Chapter 32

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.

32.1 How scripts really work

This is probably obvious by now – scripts are classes. That's the reason they start with `public class testA`. Unity scripts have some added tricks, but in every way they are for-real classes.

Some of what this 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 we make some script functions `public`, other people can call them.
- 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 private "magic word" functions.

- Like all classes, they need to be `new`'d to exist, and can be `new`'d multiple times. In Unity, dragging them onto a Cube auto-`new`'s them. That's why even printing scripts need to dragged onto something, and why double-dragging is completely fine.
- 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.

Just to be complete, the `: Monobehaviour` in a script definition is a standard programming rule called Inheritance, which we'll see much later. It's the way Unity knows a class is draggable and that it should run Start and Update. In other words, it tells the system that a class is a "script".

That's also how scripts get the free variables `transform`, `gameObject` and the free function `GetComponent`.

When the system `new`'s a class-script, the primary owner is the Cube you dragged it onto. It gets a pointer to your script. Then the script's `transform` and `gameObject` pointers are aimed back at the Cube. That's all of the magic of being "on" a gameObject.

32.2 Pointers to classes in Unity

The main thing about classes being scripts is we can have pointers to them. Suppose we have a script named `testA`. Anyone else can declare: `testA ss;`, which could possibly link to a `testA` script.

Even cooler than that, we know `GetComponent<Renderer>()` finds a pointer to our Renderer, which is a class. The input to GetComponent is any Unity class. Our scripts are Unity classes. That means we can use `g.GetComponent<testA>()` to find the script on `g`.

32.2.1 Direct pointers

Here's a simple example of one main script using pointers to scripts on two other Cubes.

This first class looks like a hybrid between a normal script and a normal class. It goes on gameObjects, but it doesn't have a Start or Update. That's legal, but means it will never do anything.

But it has all `public` member functions. Those are normal – the purpose is for other people to call them. Except the functions use the special script color/position stuff:

```
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;
  }

  // private helper function, just to show we can:
  void changeCol(Color c) { GetComponent<Renderer>().material.color=c; }
}
```

Suppose this was on a Cube and we had `cubeScriptC cc;` that was aimed at it. Then `cc.turnRed();` would reach out to the script and have it turn itself red.

Let's say we drag that script onto 2 Cubes. Just to be sure: Play would do nothing (but no errors, either.)

This demo script, on anything else, could be used to change both of them:

```
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();
  }
}
```

If we drag the two Cubes into those slots, it links `c1` and `c2` to them. Before, that trick linked GameObject variables to Cubes. Now, it keys off the type and links to the script.

For fun, drag in a Cube without that script – it won't let you.

Start runs and calls `c1.turnRed();`. That's a normal member function call. And `turnRed()` runs like a normal function in a script, turning its Cube red.

For more fun, type just `c1`-dot. It pops up those three member functions, just like a real class.

32.2.2 GetComponent to find scripts

We can do the same trick with cubes we create using Instantiate. When we create a Cube, we saved the pointer to it in a GameObject variable. If it has a script, we can get with using GetComponent.

Assume we have a Cube with that same `cubeScriptC` set up as a prefab (which means to drag it down into Project.)

This script uses Instantiate to create 5 of them, grabs the script on each, and uses it to set up each one:

```
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();
    }
  }
}
```

As usual, once we have `cc` aimed where we want it, it doesn't matter how it got that way. We get to use `cc.teleoport()` and `cc.turnYellow()` to remote control that Cube.

Of course, we already knew how to do that stuff to an Instantiated Cube before, but this is using a cool member function built into it.

So overall, now that we know scripts are classes, we can treat them like normal classes. We fill them with useful member functions, possibly some hidden variables, and change Cubes by calling the functions.

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

32.2.3 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.

32.2.4 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.