Chapter 39

Inheritance

After playing around with Unity for just a little while, you probably figured out *component* is the word it uses for things you can add to gameObjects. For example, rigidbodies are in the Component menu. You probably also noticed the Inspector doesn't seem to have preassigned slots for each type – it acts like one list with combined meshes, scripts and colliders, all jammed side-by-side.

You may have looked it up and seen component isn't just a word for humans. There's an actual C# class named `Component`, and the Rigidbody class also counts as a Component. What you see in the Inspector really is a single List with different classes in it. That's illegal, except it's somehow fine because it's also a list of `Component`'s, which everything also magically counts as.

That trick is accomplished using *inheritance*.

There are three parts to inheritance. Part one is the rules for making a class that grows from another one. Part two is making a pointer that can aim at either class. Part three is about using that pointer to call functions in a nice way.

Part one is mostly useless by itself, but we have to know it for part two. I'll write examples, but don't try to figure out how they'd be useful for real. Part two is how Unity accomplishes the single list of different types of things trick. But part three is where you finally see a real example and can understand why we invented inheritance.

39.1 Pre-inheritance example

The simplest, most basic use of inheritance is making two classes which have a lot in common. For example, Cats and Dogs will share name, age and weight.

We can do that pretty well using tricks we already know.

We might split off common animal data into a "helper" class, maybe named `Animal`:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now we can make the `Cat` and `Dog` classes using an `Animal` plus specific variables for that one kind. There are no new rules here yet:

```
class Cat {
  public Animal A;
  int cuteness;
}

class Dog {
  public Animal A;
  public float barkVolume; // in decibles
  public string favoriteChewToy;
}
```

The advantages of this idea are probably obvious – it's basic "don't write the same thing twice." But I'll list them anyway:

- Shorter code (especially if we have lots in common, and create more than just Cat and Dog.)
- Can be easier to read. Once you get used to it, `Animal A;` is the shortest, clearest way of saying a cat has all the basic animal variables.
- It ensures they're all spelled the same. The weight is automatically `float wt` for every animal. It can't accidentally be `int wght` for one.
- Easier to add a common stat. Anything you add to `Animal` automatically goes into `Cat` and `Dog`. It also makes sure all they stay in synch.

Another benefit is we can pick out only the `Animal` part. For example getting a pointer to it, or sending it to a function:

```
Animal a1;
// can point to a cat's or dog's animal:
a1=c1.A; a1=d1.A;

void printAsTons(Animal a) { print("Tons="+a.wt/2000; }
printAsTons(c1.A); // tons of Cat
printAsTons(d1.A); // tons of Dog
```

There are two sort-of drawbacks. We have to remember to `new` the `Animal`, or else get a null reference error. Not a big deal – we'd just put that in the constructor (this has nothing to do with inheritance, but it is a good constructor example):

```
class Cat {
  public Animal A;
  public int cuteness;
  public Cat() { A=new Animal(); }
}
```

The other annoyance is having to use the `A` for some variables. For a cat, the cuteness is just `c1.cuteness`. But we have to remember the name is `c1.A.name`.

And, to repeat, there's no inheritance yet. This part is showing that we can do a pretty good job having two classes share common data, using the old rules.

39.2 Basic Inheritance

The simplest Inheritance rule makes sharing common variables just a little easier. We start exactly the same as before, by making a class for what they have in common. I'm copying `Animal` here, with no changes:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now, the same as before, we want to use this to make `Cat` and `Dog`.

The new Inheritance rule says you put `: Animal` after your name (that's a full colon.) Doing that directly injects everything from Animal into you:

```
class Cat : Animal {
  public int cuteness;
}

class Dog : Animal {
  public float barkVolume;
  public string favoriteChewToy;
}
```

Now Cat and Dog have name, age and wt directly inside them. You don't need to make up an extra name, or use an extra `new`. The `Animal` variables are magically inserted.

To anyone using `Cat`, it looks like a regular class with four variables:

```
Cat c1;
c1.name="Mr. Boots"; // declared in Animal, but used like it was in Cat
c1.age=12; // same
c1.cuteness=4;
```

For some technical terms: we say Cat and Dog inherit from Animal. We'd call Animal the *base class*. It's still just a regular class, but in relation to Cat we'd say it's the base.

We also sometimes say Animal is the *superclass* and Cat is the *sub-class*. That can be confusing, since the subclass has more than the superclass. But everyone uses super and sub that way.

An example with nonsense classes:

```
class Furf { public bool ready; }
class Harhar : Furf { public bool done; }
```

`Harhar` acts like a regular class, with two variables. But `Furf` is still a regular class with just `ready`:

```
Furf f1 = new Furf(); f1.ready=true;
Harhar h1 = new Harhar(); h1.ready=false; h1.done=false;
```

This is a pretty simple rule, so far. Not very tricky, but also not much of an improvement over what we could do without it.

39.3 Intro to Polymorphism

The major advantage of using the official inheritance rule is that a `Cat` also counts as an `Animal`. This is a completely new thing, it's the real reason we invented inheritance, but it takes some explaining why it's so good.

Here's a non-useful example just showing the "counts as" rule. We can make a `Cat` or `Dog`, and use an `Animal` to point to it:

```
Cat c1 = new Cat();
Dog d1 = new Dog();

Animal a1 = c1; // <- animal pointing to a cat. this is legal!
a1.name = "Biggles";

a1=d1; // <- same animal points to a dog. also legal
d1.name="Spot";
```

This isn't just technically legal – it really accomplishes what it looks like. An animal pointer can reach into a Cat to change the name, then do the same

thing for a Dog. Because they inherit from it.

The same trick works with function inputs. The Animal-input function from before can now directly take a Cat or Dog:

```
void printAsTons(Animal a) { print("Tons="+a.wt/2000; }

printAsTons( c1 ); // cat input
printAsTons( d1 ); // dog input
```

It's really the same trick. Inside the function, `Animal a` is allowed to also point to a Cat or Dog, since they count as Animals, because they inherit from it.

The trick also works for an array of `Animal`'s. For example, this function finds the longest name in a list of Animals:

```
string longestName(Animal[] A) {
  if(A.Length==0) return "";
  string longest=A[0];
  for(int i=1; i<A.Length; i++)
    if(A[i].name.Length>longest.Length) longest=A[i].name;
  return longest;
}
```

We can run this using an array of Cats or Dogs. It works because each item is an Animal, and Cats and Dogs count as Animals.

The other array trick is using an array of Animals to hold Cats and Dogs, mixed up:

```
Animal[] MyPets = new Animal[3];
MyPets[0] = new Cat(); // putting a Cat into a animal array
MyPets[1] = new Dog(); // and a Dog
MyPets[2] = new Cat(); // another Cat
```

So that's enough about arrays. Let's finish up with restrictions on this trick:

- Being brother and sister types doesn't mean anything. Cat variables can't point to Dogs or vice-versa.
- You're only allow to do stuff based on the type of variable. The real type you're pointing to doesn't matter.

That second rule sounds complicated, but it's obvious once you figure it out. Here's a typical use where `Animal a;` switches between a Cat and Dog:

```
Cat c = new Cat();
Dog d = new Dog();

Animal currentPet=c;

  // switch pets:
  if(currentPet==d) currentPet=c;
  else currentPet=d;

  // use currentPet:
  currentPet.age++;
  if(currentPet.wt>30) ...
```

Using the Animal parts – `name`, `age` and `wt` – is safe, since anything it could point to will have those. But `currentPet.barkVolume` would be an error, since it might not be pointing to a Dog.

`Animal a;` can only do things Animals can do.

Even this is illegal:

```
Animal a = new Cat(); // fine, so far
a.cuteness=6; // ERROR
```

We know `a` is pointing a Cat, but it's still an Animal pointer, so is only allowed to do Animal things.

Functions are the same way:

```
void doAnimalThing(Animal ani) {
  if(ani.cuteness<8) ... // ERROR
  ...
}
```

We can call that with Cats or Dogs, but it's only allowed to do Animal things to it's input.

39.4 Dynamic casting

To review the previous section: you can aim `Animal a1;` at a Cat or Dog, but you can only use it for common Animal stuff. And that's fine most of the time. We can have a list of mixed Cats and Dogs, and an Animal function can find the total weight, or make sure none have the same name.

But sometimes, not as often as you'd think, you want to check what `a1` is really aimed at. And there's one more problem after that. Even if we know `a1` is aimed at a Dog, `a1.favoriteChewToy` is still an error.

One new command solves both problems. `a1 as Dog` checks whether `a1` is a Dog. If not, it returns `null`. If it is, it returns a `Dog` pointer, aimed at `a1`.

An example, then more comments:

```
Animal a1;
 ...
  a1.age++; // works for any animal

  // check for a Dog and do Dog stuff:
  Dog d1 = a1 as Dog;
  if(d1!=null) d1.barkVolume--;

  // ditto, with Cats:
  Cat c1 = a1 as Cat; // is it really a Cat?
  if(c1!=null) c1.cuteness++;
```

`d1 = a1 as Dog` does two things in one. If `d1` is null, it lets us know `a1` wasn't a Dog. That's how the `as` command works. Otherwise it says "now that you know `a1` is a Dog, I'll bet you want to change it? I'll aim `d1` at it, so you can do Dog stuff."

Here's the required silly analogy, from the Seinfeld TV show: suppose you saw someone's phone number on a charity walk mailing list, and wanted to call them for a date. You still have to ask them "can I have your phone number?"

This adds the cuteness of the Cats in a mixed Cat/Dog animal array. It's the same rule, but checking `A[i]` instead of `a1`:

```
int totalCuteness(Animal[] A) {
  int cuteSum=0;
  for(int i=0; i<A.Length; i++) {
    Cat c = A[i] as Cat; // check for a Cat. If it is, c points to it
    if(c!=null) cuteSum+=c.cuteness;
  }
  return cuteSum;
}
```

If we only want to check for the type, we can skip saving the extra pointer. This separately counts how many Cats and Dogs are in an array:

```
void animalChecker(Animal[] A) {
  int cats=0, dogs=0;
  for(int i=0; i<A.Length; i++) {
    if((A[i] as Cat)!=null) cats++;
    else if((A[i] as Dog)!=null) dogs++;
  }
  print(cats+" "+dogs);
}
```

We could have done it the long way. But after a while checking "is it a Dog?" on one line tends to look nicer. Saving a variable you don't need might

confuse someone for an extra second.

There's no reason to do this next thing, but if we know `a1` is a Cat, we could use the shortcut to change part of it: `(a1 as Cat).cuteness=5;`. It would crash it wasn't a cat.

But suppose we had a function taking a Cat input, which checked for `null`. Calling `doCatStuff(a1 as Cat)` would be fine.

The technical term for `a1 as Cat` is dynamic cast. It's not exactly a cast, since it doesn't change anything (if it works, it returns the same pointer.) But it changes the pointer type, so it feels cast-y.

Dynamic is a general term for anything involving inheritance that you have to look up while you're running. The opposite is *static*, which means things the compile can do ahead of time.

As you might guess, this only works with inheritance. `int n = f as int;` doesn't even make any sense (just use `int n = (int)f;`).

39.5 Inheritance and functions

You also inherit functions from the base class. It works the usual way – they count as being directly inside of you. Here's a quick, boring example. Cat has two functions: `cStats()` from itself, and it inherits `aStats` from Animal:

```
class Animal {
  ...
  public string aStats() { return "name: "+name+" "+age+" yrs old"; }
}

class Cat : Animal {
  ...
  public string cStats() {
    return "cute: "+cuteness;
  }
}
```

We'd use both of these as if they were inside `Cat`, without having to know which used inheritance:

```
Cat c1 = new Cat();
string w1 = c1.aStats(); // inherited, but called normally
string w2 = c1.cStats(); // normal, and called normally
```

In short, inheriting functions from the base class works exactly how you'd expect, with no new rules or surprises.

There's one new rule: you're allowed to cover up a function with another one. For example, Animals have a standard cost, Dogs use it, but Cats are on sale:

```
class Animal {
  public int cost() { return age*3; } // just any formula for cost
}

class Cat : Animal {
  public int cost() {
    int price = base.cost(); // use cost from Animal
    if(age>1) price/=2; // sale on adult cats
    return price;
  }
}

// Dogs don't have another cost()
```

This works out pretty well. Most types of animals would probably inherit the basic `cost()` from Animal. But some can cover it up with their own. That's usually called overriding.

From inside of the class you're allowed to use the covered-up one: `base.cost()` looks it up. The way Cat.cost looks up the basic Animal cost and tweaks it is typical.

Constructors and base constructors

Skip this section if you don't like constructors. Otherwise this is a neat example of the `base` trick. If you inherit from something, you probably want to use its constructor to help with yours, so a special rule makes it easier.

Here a standard Animal is 2 years old. A standard Cat uses that, then some extra Cat stuff:

```
class Animal {
  ...
  public Animal() { age=2; } // constructor
}

class Cat : Animal {
  ...
  public Cat() : base() { cuteness=5; } // also sets age to 2
}
```

Putting the colon-base in-between like that is a completely special rule, just for this. I like it since it shows how much we love the "borrow stuff from the thing I inherit from" trick.

39.6 dynamic dispatch

This is the rule that makes inheritance and polymorphism worth it.

Suppose we have an Animal pointer aimed at a Cat and call `a1.cost();`. Animals and Cats each have their own cost function, so which do we use?

```
Cat c1 = new Cat();
Animal a1 = c1;

a1.cost(); // run the one in Animal, or in Cat?
```

If we want the right price, we should base it on the real type. In other words, we should do the extra work to check the real thing `a1` points to, and run that version of `cost`.

That extra look-up, each time it runs the line, is called Dynamic Dispatch. Dispatch is another word for a function call, and dynamic is about how we don't know it ahead of time.

To double-check this way is how we want, this finds the total cost of Animals in a list full of Cats and Dogs:

```
int totalCost(Animal[] A) {
  int cost=0;
  for(int i=0;i<A.Length;i++)
    cost += A[i].cost();
  return cost;
}
```

Without dynamic dispatch, this wouldn't care about which type of Animals we're trying to buy, and give wrong costs.

It also shows off the dynamic part. Each time the loop runs, `A[i].cost()`, might run a different function. Imagine we have many more Animals besides Dogs and Cats. That's a lot of work we get, for free.

This is also why we almost never need to hand-check the real type using `as Cat`. We usually have dynamic functions do all of that for us.

Various languages have different rules to enable this. Java does it automatically. In `C#` you need to add extra words. `virtual` goes in front of the function in the base class, and `override` goes in front of all the ones hiding it:

```
class Animal {
  ...
  virtual public string stats() { return "animal"; }
}
class Cat : Animal {
  ...
```

```
  override public string stats() { return "cat"; }
}
```

If we don't add both things, `virtual` and `override` in the correct spots, we get either errors or the "other" behavior – `a1.cost()` always calls the Animal version. That way is easier for the computer, but almost always gives results we don't want.

Some notes:

We still can't call non-animal functions this way. For example, if only Dogs have a `needsWalk()` function, you still can't call `a1.needWalk()`. It has to be a function that's in Animal, and then covered up in a subclass.

It also only makes sense if we start with an Animal. For example `Cat c1;` can only ever point to a Cat. There's only one possible cost function it can call, so this trick does nothing for us.