# Chapter 9

# Scope and Namespaces

This chapter is sort of a break. Just some rules about different ways and places to declare variables. We'll use them a little, and every programmer needs to know them, eventually. But they won't hurt your head as much as those `if` examples.

## 9.1 Scope

So far, we've been declaring variables in two places: inside of Start and Update, or outside of them (the Inspector variable trick.) Both ways are legal, but they have different rules for how long the variables live and who can see them.

Variables declared outside Start or Update are called **global**. Global variables live for the entire program, and everyone can see and use them.

Variable declared inside Start or inside Update are **local**. Local variables can only be used in the part where they were declared. When that part finishes, local variables are destroyed. A local variable in Update is created and destroyed each time Update runs.

The technical term for where a variable exists is **scope**. The scope of global variables is the entire program. The scope of local variables is inside the {}'s where they were declared. Being destroyed by leaving the {}'s is called *going out of scope*.

Here's a small example of local variables not existing outside where they were made. `n` is declared in Start, so doesn't exist in Update:

```
void Start() {
  string n = "Nif";
  print("n="+n); // n=Nif
}
```

```
void Update() {
  print("update n="+n); // ERROR -- no such variable n
}
```

Local variables also don't "use up" the name outside their scope. This is a good thing. It means we can declare one `n` in Start, and a different `n` in Update:

```
void Start() {
  string n = "Nif"; print("n="+n);
}
void Update() {
  int n=5; // legal. We can reuse the name
  print(n*2); // 10;
}
```

This is why local variables were invented. Imagine you have a very large program, with Start, Update and piles of other things like it. When you declare local variables in one section, you never have to worry about whether someone also declared them in some other section.

You could declare everything as global. But programs are usually easier to read if you use global only for things you need to keep around between Updates.

Here's an example using move-and-wrap-around. It snaps `y` lower as `x` crosses thresholds. `x` has to be global, since it's remembering where I am. But `y` is better as a local, since I recompute it each time:

```
public float x=-7; // <-global

void Update() {
  x+=0.1f;
  if(x>7) x=-7;

  // compute y based on x. 1.5, 0 or -1.5 as we move:
  float y; // <-local
  if(x<-2) y=1.5f;
  else if(x<2) y=0;
  else y=-1.5f;

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

`y` is really just a temporary, used to help place the Cube at the correct height. Declaring it as a local, right before we use it, makes it easier to see that.

### 9.1.1 Misc

The rule in C# is that global variables are always initialized (to 0, or `""`) and local variables are never initialized. But every language has its own rules for

what is or isn't. I think it's better for your code to always initialize everything.

Way back in the "Inspector variables for Input" chapter I mentioned we were piggy-backing off a real rule. This is it. The real Unity Inspector rule is it shows global variables with `public` in front. Just so you know `public` is also a real thing, which we'll see much, much later.

You're allowed to declare globals anywhere in the big curly-braces. It doesn't have to be at the top. They all get declared first, in one magic step (the compiler scans for them as one of its steps.)

Just to show it can be done, this silly program declares `n1`, `n2` and `n3` in funny but legal places:

```
int n1;

void Start() {
  n1=1; n2=7; n3=45; // legal to use n2 and n3
}

int n2; // <- also a legal place to declare a global

void Update() {
  n3++; // this counts as declared
}

int n3; // ditto
```

Most people declare them at the top. But I sometimes declare a global just before the part that uses it – like right before Update.

The rule about needing to declare a variable before you use it is real, but it's for local variables.

Global variables declarations aren't real runnable program lines. The computer declares them all at once, and, to be nice, does simple math like `int n=4*7;`. But you can't do anything complicated. In Unity, you have to use Start for that.

An example:

```
int x=5;
int y=x+1; // ERROR, can't run lines in gobals

void Start() {
  y=x+1; // do it here instead
}
```

### 9.1.2 block scope

If you declare a variable inside of an if/else {}, it's local to that `if`. In other words, it goes away after the `if` finishes. This is called **block** scope.

A common use for this is swapping two variables:

```
if(x>y) {
  int tmp=x; // block scope -- tmp is very, very local
  x=y;
  y=tmp;
}
// tmp no longer exists
```

It looks nice to declare `tmp` exactly when we need it, and it won't interfere with the rest of our program, since it's deleted right away.

### 9.1.3 Common errors

Sometimes when you want to change a global, you accidentally redeclare it, like this:

```
int cats;

void Start() {
  int cats=10; // Oppss! But not an error. Hides the global cats
  // cats=10; // We meant to do this
  cats += 5; // even this is messed up -- changes the local cats
}
```

This isn't a red-dot error – it's a much worse non-error error. We now have a local `cats` which hides the global one we meant to use. This puts local `cats` to 15, then throws it away. Global `cats` stayed 0 the whole time.

You might get a warning *local cats shadows global cats.*

Whenever you use a variable, just think "Am I changing an existing global, or making a new local variable?"

A similar error can happen with block scope. This accidentally makes `food` only exist inside the `if`:

```
if(ani=="cat") string food="mouse";
else string food="canned";

print(ani+" eats "+food); // error -- no variable named food
```

We have to declare it outside the `if`:

```
string food;
if(ani=="cat") food="mouse";
else food="canned";
```

## 9.2 Namespaces

C# (and Unity, and every system using C#) has lots and lots of built-in globals. A standard computer trick is to group them into things like folders, called **namespaces**.

For example, Unity has global variables for the screen's height and width, which way a mobile device is being held, and so on. These are all in a **namespace** which they named `Screen`.

The rule for looking inside a namespace is to use a dot. If you type `Screen`-dot (screen and then a period) the pop-up will show the options. `Screen.width` is the width, in pixels, of the window. It's really the `width` variable in the `Screen` namespace.

For fun, you can test by having Update copy `Screen.width` into an Inspector global. While running this, you can resize your window and watch `ht` change:

```
public float ht;
void Update() { ht=Screen.width; }
```

You might notice that the pop-up for `Screen` says `public sealed class`, and not namespace. **namespace** is just the general term. Some languages use the actual word `namespace` in the language. Others use the term *package*.

C# uses an overly complicated mix – they have limited official `namespaces`, and let you use a `class` as a namespace. The important thing is, whenever someone types `Screen`-dot, they're thinking of Screen as a namespace.

Here are more namespaces and stuff in them, just for fun examples. You won't need to know them for later:

- `Mathf` holds math stuff. `Mathf.PI` is 3.1415. `Mathf.Floor` is a built-in that rounds 3.6 down to 3 (we haven't seen the rules for this yet.)
- `Time` holds info about game-time stuff. `Time.deltaTime` is the number of seconds between the last Update and this one (it's usually about 0.02.) `Time.time` is how many seconds since you pressed Play.
- `Debug` holds testing commands. `print` is really a shortcut for the `Log` command in the `Debug` namespace. `Debug.Log("abc");` prints `abc`.
- `System.Math.PI` also holds PI. It's a namespaces in a namespace (think of it like a folder in a folder.) `System` is the standard C# namespace. If you're wondering, `System.Math.PI` is a `double`. `Mathf.PI` is a `float`.

`Time.time` is one of my favorite examples of how namespaces work, since it looks funny having `time` twice in a row, but makes perfect sense once you understand it:

The `Time` namespace hold a lot of things about time and timing things, so `Time` is a good name for it. The most commonly used thing in it is how many seconds the game has been running. That variable should have an easy, short name, so we picked `time`.

Together, it's the `time` variable in the `Time` namespace – `Time.time`.

### 9.2.1 Notes/rules

You can use the same variable in different namespaces. That's really a version of the scope rules. That's why `System.Math.PI` and `Mathf.PI` are both legal. You could have `Screen.height` and `Raccoon.height` and also name one of your variables `height`.

In the editor, typing the dot triggers the pop-up. Suppose you type `Screen.wi`, click away, come back, and the pop-up is gone. To get it back, delete down to just `Screen` and retype the dot.

Later on, C# re-uses the dot in a different way. When we come to it, I'll write this again and explain it. I want to sort of pre-warn you about it, since it's one of those things that can be really confusing.

The `using`s at the top of the program are about namespace. You don't need to know this – only keep reading if those two lines at the top of every program are bugging you. `UnityEngine` is the master namespace for all the Unity built-ins. `using UnityEngine;` means you can skip typing it.

`Screen` is really inside of `UnityEngine`. The real width of the screen is `UnityEngine.Screen.width`. The starting `using UnityEngine;` allows us to leave it out (it really says: if you can't find something they typed, try looking for it in the `UnityEngine` namespace.) But, again, not important to know.