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.
- Start and Update are private – they don't have `public` in front. So how is Unity able to automatically run them? It's using a trick we haven't seen yet.
- Like all classes, they don't exist until someone runs `new`. That's why our very first printing script needed to be dragged onto the camera. The

system runs `new textA()` automatically. Also like all classes, they can be `new`'d multiple times. That's why we can have several copies, or can accidentally double-drag 2 onto the same Cube.

- 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, like any other class.

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 can go onto a Cube, and might have a special Start and Update. It tells the system that a class is a "script".

That's also how our scripts for free get variables `transform`, `gameObject`, and `GetComponent`. They come with Monobehaviour.

32.2 Pointers to scripts

Since scripts are classes, we can have pointers to them. Suppose we have a script named `testA` on a Cube. Any other Cube can declare: `public testA ss;`, drag in the first Cube, and have a direct link to its script.

Even cooler than that, if all we have is a link to a gameObject, we can use GetComponent to find its script. The idea is: `GetComponent<Renderer>()` finds the Renderer. Renderer is a class. Our scripts are classes. So `c1.GetComponent<testA>()` to find the script on `c1`.

32.2.1 Direct script 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 unity 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 `public` member functions. This class waits around for people to call its functions:

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

If we drag this onto a Cube, it's as if we gave it 3 new commands: `turnRed()`, `turnYellow()` and `teleport()`.

This next script would go on some other object. It can reach out to 2 Cubes and run those member functions:

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

  void Start() {
    c1.turnRed(); // turn 1 red
    c2.turnYellow(); // turn the other yellow
  }

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

`public cubeScriptC c1, c2;` is a normal variable declaration, for a class we made. It's the same as `Dog d1, d2;`. The magic is how Unity allows us to drag a Cube into them. If it has `cubeScriptC` on it, Unity hooks up a pointer.

In Start, `c1.turnRed();` is a normal member function call. It runs `turnRed()` on `c1`.

As a check, we can type `c1`-dot. It pops up member functions `turnRed`, `turnYellow`, and `teleport`, just like a real class.

32.2.2 GetComponent to find scripts

We can do the same trick with cubes we create using Instantiate. If it has a script, we can find it with `newThing.GetComponent<scriptName>()`.

Assume we have a Cube prefab, with `cubeScriptC` on it. This script creates 5, grabs the script on each, and uses it to set up 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); // standard instantiate
      cubeScriptC cc=gg.GetComponent<cubeScriptC>(); // <- get the script
      cc.teleport();
      cc.turnYellow();
    }
  }
}
```

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. The player's shooting function:

```
  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); // set two variables
    bs.endTime = Time.time+4.0f; // on the new ball
  }
}
```

On to the `flyingBall` script. We want the balls to live for a few seconds, then wink out of existence. When the player set `bs.endTime = Time.time+4.0f;`, that ball is being given 4 seconds to live. Update on the balls checks that time:

```
class flyingBall : MonoBehaviour {
  public float endTime; // set when made
  public int hurtAmt; // set when made. How much we hurt the target

  void Update() {
    if(Time.time>endTime) Destroy(gameObject);
```

```
  }
// not done
```

Each ball has its own copy, with its own `endTime`. We can safely have multiple balls with multiple wink-out times.

The rest of the plan was for the balls to hurt targets when they hit one. We'll use `OnCollisionEnter` for that. We used it way, way back, so you may not remember it. It runs when we hit something:

```
  void OnCollisionEnter(Collision cc) { // in flyingBall
    // 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 trick. Only the target Cubes have the `target` script on them. So, we can check for a target by trying to get that script. Not finding it isn't an error – as usual it simply returns `null`.

If we hit a target, we reach over to it and run its `getHurt` member function (which we'll write, below). Recall `hurtAmt` was originally set when the player dropped the ball. It's worked its way to finally lowering a target's health.

The `target` script mostly sits around waiting for someone to call its `getHit` function. It should wiggle a little, and possibly die:

```
class target : MonoBehaviour {
  private int health=10; // hits subtract, until <=0
  private bool isDying=false;
  private int wiggle=0; // # of frame 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, to make `getHit` shorter:

```
  private void beginDie() {
```

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

The rest is handling wiggling and shrinking when we're dead. The style should be familiar – an Update calling a function for each task:

```
  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;
  }
}
```

Altogether, we've got the player class talking to the ball class, and the ball class talking to the target class. Only one time each, but the ability to find another class and set a variable or run a function is what makes this work.

Testing the end of a sequence can often be a challenge – testing how the targets wiggle and die without having to play the game each time. That's what the commented-out lines are for. They totally cheat, skipping past moving and dropping and colliding. Pressing A lets us see a wiggle, and X tests a hit and possible death.

I'm 99% sure that most game cheats and hidden power-ups were put in for testing, and left in by mistake.

32.2.4 Hand-running fake updates

That dropping balls example is a typical loose set-up. Everyone does their thing, talking at various times, with no one in charge. That's great when it works, but sometimes you need one master program to run things. That's a very traditional approach, which we can set-up in Unity.

Suppose we want 8 balls which occasionally change color. We could write one script for that, with an Update. Each ball runs itself. Or, we could get rid of the Update. Each ball waits for instructions from a master program. It's the same idea as the turn-red-turn-yellow example, but more systematic.

The class for the balls is merely useful color-changing, waiting for someone to call them:

```
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;
  }
}
```

It uses some Unity built-ins (checking the time, set color), but mostly looks like a normal non-Unity class.

The main program creates 8 balls from a prefab (which has the `colorBall` script in it), saves links, and hand-runs the 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; // put at random location
      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>(); // get script
      cc.setStartCol(); // calls that ball's setup function
      C.Add(cc); // save the link in our list
    }
  }

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

Start is nothing special – we've made and placed 8 things before. Update is the interesting part. Notice how it looks like using the interface of a normal class – we're using only functions, and they handle the details.

I feel like `C[i].nextCol();` would have been a complicated line a few chapters ago. Hopefully by now it's fine – find `C[i]` and run its `nextCol()` function.

The advantage of this approach is being able to more easily control events. Suppose we can pause, and also the player can freeze one ball

```
  void Update() {
    for(int i=0; i<C.Count; i++) {
      if(isPaused) continue; // paused means no color changes
      if(i==currentPlayerBall) continue; // don't change ball player is on
      if(C[i].isReady()) {
        C[i].setNextTime();
        C[i].nextCol();
      }
    }
  }
}
```

The other advantage is seeing everything that happens. It's all in the master Update function. If we skip one particular Cube sometimes, the code for it is right there.

We use these tricks in all sorts of programs. Classes that sit there waiting for calls are great. The `target` script has an Update with wiggle&die, but that's merely animation – it's a sit-and-wait class. "Fire-and-forget" objects are also useful in lots of situations, as long as they are mostly independent. The `flyingBall` class needed us to give starting values, but was fine running itself after that.

After seeing a few examples of each, you can usually figure out the idea behind various arrangements of classes, and how they should work together.