Chapter 31

Scripts as Classes

Now that we know how classes really work, we can see what's actually going on with Unity scripts. And now that we know about pointers, we can play with pointers to scripts.

These are examples of using Unity, but also the way regular programming works.

31.1 How scripts really work

This is probably obvious by now – scripts are classes. The front part:
`public class testA : MonoBehaviour {`
defines a class named `testA` (the colon-Monobehaviour part is a rule for later.)

Dragging a "script" onto a gameObject tells Unity to `new` a copy. Dragging it twice will `new` two copies of that class. If you don't drag it onto anything, there are no copies, which means there's nothing for Unity to run.

So, finally, this explains from way back in chapter 2 why a simple `print("A");` script had to be dragged into a gameObject: once you have a class definition you need a variable of that type to do anything.

More about what "scripts are really classes" means:

- All of the script functions we wrote are really member functions. Even Start and Update.
- All of the global script variables are really member variables. When we have `void Start() { ticks=0; }` we're really setting a member variable in a member function.
- We didn't write `public` in front of everything, which means most things are private. That doesn't matter with just one script, since private/public is about what people *outside* of you can see.

- If it's not bugging you that Start and Update are private, skip this. Otherwise: things outside the class can't see private functions. If Unity ran `c1.Start();` on one of your scripts, it really would get a *"Start is private. Access denied"* error. It turns out Unity uses an obscure `C#` trick to access Start, Update and all of the other built-ins..
- Two copies of a script have two different sets of variables. It's the same rule as `Dog d1` and `d2` having different names and ages.
- As a check, you can try the "this" trick: in a script, type `this`-dot. Your "globals" (really your member variables) and functions will appear in the pop-up, just like for any other class.

There are two bits of magic that make you think your scripts aren't really normal classes. For free we get `transform` and `GetComponent` (as in `transform.position` and `GetComponent<Renderer>` for color changes.)

It turns out we inherit those from `Monobehaviour`. Inheritance is a rule we haven't seen yet, but it's standard computer programming.

The way a script knows it's on a gameObject is through the `transform` pointer. Whenever you drag a script onto a gameObject, Unity `new`'s a copy and sets the script's `transform` member variable to point there. If you drag another copy of the script only another gameObject, its `transform` points to that other gameObject.

Here's a picture of a gameObject with a script on it:

```
GameObject (a cube, for example)
  List of Components:
    transform
    renderer
    testA <-your script is a component
    Rigidbody (if you added one)
```

The gameObject is a simple class with a `List` of components. Your script is one of them. The gameObject has a pointer to your script (using the List) and you have a pointer back to it. That's 100% why you count as being "on" that thing.

`GetComponent` is an inherited member function. It works since the system knows the set-up – it knows every script is part of a list of other components. `GetComponent<Renderer>` uses the `transform` pointer to find the List we're in, then searches it for the Renderer (or whatever else.)

Like `GameObject`, `Monobehaviour` is a Unity add-on, not a part of `C#`. But the way we inherit things from it, which we can use for free, is standard programming.

31.2 Pointers to classes in Unity

So far we can use pointers to aim at GameObjects. We can grow that into using pointers to point to other *scripts*.

The basic idea is the same as before. If we have pointer a normal class, like `Camel c1;`, then we can run `c1.setName("Bruce");`. If instead we have a pointer to another script, like `scriptC c1;`, we can run `c1.turnRed();`.

A simple example: this script has no Start or Update. But we'd still drag it onto a few Cubes we want to be able to change:

```
class cubeScriptC : MonoBehaviour {
  public void turnRed() { changeCol( new Color(1,0,0) ); }
  public void turnYellow() { changeCol( new Color(1,1,0) ); }

  public void teleport() {
    Vector3 pos;
    pos.z=0
    pos.x=Random.Range(-7.0f, 7.0f);
    pos.y=Random.Range(-4.0f, 4.0f);
    transform.position=pos;
  }

  void changeCol(Color c) { GetComponent<Renderer>().material.color=c; }
}
```

First, notice how they're normal script "change me" functions. `turnRed()` turns our Cube red. The only thing funny is how this script never calls `turnRed`. But that's OK – normal classes have functions for *other* people to call.

That leads to the second thing: most functions are `public`. So cubeScriptC is a script-style class, but it's also written using normal class thinking.

Now here's a master script which uses it. As normal, it has to go on something, but it can be the camera or just any gameObject:

```
class testC : MonoBehaviour {
  public cubeScriptC c1, c2; // <- links to 1st script

  void Start() {
    c1.turnRed();
    c2.turnYellow();
  }

  void Update() {
    if(Random.Range(0,200)==1) c2.teleport();
  }
}
```

That first line declares 2 pointers, `c1` and `c2`, to a class. It's a normal class we wrote, which also happens to be a script on a gameObject. The only special part is how Unity lets us drag in the Cubes we pre-made (it checks they have a `cubeScriptC` on them, and links to it.)

Then below, Start and Update can use those variables like normal: `c1.turnRed();` and `c2.teleport();`. Those are just normal member function calls to a class *and* they're reaching out to change some other gameObject.

If you type it in, notice how `c1`-dot pops up the three member functions, just like a real class, since `cubeScriptC` is a real class.

We can do the same trick with cubes we create using Instantiate. Assume we have a prefab Cube with `cubeScriptC` on it. This would create 5 and set them up:

```
class spawnTestA : MonoBehaviour {
  GameObject prefabC; // drag in a prefab with cubeScriptC

  void Start() {
    for(int i=0;i<5;i++) {
      GameObect gg=Instantiate(prefabC);
      cubeScriptC cc=gg.GetComponent<cubeScriptC>(); // <- new
      cc.teleport();
      cc.turnYellow();
    }
  }
}
```

This uses our same old Instantiate command to create the Cubes, then one new rule. The cubes we create have a `cubeScriptC` on them, since we put it there. Clearly the system has a way for us to find it – it wouldn't be very good if we couldn't. It turns out `gg.GetComponent<cubeScriptC>()` grabs the link. It's not completely obvious, but it should be somewhat familiar.

Once we have `cc` aimed there, we can use `cc.teleport()` and `cc.turnYellow()`, the same as before.

So overall, now that we know scripts are classes, we can write scripts with useful functions and put them on gameObjects. Then some other script can get a link to that variable and run the public member functions.

If you know basic class use, you know 90% of how Unity scripts talk to each other.

31.2.1 Multi-script ball-dropping game

This section will make a little game about dropping balls on targets, using three scripts. Here's the plan:

- There will be three classes: `player`, `flyingBall` and `target`. In Unity terms, we'll write three scripts.
- The player will be a cube that we can steer along the top, dropping balls (the whole thing will be from a front view, so the balls will just fall.) Same as always, the player-Cube will have the only copy of our "main" script, `player`.
- We'll make 4 target Cubes. Just regular Cubes with the `target` class on them. Position them anywhere interesting, with z=0 (so the balls we drop can hit them.) We'll be trying to destroy them.
- The ball should be a prefab with a Rigidbody added (so it falls) and the `flyingBall` script on it. The player creates one, with Instantiate, for each ball drop.
- When the player creates a fresh ball to drop, it will find the `flyingBall` class on it, reach over, and set some member variables. When a ball hits something, it will try to find the `target` script on what it hit, and call a member function to make the target react.

The player is mostly old code. This puts us near the top and makes the arrow keys move us back-and-forth. Update calls `handleShoot` when we press the space key, which I'll write later:

```
class Player : MonoBehaviour {
  Vector3 pos;
  float nextDropTime=0; // game seconds. can't shoot if < this

  // start in upper-center:
  void Start() { pos = new Vector3(0,4.5f,0); transform.position = pos; }

  void Update() {
    if(Time.time>nextDropTime && Input.GetKeyDown(KeyCode.Space)) {
      nextDropTime=Time.time+0.5f; // 1/2 second until next shot
      handleShoot();
    }
    handleMove();
  }

  void handleMove() {
    int mv=0;
    if(Input.GetKey(KeyCode.LeftArrow)) mv=-1;
    if(Input.GetKey(KeyCode.RightArrow)) mv=+1;
    if(mv!=0) {
      pos.x=Mathf.Clamp(pos.x+0.1f*mv , -7, 7);
      transform.position = pos;
    }
```

```
}
```

The only thing interesting so far is how `nextDroptime` uses the `Time.time` trick to force a half-second delay between shooting.

The `handleShoot` function uses `Instantiate` to create a ball, then grabs the script and sets 2 public variables. For simplicity, I'm placing the new ball below the player-Cube and letting it drop.

Here's the player's shooting function, then more explanation:

```
  public GameObject ballPF; // drag in prefab ball with flyingBall script

  void handleShoot() {
    GameObject ss = Instantiate(ballPF);
    Vector3 pos = transform.position; pos.y-=1.0f; // just below us
    ss.transform.position=pos; // place bullet below us

    // get a pointer to our bullet's flyingBall variable/script:
    flyingBall bs = ss.GetComponent<flyingBall>();
    bs.hurtAmt=Random.Range(2,5+1);
    bs.endTime = Time.time+4.0f;
  }
}
```

The flyingBall script needs those two variables to be `public`. Here's part of it. Update only uses the `Time.time` trick to kill it when it's lived for long enough:

```
class flyingBall : MonoBehaviour {
  public float endTime; // set when player makes us
  public int hurtAmt; // ditto

  void Update() {
    if(Time.time>endTime) Destroy(gameObject);
  }
// not done
```

This is where each one being a different thing matters. If we've dropped three balls, we have 3 copies of the class, each with their own `endTime` and `hurtAmt`. Just like Dogs each have their own name and age.

The rest of `flyingBall` checks when it hits something. I used `OnCollisionEnter` once way back, so you may not remember it. It runs when we hit something:

```
  void OnCollisionEnter(Collision cc) {
    // try to get pointer to the target script on what we hit:
    target trg = cc.gameObject.GetComponent<target>();
```

```
    if(trg==null) return; // what did we hit? Not a target

    // use target member function (not written yet):
    trg.getHit(hurtAmt);
    Destroy(gameObject); // make us vanish
  }
}
```

Those first two lines are a common, slick trick. We want to check if we hit a target, and we need a link to the target script if we did. So we try to find the script – if we can't, it wasn't a target.

Obviously GetComponent returns `null` when it can't find something, since that's how all pointer-searching functions work.

We know the `target` script has a public `getHurt` member function. Here's the whole script:

```
class target : MonoBehaviour {
  private int health=10;
  private bool isDying=false;
  private int wiggle=0; // # of times to wiggle (set after a hit)

  public void getHit(int amount) { // <-balls call this
    if(isDying) return; // can't kill it twice
    health-=amount;
    wiggle+=15+amount*5; // update uses this
    if(health<=0) beginDie();
  }
// not done
```

`beginDie` is a typical private function. It's only purpose is to make `getHit` look nicer. It's only two lines now, but I thought it might get bigger later:

```
  private void beginDie() {
    isDying=true; // Update looks at this
    GetComponent<Renderer>().material.color = Color.Red;
  }
```

The rest of `target` is sneaky tricks. I want the target to wiggle on a hit, and if a hit kills it (reduces health to 0 or less) it should shrink to nothing. So Update checks whether `isDying` is true (it's a flag) and whether `wiggle` has been set (more than 0):

```
  void Update() {
    //if(Input.GetKeyDown("a")) wiggle+=10; // testing
    //if(Input.GetKeyDown("x")) getHit(5); // testing
```

```
    // getHit will set these, this is where we use them:
    if(wiggle>0) handleWiggle();
    if(isDying) handleDie();
  }

  private void handleWiggle() {
      // move a little bit, stay in bounds, reduce wiggle counter:
      Vector3 pos = transform.position;
      pos.x += Random.Range(-0.03f, 0.03f);
      pos.x = Mathf.Clamp(pos.x, -7, 7);
      pos.y += Random.Range(-0.03f, 0.03f);
      pos.y = Mathf.Clamp(pos.x, -3, 4);
      transform.position = pos;
      wiggle--;
  }

  private void handleDie() {
    // get a little smaller, die at size 0:
    float sz = transform.localScale.x - 0.02f;
    if(sz<=0) { Destroy(gameObject); return; }
    Vector3 sc = new Vector3(sz, sz, sz);
    transform.localScale = sc;
  }
}
```

I actually wrote `target` first, which is why the top two lines are there. I wanted to test if wiggling and dying worked, so made A and X keys give fake hits. It took me a while to get the wiggle to look right, so this was a real time-saver.

31.2.2 Hand-running fake updates

This next example shows how a more traditional program can fake the Unity set-up. It's still using Unity, and really runs, but we're thinking in a more traditional programming way.

We'll make a standard-seeming class, with member variables and functions, but no Start or Update and anything else which runs by magic. This would go on a prefab ball. The main program will create 8 of them, save links to the class in a List, and manually update them.

The class for our ball prefab:

```
class colorBall : MonoBehaviour {
  float nextChange; // time for next color change
  Color baseCol; // color will flicker around this
```

```
  public bool isReady() { return Time.time>=nextChange; }
  public void setNextTime() { nextChange = Time.time + Random.Range(0.2f, 0.6f); }

  public void setStartCol() {
    baseCol.r = Random.Range(0.0f, 1.0f);
    baseCol.g = Random.Range(0.0f, 1.0f);
    baseCol.b = Random.Range(0.0f, 1.0f);
  }

  public void nextCol() {
    Color cc=baseCol;
    // give slight tweak:
    baseCol.r += Random.Range(-0.2f, 0.2f);
    baseCol.g += Random.Range(-0.2f, 0.2f);
    baseCol.b += Random.Range(-0.2f, 0.2f);
    GetComponent<Renderer>().material.color=cc;
  }
}
```

Even though this will be on the 8 Cubes we make, it's more like a hand-made class like `Dog`. The only magic part is the last line, reaching to our gameObject to set color.

The main program could go on the light, or a dummy object. It makes the 8 balls, saves pointers to the 8 `colorBall` variables in an array, and runs them in Update (by calling the a few member functions):

```
class player : MonoBehaviour {
  public GameObject ballPF; // drag prefab ball with colorBall here

  List<colorBall> C; // will point to everyone's scripts

  void Start() {
    C=new List<colorBall>();
    for(int i=0; i<8; i++) {
      GameObject gg = Instantiate(ballPF);
      Vector3 pos; pos.z=0;
      pos.x=Random.Range(-6.0f, 6.0f);
      pos.y=Random.Range(-3.0f, 3.0f);
      gg.transform.position = pos;
      colorBall cc=gg.GetComponent<colorBall>();
      cc.setStartCol(); // calls that ball's setup function
      C.Add(cc);
    }
  }
```

```
  void Update() {
    for(int i=0;i<C.Count;i++) {
      if(C[i].isReady()) {
        C[i].setNextTime();
        C[i].nextCol();
      }
    }
  }
}
```

The one Update here is the only thing that runs. It fakes running updates on the 8 Cubes using their member functions.

Notice how `colorBall` is using the interface/implementation split. `setNextTime()` and `isReady()` do all of the work we need. They use private variable `nextChange`, but we don't need to know that.

Also notice how this naturally uses an array of classes. `C[i].nextCol()` would be a complicated line a few chapters ago, but hopefully now it makes sense.