

Chapter 15

Constants and enums

This section is another little vacation about cutesy language features. It's fine to skim or skip. The main reason to know them is when you see them in other people's code.

You can use these tricks, but they're sort of like +=. They don't do anything you couldn't do before.

15.1 Constants

A trick we'd like to borrow from math is using letters for constants. For example, we might want to make `cpi` a shortcut for 2.54 (centimeters per inch.)

We could just write it like this:

```
float cpi = 2.54f; // DON'T CHANGE! centimeters per inch
```

But the program might be easier to read, and a little safer, if we had a rule to lock `cpi` to that value, and make it an official constant. We do. The rule isn't a huge improvement, but it's short and easy to remember.

When you declare a variable, putting `const` in front locks it to that value. You have to use the declare-and-= trick when you do this. Examples:

```
const float cpi=2.54f;
const int SecsPerMin=60;
const float WaterFreeze = 32.0f;
```

You don't have to use them only for "real" things. You're allowed to use the rule for anything:

```
const string HomeTown="Townville";
const float XMin=-7, XMax=7; // both are locked
const int abc=12;
```

I can't think of any possible reason anyone would want to have `abc` as a locked-in shortcut for 12, but it's legal.

You use them like regular variables, except you can't change them (trying to is an error):

```
if(temp>WaterFreeze) ...
float x = Random.Range(XMin, XMax);
print("You wake up in "+Hometown+".");

secsPerMin=45; // ERROR
// The left-hand side of an assignment must be a variable, a property or an indexer
```

The advantages are minor. They look a little different in the pop-up (mine have an orange F next to them) and we can't accidentally change them. The main advantage is making the program a little easier to read.

You can declare `const`'s locally. This code checks whether we're off the sides, with an extra 0.02 unit tolerance:

```
void fixRange() {
    const float EDGE_GAP=0.02f; // how far from edge counts as out-of-bounds
    if(x<XMin+EDGE_GAP) {} // do off left edge stuff
    if(x>XMax-EDGE_GAP) {} // do off right edge stuff
}
```

Using all caps and underscores is an older style for writing constants.

Constants have one other small, tiny advantage – they run just a tad faster. The compiler replaces them with the numbers. When your program runs, there's no variable look-up needed.

15.2 enumerated types

Sometimes we use `int`'s to stand for an option, in a clunky sort of way. For example, we might make a variable holding different types of cars, set up like this:

```
// Car Types: 0=sedan, 1=compact, 2=convertible and 3=van
int car1type; // use the car table
car1type=3; // car1 is a van
...
if(car1type==1) // is car1 a compact car?
```

Using numbers to stand for things is a solid computer idea, but that last line is icky.

An **enumerated type** is very simple way to make this easier to read. Here's a rewrite. The first line describes a car-type to the computer:

```
enum CarT {sedan, compact, convertible, van};
```

That says we can declare `CarT` variables, which are really ints going from 0 to 3. The computer automatically numbers the words. `sedan` is 0, `compact` is 1, `convertible` is 2 and `van` is 3.

`enum` is a keyword. It's only used in this one place.

Here's a rewrite using our new `CarT`:

```
CarT car1type; // computer knows this is a 0-3 int
car1type = CarT.van; // really just 3
...
if(car1type==CarT.compact) ... // still comparing to 1
```

This is *exactly* the same as the previous version. It declares `car1type` as an int, assigns 3 to it, and compares it to 1. But it looks so much nicer.

If you type out the second line, `CarT-dot` even gives you a pop-up with `sedan` through `van`. It's reading them from the first `enum` line we wrote.

To show it in use, this function tells you how many doors your car has. This is the "old" version, using ints and the hand-made table:

```
int numDoors(int carType) {
    // carType should be a 0-3 number based on the carType table
    if(carType == 1 || carType==2) return 2; // compact/convert
    else if(carType==3) return 3; // van: 2 side, plus back
    else if(carType==0) return 4; // sedans are basic 4-doors
    return -1; // can't happen, but just in case
}
```

This version is exactly the same, using the exact same numbers, but using the `CarT` enum:

```
int numDoors(CarT carType) {
    if(carType == CarT.compact || carType==CarT.convertible) return 2;
    else if(carType==CarT.van) return 3; // 2 side, plus back
    else if(carType==CarT.sedan) return 4;
    return -1; // can't happen, but just in case
}
```

Notice how it takes a `CarT` input. That's a much nicer hint about how it works, even if it's still just an int.

Another, small, enumerated type example that doesn't really do anything:

```
enum bird {crow, swan, duck};
```

```
bird b1 = bird.duck; // this is really a 2
bird b2 = bird.crow; // really just 0
```

```
void Start() {
    // 50% chance b2 is really a swan:
    if(Random.Range(0, 1.0f)<0.5f) b2=bird.swan;
```

Here's a real enumerated example from Unity. You can spin a phone four different ways. 1-4, plus 0 for unknown. Unity uses an `enum` to label them:

```
enum DeviceOrientation {Unknown, Portrait, PortraitUpsideDown,
    LandscapeLeft, LandscapeRight};
```

That tell us `DeviceOrientation` is another word for `int`, and it has 5 possible values, with those names.

Unity creates one global of that type, `Input.deviceOrientation`, and auto-sets it as you tilt the tablet. Notice it's in the `Input` namespace.

Sample code using it:

```
// copy into var with short name:
DeviceOrientation d = Input.deviceOrientation; // really just 0-4
// portrait mode is the long way up/down:
if(d==DeviceOrientation.Portrait || d==DeviceOrientation.PortraitUpsideDown)
    print("please hold it sideways.");
```

This is really checking whether `d` is 1 or 2. But it's easier to understand with those words.

One more example, from standard C#. You can open a file for reading only, writing only, or both. A 0-2 `int` says what you want. In the old days, you wrote `open("cow.txt", 0);`. The second input was how to open it, and everyone knows 0=read only.

C# added a 0-2 `enum`:

```
enum FileAccess {Read, Write, ReadWrite};
```

Now we can use the nicer-looking `open("cow.txt", FileAccess.Read);`. It's extra nice because if we didn't know how the command worked, we'd see the second input is a `FileAccess` variable. Typing `FileAccess-dot` shows the options.

15.2.1 value returning ?: ifs

What makes `ifs` useful is you can put any commands, and all you want, into each part, and you don't even have to have an `else`. But there's one common way to use an `if` that could use a short-cut:

```
string desc;
if(size>100) desc="big"; else desc="small";
```

In our minds, the `if` is really picking between "big" and "small". We're thinking of it more like this:

```
// totally illegal, but a cool idea:
string desc = if(size>100) "big"; else "small";
```

We decided to make a special type of `if` that works like that. The true and false parts are naked values, of the same type. It returns one of them. Legal code for what we were trying to do:

```
string desc = size>100?"big":"small";
```

Everything after the `=` is the special `if`. The rules are:

- It has three parts with `?` and `:` separating them. The second symbol is a colon (not a semicolon.)
- The first thing is any true/false test, the same as an `if`. The next two are possible answers: *test?answerIfTrue:answerIfFalse*
- The two answers must be the same type, but can be any type. In other words, two `ints`, or two `strings`, but not one `int` and one `string`. Ex:'s:

```
int n = (x<0)?-1:+1; // n will be +1 or -1
float g = (w=="cow"?1.5f:2.9f); // cows are 1.5, everything else is 2.9

print( n<10?"yak":7 ); // error -- "yak" and 7 aren't same type
```

You can use the int-instead-of-float shortcut, like in `float f = n<10?4:3.5f`.

- You have to give both answers. If you don't want to give one of them, you can usually put `0` or `""` to make it work. Ex:

```
string w = (x>=100)?"mega": ; // ERROR what goes into w if x<100?
string w = (x>=100)?"mega":""; // legal
```

- You don't need parens around the test, but sometimes it looks nice.
- It counts as math. You can put it inside a longer equation (this sounds tricky, but is a really nice feature). Ex:

```
int n = 3 + (a>10)?5:1; // n is 3 plus either 5 or 1
print("I have " + (fr==0?"no":"some") + " frogs); // I have no/some frogs
```

Putting it inside string math is semi-common. Two more examples of that:

```
print(num + " " + (num==1?"child":"children") ); // 1 child / 2 children

print( "I see " + (num==1?"a":"some") + animal + (num==1?"":"s") );
// I see a frog / I see some frogs
```

`num==1?"":"s"` is sneaky. We only want to print an extra "s" for plural, but we have to give some answer otherwise, so we use "".

Some general examples:

```
score += round==1?10:25; // 10 for round 1, else 25

// leap year has one extra day:
int daysInYear = year%4==0?366:365;
```

Some people use this trick for “n can’t be less than 0.” This looks funny but it works:

```
num = num<0?0:num; // can’t be less than 0

num = num>10?10:num; // can’t be more than 10
```

It can also be nested, but when they start getting that long, it might be better to rewrite them as real ifs. This figures out “ice”, “water” or “steam” based on temperature, using the same logic as a cascading if:

```
string H2O = degs<=32?"ice":(degs<212?"water":"steam");

//same thing, written another way:
H2O = degs<212?(degs<=32?"ice":"water":"steam");
```

15.3 More enumerated types

The computer will sometimes allow you to mix enums and ints. They’re the same thing, but not every combination is allowed.

`car1type=3;` is an error. Even though 3 is Van, it wants you to write out `CarT.van`.

You almost never want `<` and `>` to compare enums, but you can. It compares the int values:

```
if( car1type <= CarT.compact ) // legal. compares as ints
```

Likewise you almost never want to add, but you can. `car1Type++;` is legal. It goes to the next car in the list. Past Van it will go to 4, which isn’t a car.

That's odd, and makes no sense, but legal.

The numbering normally starts at zero. We usually don't care about the exact numbers, so 0 is fine. But, if you want, you can pick the numbers using = after an item. Items without = go up by one. Examples:

```
enum Size {tiny, small, medium=10, hefty, big=20, huge, colossal};  
//          0      1      10      11      20      21      22
```

This is a pretty obscure rule, and there's rarely a reason for it. But I like how it shows how `enums` really are just `int`'s.