

Chapter 16

Overloading and default parms

16.1 Introduction

This is another relaxing chapter, with two fun rules that let you reuse the same function name: **default parameters** and **overloading**.

They can make it look like the same function takes all sorts of different inputs, so you can use these tricks to write an `A` so that `A(3)`; `A("goat")`; and `A(3,6)`; are all legal.

But they're just shortcuts and tricks. They don't change the way functions really work.

16.2 Default parameters

A **default parameter** is a shortcut that lets the computer auto-fill parameters in some function calls. Here's an example of my old `setPos` function using them. The only new thing (which I haven't explained yet) are the extra two `=`'s in the heading:

```
void setPos(float x=0, float y=0.5f) {
    transform.position = new Vector3(x, y, 0);
}
```

The first thing to know is if we call it the normal way, like `setPos(5,2)`; it runs the normal way. Those two `=`'s will do nothing, like they weren't even there.

The new rule is that if the function has an *equals-something* after a parameter, we can leave it out of the function call. The computer will fill in that value. `setPos(4)`; matches 4 to `x` and fills in 0.5f for `y`. It's a shortcut for `setPos(4, 0.5f)`;

`setPos()`; fills in 0 for `x` and 0.5f for `y`. It's a shortcut for `setPos(0, 0.5f)`;

The other important thing to understand is that it makes no difference in how the function works. `setPos(3)`; and `setPos(3, 0.5f)`; are 100% the same thing to the computer. Both ways, `y` is 0.5 and the computer doesn't care how it got that way.

Here's another silly example with strings, showing that not all parameters have to have them:

```
string adventure1(string name, string ani="dog", string place="park") {
    string w = name+" and a "+ani+" played in the "+place;
    return w;
}
```

This says if you leave out the last input, `place`, it fills it in with "park". If you leave out the last two inputs, it also fills in `animal` with "dog". But you have to always give one input, `name`, since it has no default. An example of every way to call it:

```
// calls:
string a = adventure1("Al", "ox", "bog"); // Al and a ox played in the bog
a = adventure1("Bev", "fish"); // Bev and a fish played in the park
a = adventure1("Carl"); // Carl and a dog played in the park
a = adventure(); // ERROR
```

There are two ways we usually use them. One is to add a very, very optional extra input. For example:

```
void setPos(float x, float y, float z=0) {
    transform.position = new Vector3(x, y, z);
}
```

In our minds, this is a 2-input function. `X` and `Y` are all we change most of the time. But in the oddball case when we need to, we can throw in a value for `Z`, like `setPos(-7,4,1)`.

The other common use is to make a shortcut for a common value. Suppose my cube game often has `y` near the top, at `y=4`. We might write `void setPos(float x, float y=4)`. In our minds, this is still a 2-input function, but sometimes we'll write `setPos(-7)`; as a shortcut for “-7 `x`, and the usual height.” Even if we never do that, we still have a nice built-in reminder that the normal `y` is 4.

Built-in functions use lots of default parameters. Here's a Unity spin command that really has 3 inputs. It technically has a 4th, but no one ever uses it. The heading is:

```
void Rotate(float xAng, float yAng, float zAng, Space relativeTo = Space.Self)
```

The = after the fourth input means you can skip it. `Rotate(0,90,45)` is legal. It spins you 90 on y and 45 on z (rotations are a little tricky, but hopefully this makes some sense.)

But the fourth input is still fun to look at. The type is `Space` – huh? Oh, it’s an enumerated type. If we look it up there are only 2 options: `enum Space {Self, World};`, and it’s set to the first one.

Your other option is to put `Space.World` as the fourth input. But the system is telling you that the first “Self” way is the one you almost always want.

Even better, people who know rotations know what world-space and self-space mean. For them, that enum is much more helpful than 0 or 1.

Finally, just so you know, I simplified it – the real command is `transform.Rotate(0,90,45);`.

Errors

Inputs are still filled in left to right exactly in order – it won’t skip any to try to make things work. Suppose we have this:

```
void A(string w="cow", int x=0) { ... }
```

You don’t have to give this any inputs, but if you do, the first one has to be a string, for w. `A(3);` gives you error *No overload for method ‘A’ takes ‘1’ arguments* (which isn’t the real problem, but gives you the idea – it can’t match your call to the function.)

Because of the left-to-right, no-skipping rule, default parameters have to start from the right end. This next function is illegal. It tells you *Optional parameter cannot precede required parameters*:

```
void A(string w="cow", int y) { ... } // ERROR
```

16.2.1 overloading

You’re allowed to reuse a function name if the parameters are different. This is called **overloading**. That’s the entire rule. The rest of this is just examples and how we usually use that trick.

A different number of inputs lets you reuse the name. For example, we can make a 2 and a 3 input `max`:

```
float max(float a, float b) { if(a>b) return a; else return b; }

float max(float a, float b, float c) {
    if(a>b && a>c) return a;
```

```

    if(b>c) return b;
    return c;
}

```

They're still two completely different functions. But if we type `max(` the pop-up shows us both, making it look like there are two versions of `max`.

Different types also let us overload a name. For example we can write another version of `max` with 2 inputs, but both are `int`'s:

```
int max(int a, int b) { if(a>b) return a; else return b; }
```

This lets us write `int n=max(5, 2);` without a cannot-convert error. It runs the `int` version. But it's still nothing special – we have three completely different functions which happen to share a name.

Here's a silly overloading example that just shows the rules:

```

void silly(float x) { print( "A" ); }
void silly(string w) { print( "B" ); }

silly(1.2f); // A
silly(""); // B
silly(4); // A (4 can turn into a float, but not a string)

```

An example built into Unity: we've used the float version of `Random.Range`. There's also an `int` version, which only gives whole numbers, like rolling a die. It happens to be an overload:

```

// int version, since inputs are ints:
int n = Random.Range(1,4); // random 1, 2 or 3
// it rolls 1 less than the maximum, just because

// old float version. Could roll 1.653 or 3.983
float f = Random.Range(1.0f, 4.0f);

```

The same as before, these are 2 completely different functions. It just feels like 2 versions of `Random.Range`. They could have named the second one `RangeInt`, but most people think it's nicer to type `Random.Range(` and look through the options.

The full rules for overloading are complicated. For example `Random.Range(2, 5.0f);` has one `int` and one float. It runs the float version since it's closer. And if you write `float f=Random.Range(1,4);` it runs the `int` version. There are more, but if you need them you've gone a little too nuts on overloading.

16.3 Style

Overloading and default parameters are often combined to make it look like we have one very flexible function. When you type a function name and see the little “1 out of 4” indicator, you can arrow down to see the overloads or versions with and without default parameters.

Here’s an example of setting color that counts as 4 functions. The first one is standard. The last two are really just front-ends for the first, which is common:

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

void setCol(float greyScale) { setCol(greyScale, greyScale, greyScale); }

void setCol(string colName, float bright=1.0f) {
    float r=0, g=0, b=0;
    if(colName=="red") r=1;
    else if(colName=="yellow") { r=1; g=1; }
    else if(colName=="purple") { r=1; b=1; }
    ...
    r*=bright; g*=bright; b*=bright;
    setCol(r,g,b);
}
```

The pop-up for `setCol` will show us four these options. The last two are the third version we wrote, with and without the default `bright`:

```
setCol(float r, float g, float b)
setCol(float greyScale)
setCol(string colName);
setCol(string colName, float bright);
```

Notice how the last two `setCol`’s we wrote each call `setCol` themselves. That looks really weird, but it’s more overloading weirdness. They’re just calling the first version.

Just to say it again, default parms and overloading don’t do anything. It’s just a cute way to group a few functions under the same name, to try to make it easier to find them.

Before we had overloading, we gave functions slightly different names like `max2f`, `max3f` and `max2i`, and it was fine.