Chapter 33

Nested classes, static

This section is just two funny rules about classes jammed together. The enum/class section isn't really new. It's more funny stuff that using `private` can do. And remember, you never need to use private. But `static` is a new rule, used to make real globals.

33.1 Nested `enum`'s, classes and `public`

In a class, `public` and `private` also apply to `enum`'s or any other classes you write. So far it didn't matter, since we only used them from inside of that class. But outsiders can only use the ones you make `public`.

An example, this makes one public and private of each:

```
public class Player : MonoBehaviour {

  public enum statusT { resting, walking, running };
  private enum hatT { none, cowboy, bowler, baseball, bonnet };

  public class Dog { public string name; public int age; }
  class Lion { public string noise; public float tailLength; }
  // Note: Lion is private, since we didn't write anything
```

Inside of `Player`, we can use all 4 of these (obviously, since we've been doing it for a while). Outside, we can't see `hatT` or `Lion`, since they're `private`.

We'd make them `private` for roughly the usual reason: they're "helper" classes and enums. No one outside of the class should be using them, or should even see them in the pop-up.

To use the `public` ones outside of player, we're required to add player-dot in front. Here's an example of a new class `Kitten` using them:

```
public class Kitten : MonoBehaviour {
  Player.Dog d1; // the Dog inside of Player
  Player.statusT s1; // same
  //Player.Lion L1; // ERROR -- Lion is private in Player

  void Start() {
    d1 = new Player.Dog(); // <- also here
    s1 = Player.statusT.resting; // <- yow, and here

    d1.age=5; // whew! this is normal
  }
```

The reason should make sense. We could have one `Dog` in a file by itself, and another inside of `Player`, and another inside of `testA`. Three different Dog classes. Their names would be `Dog`, and `Player.Dog`, and `testA.Dog`. It's the usual namespace situation.

And we like it that way. If `testA` needs its own personal `Dog` class, it should be able to have one.

Obviously, you can skip the first part from inside the class. Inside of `Player` we can use `Dog` as a shortcut for the real name, `Player.Dog`.

You can't have a `public` variable of a private type. It's easy to do by accident:

```
public class Player : MonoBehaviour {
  // Lion class is private:
  class Lion { public string noise; public float tailLength; }
  public Lion L1; // ERROR
```

You get a really cool error: *Inconsistent accessibility: field type 'Player.Lion' is less accessible than field 'Player.L1'.*

The problem is obvious, but takes some thought. Anyone outside the class trying to use `L1` will need to look up what `Lion`'s are, which is private, so they won't find it. They can't possible use `L1` if they aren't allowed to read its type.

When you get this error, it almost always means that you wanted to make the class public, and simply forgot to.

You might remember the special way to put a class in the Inspector from way back:

```
// dog that can be in the Inspector:
[System.Serializable]
public Dog { public string name; public int age; }

public Dog d1; // has an Inspector slot
```

Now we know why we needed the extra `public` in front of the class. We can't make `d1` public without having the class itself be public.

And finally, as usual, if you ever have any problems caused by private classes or private enums, make everything public. You never actually need to make something private.

33.2 Static / namespaces

`static` is a special word that allows us to make true global variables and functions. Write something inside of a class and add `static` in front. Now it's a true global, using that class as a namespace.

33.2.1 static functions

The trick is most common with functions, First here's an example making a global function, inside of `Dog`:

```
class Dog() {

  // normal function:
  public static bool isEqual(Dog d1, Dog d2) {
    return d1.age==d2.age && d1.name==d2.name;
  }
}
```

Because of the `static`, `isEqual` is a normal function. You can't call it with a Dog in front, and wouldn't want to. But anyone, anywhere could call it like `if( Dog.isEqual(pet1, pet2) )`.
The `Dog` in front is just a namespace, telling us how to find `isEqual`.

This is a pretty typical example. We wanted to write a regular function to compare two Dogs. Then we thought it would be nice to group it with the dogs, so we put it inside with `static`. An advantage is we can now write `Dog`-dot and find it.

I think the biggest confusion about `static` is that it goes in front like public/private, but it's completely different. Public/private don't change how something works, just who can use it. But `static` is a change in how it works.

Our old function `Vector3.Distance(v1, v2)` uses this trick. It's a normal function telling you how far apart the two input points are. It's put inside the `Vector3` namespace to make it easier to find:

```
public struct Vector {
  public float x, y, z;
  // static in front means it's not a member function:
  public static float Distance(Vector A, Vector B) {
    // standard 3D pythagorean theorem:
    float dx=B.x-A.x, dy=B.y-A.y; dz=B.z-A.z;
    return Mathf.Sqrt(dx*dx+dy*dy+dz*dz);
  }
}
```

The `static` is telling us we don't use `v1.Distance(v2, v3)`. It's not a member function. Notice how the inside is all `A.x` and `B.y`. There's no direct use of "our" member variables, since we're a normal function.

Here's an example writing `desc` both ways:

```
public class Dog {
  // member function:
  public string desc1() { return name+" "+age; }

  // non-member function:
  public static string desc2(Dog d) { return d.name+" "+d.age; }
}
```

Notice the first has no dog input, since a dog calls it: `d1.desc1();`. The second has an input since it's a normal function: `Dog.desc2(d1);`.

33.2.2 Fake classes for namespaces

We've seen the idea of a namespace as a folder for global functions. C# fakes this using the static rule and a do-nothing class. Here, `Rnd` is a folder for some dice-rolling functions:

```
using UnityEngine;
using System.Collections;

public class Rnd { // this is not a class
  public static bool pct(int chance) { return Random.Range(1,101)<=chance; }

  public static int roll_xDy(int numDice, int diceSides) {
    int sum=0;
    for(int i=0;i<numDice;i++) sum+=Random.Range(1,diceSides+1);
    return sum;
  }
}
```

We now have global functions `pct` and `roll_xDy`, who's full names are `Rnd.pct` and `Rnd.roll_xDy`. Technically `Rnd` is a class, but not really. We're abusing the rules to use it as a folder.

It's an idiom. For people who haven't seen the trick, a class with no member variables is horribly confusing. But long-time users don't even see the word class anymore. This is the C# way of making a namespace.

Unity has several of these. `Mathf` holds functions like `Mathf.Sqrt(8)` (square root) and trig functions and so on. Mathf is clearly a folder – a namespace. But the actual way it's written is a struct with no fields and all static functions.

33.2.3 static variables

`static` in front of a variable stops it from being a field, and turns it into a regular global. This makes two globals:

```
class gameStats { // this is not a class
  public static int currentLevel;
  public static int livesLeft;
}
```

Anyone, anywhere can use `gameStats.livesLeft--;` or `if(gameStats.currentLevel==5)`. As before, `gameStats` isn't really a class.

To compare, if they didn't have `static`, we'd start with no variables. And declaring `gs1`, `gs2`, and `gs3`, would create 3 sets. Adding `static` means we start with them, and can never make any more. They're regular declared variables.

Unity has global variables using this trick. The global for the time in seconds since it started, is `Time.time`. It turns out `time` is a static float, and `Time` is a class abused to be a namespace.

There's rarely a reason to mix `static`'s into a real class, but you can do it. Suppose I want a variable holding the total number of Dog's ever made. I may as well keep it in the Dog class, as a static:

```
class Dog {
  public string name; // these are the fields
  public int age; // in each Dog

  public static int count; // global variable
}
```

This gives us one global, named `Dog.count`. Each Dog we declare has name and age, as usual.

It's common enough to put global functions and variables in the same fake class. For example:

```
class gameStats { // this is not a class
  public static int currentLevel;
  public static int livesLeft;

  public void reset() { currentLevel=1; livesLeft=3; }
}
```

`gameStats.reset()` is a global function, changing global variables.

33.2.4 File of classes

I've been defining classes, and everything else, inside of scripts since that was the only place we had.

If multiple scripts in a program wanted to use a class, it's fine to put it in a file by itself. We'd create this file the normal way, and anyone could declare things from it:

```
using UnityEngine;
using System.Collections;

public class Dog { public string name; public int age; }

public class FullName {
  public string first, last;
  public string desc() { ... }
  ...
  }

  public enum catBreed {persian, siamese, tabby, AmerShorthair};
}
```

Notice how pretty this is. It's our normal definitions, without any of the extra stuff from a script.

Also notice how these are "naked". They're not wrapped in a namespace or anything. `Dog`'s entire name, for anyone, is just `Dog`. Any script could declare `Dog d;` or use `catBreed cb1=catBreed.tabby;`.

We could throw a namespace around them. `class Pet` { `...` } around the whole thing. Then we'd have `Pet.Dog d1` and so on. The advantage is the usual scope help: with these hidden inside of `Pet`, another part of the program can re-use the names.