Chapter 32

Nested classes, static

This section is just two funny rules about classes jammed together. The enum/class section is just explaining and isn't really new stuff. But `static` is a new rule, used to make real globals.

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

Inside of a class, for example a script, `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`.

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

  void Start() {
```

```
    d1 = new player.Dog(); // <- also here
    s1 = player.statusT.resting; // <- yow, and here

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

This is the rule from way back about being able to re-use a class as a namespace. We specially defined `Dog` inside of `player`. So it's the `Dog` class inside of player. The full name is `player.Dog`.

As usual, we can define different `Dog`'s in other scopes. `testA` could have it's own different `Dog`. Then we'd have a choice between `player.Dog` and `testA.Dog`.

You can skip your own name inside the class, since you're already there. Inside of player, you could write `player.Dog`, but just `Dog` as a shortcut is fine (obviously, since we've been doing it that way.)

Why would you make one of your classes or `enum`'s private? Maybe no one outside of you needs to see it, so why confuse them.

You might remember that to put a class in the Inspector it needs `public` in front (and the Serializable thing):

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

This new `public` rule explains the first part. If `Dog` isn't `public`, no one else can see it. And if no one can see it, it can't be in the Inspector.

If you try to make a public variable for a non-public thing, you get a cool error:

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

Other people can use `L1`, but they can't really, since they aren't allowed to look at `Lion`s. The error message is *Inconsistent accessibility: field type 'Player.Lion' is less accessible than field 'Player.L1'*. Which is a fancy way to say if they can't see `Lion`'s, it makes no sense to let them use a `Lion` variable.

32.2 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 global, using that class as a namespace.

32.2.1 static functions

The trick is most common with functions – we've been using them. First here's an example making a global function, inside of `Dog`:

```
class Dog() {

  // normal function:
  public static string randName() {
    string[] W={"Rolf", "Spot", "Lu-lu", "Blackie"};
    string w=W[Random.Range(0,A.Length)];
    if(Random.Range(0.0f, 1.0f)<0.2f) w="The most noble "+w;
    return w;
  }
}
```

Because of the `static`, `randName` is a normal function. You can't call it with a dog in front. You wouldn't want to since the inside doesn't use a dog. Anyone, anywhere could call it like `string n=Dog.randName();`.

The `Dog` in front is just a namespace, letting us know how to find `randName`.

This is a pretty typical example. We wanted to write a regular function to generate random dog names. Then we thought it would be nice to group it with the dogs, so we put it inside with `static`. Now we can 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 big change to whatever it's in front of.

The 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 a "teaching" 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: `FullName.desc2(d1);`.

32.2.2 Fake classes for namespaces

A common use of `static` is to make a fake class who's only purpose is to be a namespace. In Unity, `Mathf` is like this. It's technically a class, but it's really just a holder for minimum, Square-root and cosine.

The "class" will only have `static` functions, and nothing else. It's just a way to group them. Here's a fake class named `Rnd` which is only used to group some custom-made random functions. You'd write it just like this in a new file:

```
using UnityEngine;
using System.Collections;

public class Rnd {
  public static pct(int chance) { return Random.Range(1,101)<=chance; }

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

Now anyone can use `if(Rnd.pct(15))` to get a 15% chance, or `roll=Rnd.roll_xDy(3,6);` to roll 3-18 (3 6-sided dice.)

This is a fake class since it has no member variables or functions. You could create one: `Rnd r1=new Rnd();`, but nothing would be in it. `r1`-dot would show nothing. The only use of `Rnd` is as a namespace for those global functions.

If you hated the magic get/set, skip this next one. In Unity, `float n=Random.value;` is a shortcut for a random 0.0f and 1.0f. It's written as a static property (yikes!):

```
class Random {
  ...
  public static float value {
    get {
        int n=Range(0.0f, 1.0f); // shortcut for Random.Range
        return n;
    }
  }
  ...
}
```

It's a strange beast. One cool thing: notice how it uses `Range` inside. It doesn't need the `Random` in front since we're already inside of Random.

32.2.3 static variables

`static` in front of a variable makes it be a regular global. Even though it's inside the class, it's not a field. The only funny thing is it now uses the class as a namespace.

A typical use, this makes two real globals:

```
class gameStats {
  public static int currentLevel;
  public static int livesLeft
}
```

Anyone, from any script of any type, can use `gameStats.livesLeft--;` or `if(gameStats.currentLevel==5)`.

There's almost never a reason to mix `static`'s into a real class. Here's an example where I want one global count of all dogs:

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

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

Each dog gets `name` and `age`, but there's one copy of `dogCount`. Even before we have any dogs at all, we can use `Dog.count=0;`. It's a real global variable.

32.2.4 Fake global class

This is pretty silly, but you can make a class with only static variables and functions. The advantage is that everyone can use it from anywhere – no pointer

required – and you don't need to use `new`. The drawback is you can never have a second copy.

Here's one to manage stats in a match:

```
public class Match {
  public static int round; // which round of this Match
  public static float timeStarted;

  public static int[] Score; // Score[0] is player1 ...

  public static init() { round=0; Score=new int[2]; }

  public static void beginNewRound() {
    Score[0]=0; Score[1]=0;  timeStarted=Time.time;
  }

  public static addScore(int pointChange, int playerNum) {
    Score[playerNum]+=pointChange;
  }
}
```

Since these are all normal functions and variables, we could use it like:

```
Match.init();

Match.beginNewRound();
Match.round=4;

Match.addScore(+2, 0);
```

This is almost always a bad idea – you almost always want the ability to create several of these: for replays, multiplayer, or just for testing. But it's a neat example of how static works.

32.2.5 File of classes

If multiple scripts in our program wanted to use `Dog`, `Cat` and so on, we can put them all in one file, which just lives in Project and never goes on anything:

```
using UnityEngine;
using System.Collections;

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

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

```
  ...
  }
}
```

Now any script, anywhere, can use `Dog d1 = new Dog();` and `FullName f1=new FullName();`.

Putting them in a file like this, instead of "nested" inside a script, like we usually do, is just preference. If the player-script mostly uses Dog, putting it inside there seems fine.

32.2.6 Fun with scope

Inside of a member function, we now have two different things we could be taking about – members or statics. Here's `Dog` using a member and static variable:

```
class Dog {
  public int age;
  public static int count;

  public void dummyMemFunc() {
    age=5;
    count++;
  }
}
```

`age=5` is my age, but `count++;` is the global dog count. If we wanted to tell them apart we could use `this.age` and `Dog.count`. But we don't usually do things this confusing and can usually tell from the names.