somewhere else and give them a name. Then anyone can use that name to run those lines. It's merely an organizational trick – we won't be able to do anything we couldn't before – but it's a really useful trick for making larger programs.

`Start()` and `Update()` are functions. The difference between them and the ones we'll write is Unity won't run ours automatically. Our functions will just sit there until Start or Update need to use them.

`print("abc")` is running a function. For now, ours will be even simpler, for example `printHello();` will always print `hello`.

Just so you know, the stuff in this chapter is going to seem a little limited. It's only the first part of the rules for functions. But we'll still be able to do helpful things.

11.1 Intro Function examples

Here's a program with a `printHello` function. It does nothing, but it's legal:

```
public class testC : MonoBehaviour {
  void Start() {}

  void Update() {}

  void printHello() {
    print("hello I say");
  }
```

```
}
```

Look how nicely `printHello` fits in there with it's friends Start and Update. For fun, go to the code editor and play with the -/+ box (left side) to hide/show it. The editor knows this is named `printHello` and it owns the stuff in the curly-braces.

It never runs because we never tell it to. The system has a special rule to auto-run Start and Update. But the general rule is functions just sit there, waiting for their names to be called.

If you remember from way, way back, I wrote that `void Stork()` instead of Start wasn't an error, but wouldn't run:

```
// another legal program that does nothing
public class storkTest : MonoBehaviour {
  void Stork() {
    print("starting program");
    print("ending program");
  }
}
```

The computer thinks you wrote a program with no Start or Update, which is legal, and one function named `Stork`. Stork is a perfectly good function – it just doesn't have a special auto-run name.

To run a function, use the name. To avoid confusing it with a variable, add a paren-pair at the end. This uses printHello, twice:

```
public class testC : MonoBehaviour {

  void Start() {
    print("What does the alien say:");
    printHello();
    print("Again, again!:");
    printHello();
}

  void printHello() {
    print("hello I say");
  }
}
```

The system still runs only Start. But now Start runs `printHello`. The output is:

```
What does the alien say:
```

```
hello I say
Again, again!:
hello I say
```

Of course this is useless so far. The only important thing is how Start jumps to the function and runs the line, then Start keeps going, jumping back to the function a second time. And the whole time we count as running Start line-by-line.

Here's another with two functions. I'm putting one of mine in front of Start, just to show it can be done. The computer doesn't care about the order, since it looks them up by name:

```
string w;

void addZsToBack() {
  w=w+"zzz";
}

void Start() {
  w="frog";
  addZsToBack();
  w+="#";
  addZsToBack();
  print(w); // frogzzz#zzz
}

void addFrontD() { // this is never run (since Start never calls it.)
  w="D"+w;
}
```

The basic run is the same as the printHello example. Start runs, and calls a function twice.

An interesting thing is we never use the second `addFrontD` function. That's fine and not an error or even a problem. The other interesting thing is how `addZsToBack` uses an assignment statement and a global variable, which is also fine.

11.2 Function rules

As usual, it's nice to summarize the rules and notes, in one place. There's nothing really new here:

- Our new functions go "next to" `Start` and `Update` – just inside the big outer `testC : Monodevelope` { } curly-braces; but outside of all other functions.

- The order of functions never matters. You could move `Start` to after `Update` and the program would work the same. You can call any function no matter if it was written before or after you. You can use cut&paste to rearrange functions later on.

- The function name is any legal identifier (the same rules for variable names.) `do_stUff34` is a legal, bad, function name. Like variable names, they're case-sensitive (`printhello();` would be an error, on account of the lower-case H.) They don't need to start with a capital letter. Unity just thought it looked nicer for Start and Update.

- For now the heading is always `void`, the name, then empty `()` parens. Eventually we'll put different things in there, but not this chapter. For now the function call is also always empty `()` parens.

- The body of a function goes inside of { and }. It can have any statements and any number of them: assignment, print, `if`'s, declaring local variables.

Functions normally just sit there, waiting to be run when we need them. Here's the obligatory terrible analogy:

Imagine a cookbook. There's a section on buttering a pan, sifting flour, browning onions – those are like functions. Now suppose you're making lasagna. You go to that recipe, follow it, and you're done. But, the lasagna recipe says "butter a 9 inch pan (page 126,)" which you do. Then later "brown 3 large onions (page 428,)" and you do that. Then near the bottom, it's back to "butter a caserole dish (page 126.)"

Each time it does that, you save your spot, flip to the page it says, follow those instructions, then back to where you were in the lasagna recipe. We never looked at the flour sifting part of the cookbook, since it never told us to.

If you think about it, cookbooks are pretty old and those guys worked out a good system. Computer programs would be crazy not to copy it.

Again, just to have them, here are the written-out rules for running functions:

- The technical term for running a function is **calling** it. You call a function by using the name, followed by `()`. In computer talk, `printHello();` is a function call.

- Function calls count as statements.

- Function calls always return to just after where you called them. A call jumps to the function, runs lines from the top, then jumps back when it hits the end.

- It's fine to have extra unused functions.

- Currently we can't mix a function call as part of a longer line. For example `q=printHello()+"abc";` is junk. `printHello();` all by itself is the only way to use them, for now.

11.3 More function examples

I promise functions will be useful, but this is more practice with the rules.

Let's start with two functions and a Start which doesn't use either. This prints `x`:

```
void A() { print("this is A"); }

void B() { print("I am B"); }

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

If we changed `Start` to be like below, the program would print `"I am B"` three times. The first two will be on different lines because the program doesn't care about where you put returns and spaces (the last one is also on a different line, but you knew that):

```
void Start() {
  B(); B(); // put two function calls on same line, just for fun
  B();
}
```

We can mix it up however we like, or put them inside `if`s:

```
void Start() {
  int x=0;
  B();
  if(x>0) {
    A();
    B();
  }
  x=5;
  if(x>0) { A(); A(); }
}
```

Notice how the function calls in the first `if` are indented like regular statements (since they are.) They should look pretty natural in there. As usual, we only do them if `x` was more than zero (it isn't, so we never run them.)

Then the second `if` used the "combine short stuff on 1 line, but don't forget the {}'s" style. I like how the two function calls look just like regular statements in there.

This whole thing runs B, skips the first `if`, then runs A twice.

Anyone is allowed to call a function – we can call one from `Start` and also from `Update`. Here's a simple example where `Start` calls a function, and `Update` uses a counter to call it a few times:

```
int x=0;

void A() { print("this is A"); }

void B() { print("I am B"); }

void Start() {
  B();
  print("end of start");
}

void Update() {
  if(x<6) A(); // first 6 times
  if(x<3) B(); // only first 3 times
  x++;
}
```

`Start` calls `B` and is done. Then Unity-magic runs `Update` over and over, same as it always does. As usual, `x` will be 0,1,2 .... . Update calls `A` then `B` for a while, then just `A`, then nothing. Output will be:

```
I am B
end of start
this is A
I am B
this is A
I am B
this is A
I am B
this is A
this is A
this is A
```

Functions are allowed to do anything `Start` or `Update` could do. This function steps `cows` from 0 to 100 then wraps. The result is that Update Moo's once every 100 frames:

```
public int cows=0;

void doOneStepCowWrap() {
  cow=cows+1;
  if(cows>100) cows=0;
}

void Update() {
  doOneStepCowWrap();
  if( cows == 0 ) print("moo");
}
```

This is looking more useful. Pretend you are *so sick* of looking at add-and-wrap-around lines. In Update we only have to see `doOneStepCowWrap`, which we can assume does one step of the cow wrap-around. This is also our first function with an `if`. Pretty exciting stuff.

A common real use of functions is to hide ugly stuff. I'm not a fan of the long GetComponent color-changing line. My plan is to use globals `r`, `g` and `b`. To set color I'll merely change those globals and call a function to run the line I hate typing:

```
float r, g, b; // global color variables

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

Now we never have to see `GetComponent` again. It's like we made a new command named `applyColor();`. A program fragment using it:

```
void Start() {
  r=0.4f; g=1; b=1; // light aqua
  applyColor();
}

void Update() {
  ...
  if(laps==5) { r=1; g=0; b=0; applyColor(); } // red
}
```

This is clunky, and we'll make it much better in a later chapter. But I think even this is an improvement, especially if we change our color a lot.

We can do something similar with the position command. Assuming we're using global `x` and `y`, we can make it so `applyPos();` runs the real line for us:

```
int x=-7, y=0; // the position variables we always use

void applyPos() {
  // assumes global x and y set to where we should be
  transform.position = new Vector3(x, y, 0);
}
```

Now we can move with `x=0; y=2; applyPos();`.

11.3.1 Longer resetToLeft example

In this section, I want to have an example of a function we just naturally make as we're writing a program. I want to start with a semi-interesting program that shoots a ball with a curve, using `xSpd` and `ySpd` tricks.

Here's the program. So far, it's just tricks with movement code, and no functions yet (well, I'm using `applyPos`, from above, since I like it):

```
float x=-7, y=0;
float xSpd=0, ySpd=0;

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f; // ball speeds up

  if(x>7) {
    x=-7; y=0;
    xSpd=0;
    ySpd=Random.Range(-0.3f, 0.3f);
  }
  applyPos();
}
```

I haven't done exactly that before, but it's just old stuff combined. `xSpd` is increasing, starting from zero, so it looks like it's falling from left to right. And `ySpd` just makes it move up or down. Combining them gives a curve like tossing a watermelon off a roof, except going from left-to-right.

Now onto the function part. I want to pull the reset code from out of Update and into a function. This code has the same lines as above, just moved around:

```
void resetToLeft() { // these 4 lines used to be in update
  x=-7;
  y=0;
  xSpd=0;
  ySpd = Random.Range(-0.3f, 0.3f);
}

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f;

  if(x>7) resetToLeft(); // this used to be 4 lines in {}'s
}
```

It turns out that, depending on the y speed, the ball could fly off the top, or the right side, or the bottom. I'd like it to check for all 3, changing into a different color. The code for that is simple, and uses our new functions. The good part is in Update:

```
float x, y;
float xSpd, ySpd;
```

```
float r,g,b; // set these then call applyColor

void Start() {
  resetToLeft(); // may as well use these functions to set it up
  applyPos();
}

void Update() {
  x += xSpd; y += ySpd;
  xSpd += 0.01f;

  if(x>7) { // fall off right = blue
    r=0.4f; g=0.4f; b=1; applyColor();
    resetToLeft();
  }
  else if(y>3) { // fall off top = yellow
    r=1; g=1; b=0; applyColor();
    resetToLeft();
  }
  else if(y<-3) { // fall off bottom = red
    r=1; g=0.3f; b=0.3f; applyColor();
    resetToLeft();
  }
  applyPos();
}
```

We could have written that without functions. Every function call would be the lines inside of it, pasted in. It would be longer and harder to read. That's the only problem functions solve – avoiding cut&paste.

Functions also help keep things consistent. Suppose we needed to change where `x` starts on a reset. `resetToLeft` is a single spot to change that. Without that function we'd need to hunt down all 4 places and change them.

11.4 functions with more math

Functions are allowed to have all sorts of program lines in them, even complex ifs. Here's a function that prints a silly random name, that Start uses twice:

```
void Start() {
  print("Hal starts at:");
  randomTown();
  print("and walks to:");
  randomTown();
}
```

```
void randomTown() {
  // makes things like "Clarkberg" or "New Tomatown"

  string mid="";
  string ending="";

  // use a look-up table to pick "town" "ville" or "berg":
  float endingRoll = Random.Range(0.0f, 1.0f); // using the % and table trick:
  if(endingRoll<0.33f) ending="town";
  else if(endingRoll<0.66f) ending="ville";
  else ending="berg";

  // flip a coin for "Clark" or "Toma":
  float midFlip=Random.Range(0.0f, 1.0f);
  if(midFlip<0.5f) mid="Clark";
  else mid="Toma";

  string townName = mid+ending;

  // for fun, 10% chance to add "New" in front:
  float newCheck=Random.Range(0.0f, 1.0f);
  if(newCheck<0.1f) townName="New "+townName;

  print( townName );
}
```

`randomTown` has got all the stuff that's usually in Update or Start: declaring local variables, ifs, cascading ifs, growing variables. Functions can use it all. It would be weird if they couldn't.

To finish up, it might print this:

```
Hal starts at:
Clarkberg
and walks to:
New Tomaville
```

11.5 Common Errors

As usual, just so we see them, here are some common errors and their messages:

- Forgetting the parens in the function call. Ex: `randomTown;`. The computer thinks you're trying to use the function name like it was a regular variable. It gives the general-purpose error: *Only assignment, call, increment, decrement, and new object expressions can be used as a statement.*

It doesn't say anyhing about missing parens (but at least it tells you the line.)

- Trying to "call" a variable like a function. Ex: `n();`. That gives a pretty cool error: *Expression denotes a 'variable', where a 'method group' was expected.*
  Method is another word for function, so this isn't too bad. It's saying the `()`'s make it expect a function.

11.6 Style

This is mostly a summary of things I've mentioned before:

- Function names that do things usually have a verb in them, like `applyColor`, `resetToLeft` or `setSize`.
- It's fine to use 1-line function to "rename" a command. For example, `void turnYellow() { GetComponent<Renderer>().material.color = new Color(1,1,0);}`.

  Sometimes we call that a wrapper or a front-end, since all it does it pass you along to a real command doing the actual work.
- One type of function is something we think will be of general use. `applyColor` is like this. We might paste it into every new script we start (assuming we always use the global `float r, g, b;` trick. We'll find a better way soon.)
- Another idea for a function is something we'd only use in one program, in several places. `resetToLeft` is an example of this. No other program would use it, but it shortens the one it's in.
- Another type is a function used only once, like `doCowWrap`. All it does is move some code out of Update, which is fine.
- Figuring out what would be good to turn into a function is difficult, and takes a lot of practice. You have to waste a lot of time writing bad ones. You can look up advice about "cohesion" and so forth, but sadly, they only make sense after you've written a lot of bad functions.
  So don't feel like you have to try too hard to make them. If you suddenly think "hey, this could be a function," go ahead and write one.

11.7 Chains of functions

The rule "call the function, run it, come back where you came from" also applies to chains of functions. A function is allowed to call another function. The computer can remember as many return locations as it needs to.

Here's a mechanical example showing a function chain. In this, `A` calls `B` which calls `C`. Printing "start" and "end" in each function is sometimes added for real, to try to track down funny behavior. Here it just makes a nicer example:

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

void A() {
  print("starting A");
  B();
  print("ending A");
|

void B() {
  print("B1");
  C();
  print("B2");
}

void C() { print("C"); }
```

The first call, `C()` is nothing special. It runs `C` and comes back to `Start`.

This next one, `B()`, prints `"B1"`, then jumps to `C` which prints `"C"`, back to `B` which prints `"B2"`. Then finally back to Start. It prints this:

```
B1
C
B2
```

The last function call, to A, has a bigger chain: `Start` calls A, prints and calls B, which prints and calls C, which prints and pops back to `B` then `A` then back to Start. Output is:

```
starting A
B1
C
B2
ending A
```

You may notice that it's just `starting A` and `ending A` wrapped around what B does. Which it obviously has to be.

A way to think about the function-chain rule is, suppose you call function `Dog` that does something you want. Maybe `Dog` also calls `E` and `F`. Maybe `E`

calls a bunch of other functions. You don't have to care. It doesn't matter. All that matters is that `Dog` did what you wanted and came back. What happened inside of `Dog` are just unimportant details.

Suppose we often want to recenter a ball and turn it blue. We could write this function:

```
void centerBlue() {
  r=0; g=0; b=1; applyColor();
  x=0; y=0; applyPosition();
}
```

It didn't need to call applyColor and position – it could have included those lines inside of itself. But it doesn't matter that it used them, either. Calling `centerBlue` does what it should, and that's all that matters.

11.7.1 Multiple local scopes

This section is just explaining that local variables are fine even when one function calls another. The way the computer handles that is sort of interesting, and might help you understand functions better. But this isn't a new rule or anything you need to know for a while.

Here's some code to start with. `Start` calls a function named `abc`, nothing special so far:

```
int x=5;

void Start() {
  int n = 7;
  abc();
  print("start n = "+n); // prints 7
}
```

It seems pretty obvious that last line has to print 7. We didn't change `n`, and it's our personal, local variable, so no one else can even see it. `x` could change, but it's a global, so we expect that.

Suppose `abc` also declares local `n`. That shouldn't matter to `Start`, and it doesn't. But it seems funny that we now have local Start-`n` and local abc `n`, and both at the same time when Start calls `abc`.

Here's the rest of the example, and a picture of how the computer handles it. Not only does `abc` have a local `n`, but it calls another function `def` which has a third local `n`:

```
void abc() {
  string n;
  n="cow";
```

```
  def();
  n += "y";
  print("abc n is "+n); // cowy
}

void def() {
  int n=20;
  n += 5;
  print("def n is "+n); // 25
}

Output (from the original Start):
def n is 25
abc n is cowy
start n = 7
```

Here's a picture of when we get to `def`. It really means that Start and `abc` are waiting, with their own saved local `n`'s:

```
Start |   abc    |  def
----- |  ------  |  ---
n: 7  |  n: cow  | n: 25
```

These are formally called **Stack frames**. Each function (even `Start` and `Update`) gets one. They hold all the local variables for each function. When you call a function, a new stack frame is made for it, local vars are set up, and it's put on top of yours. When a function ends, its frame pops off the stack, leaving yours the way it was.

In the picture, `Start` is waiting for `abc`, which is waiting for `def`, which is running. Everyone's local variables are remembered, and they never conflict with each other.

Again, the important thing: local variables work the way you think they should. Re-using the same name won't mess you up, not even with function calls.

This next thing is really far from things you need to know, but kind of interesting: stack frames are why creating and destroying local variables takes zero time. Before the program runs, the compiler figures out the stack frame for each function, with the exact arrangement of each local variable.

When you call a function, the computer gives you the precomputed amount of space for all local variables. The exact location of each one is already built into the function.

And again, that's not important to know. I just don't want you to be worried about local variable creation and destruction slowing down the program. If they were bad, we wouldn't have invented them.