Chapter 35

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 `Mesh` class also counts as the `Component` class. What you see in the Inspector really is a single list with different classes in it, which is somehow legal because rigidbodies and even your scripts officially also count as `Component`s.

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.

35.1 Pre-inheritance example

The simplest, most basic use of inheritance is when you want to create two classes which have a lot in common. For example, Cats and Dogs will each have name, age and weight.

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, with the exact same spelling as every other Animal.
- 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 split off just the `Animal` part. We can have a pointer to it, or send it to a function:

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

void animalTons(Animal a) { print("Tons="+a.wt/2000; }
animalTons(c1.A); // tons of Cat
animalTons(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, and just having to remember it's an `A`. `c1.cuteness=5;` is easy. `c1.A.name="Henry";` isn't as nice.

35.2 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.

Here's one teaching example using inheritance, with nonsense classes:

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

`Furf` is a regular class, that we can use the regular way. `Harhar` also acts like a regular class, with two variables:

```
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.

35.3 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.

Before explaining the rules more, here's a useful example. This ordinary function finds the longest name in an array of `Animal`s:

```
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;
}
```

The special part is we can call it with arrays of Cats and Dogs, since they count as Animals:

```
Cat[] CList = new Cat[10]; // pretend we fill this with cats
string w = longestName(CList);
// same for an array of Dogs
```

This counts as legal, but also should make sense. The important part of the function is where it uses `A[i].name`, which is the `name` variable of an Animal. Cats and Dogs have that exact thing in them.

Saying they count as animals is like saying you can look at a Cat and squint, to see just the Animal part.

We can also mix `Cat`s and `Dog`s in the same `Animal` array:

```
Animal[] MyPets = new Animal[4];
MyPets[0] = new Cat(); // putting a Cat into a animal array
MyPets[1] = new Dog(); // and a Dog
MyPets[2] = new Cat(); // and so on
```

Before, we could never make an array of mixed types, now we can. To the computer, it's just an array of `Animal`s, so it's happy. We could even send this mixed array into `longestName` and it would work just fine.

Just to be clear, we can't put `Dog`s into an actual `Cat[]` array, or vice-versa. And we can't send an `Animal[]` into a function wanting a `Cat[]`. The only special rule we have is that `Cat`s and `Dog`s count as `Animal`s, because of inheritance. Doing it with an array is just a useful way to use that rule.

The rule for "downcasting" like this is you still have to respect the official type, and can only use the things it has. That sounds more complicated than it is. It means that if you're using an `Animal` pointer, you can change name, age or wt. You can't change cuteness, even if you know it's really pointing to a Cat:

```
Animal a1; // this could be pointing to a Cat or Dog
a1.age=6; // legal
a1.barkVolume=39; // ERROR (even if it really points to a Dog)

MyPets[3].name="Fritz"; // legal
MyPets[3].cuteness=6; // ERROR, Animal doesn't have cuteness
```

That rule is really very natural. One way to see it is that if we have `Animal a1;`, the normal rules say it can only use `Animal` variables. Being able to aim it at a `Cat` or `Dog` doesn't change that.

Another way to think of it is that if an `Animal` can go back and forth pointing to a `Cat` or `Dog`, it can only logically use things they have in common.

35.4 Dynamic casting

Now that an Animal pointer can aim at a Cat or a Dog, we have a new problem. How can we go back? More formally, how can we tell if `a1` is really pointing to a Cat?

Then we have one more problem. Even if we know `a1` points to a Cat, the computer doesn't. `a1.cuteness` is still an error. We'll solve both problems at once.

What we really want to do is create a fresh cat-pointer, write `c1=a1;`, then check if it worked. If it did, we can use `c1` to do Cat stuff. If not, then `a1` wasn't pointing to a Cat. Here's a working example doing that, using our one new rule:

```
void animalFunc(Animal a1) {
  print(a1.name); // warm-up. This part is always fine

  Cat c1 = a1 as Cat; // <- new rule
  if(c1!=null) print(c1.cuteness);
}
```

Using a function is just a way to establish how `a1` could point to just an Animal or a Cat or a Dog. The last two lines are the new part – the standard way to check for a cat. "`as`" is a new keyword.

`c1=a1 as Cat;` says we know `a1` might be a `Cat`, or might not. It says to try `c1=a1;`. If we can't do it, since it isn't a Cat, just make `c1` be `null`. The last line, `if(c1!=null)` is really asking "did it work?"

Technically, it combines three ideas. The first thing is it checks for a Cat. The second is it doesn't just say yes or no. For yes we get the pointer, and for no we get `null`. The last part of the idea is it converts an Animal pointer into a Cat pointer. The place the pointer goes doesn't change, but instead of an

`Animal` pointer to a `Cat`, we get a `Cat` pointer to that same cat.

Here's a working example using that trick to add the cuteness of just the Cats in an array, skipping any Dogs:

```
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 want to know the exact sub-type, there's no special way. We just use the "`as`" trick as many times as we need. This separately counts Cats and Dogs:

```
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);
}
```

I got a little sneaky, but it should be obvious what's going on. The real command is `A[i] as Cat`, which returns `null` if it wasn't a Cat (it returns `A[i]` if it was, but we don't care about that here.) Usually we assign it to a variable, but we can cut out the middleman by putting it directly inside the `if`.

Formal rules and examples

This is probably obvious: `as` only works with inheritance chains. In `a1 as Cat`, the first thing has to be a pointer of some base class, and the second thing has to be the name of a subclass. In other words, it has to be a question where the answer really could be yes or no.

For example, `c1 as Dog` is an error. A Cat can never point to a Dog so there's no reason to ask.

`as` won't work with non-inheritance stuff at all. This first part is total junk (for fun, the last part is how to do it, but has nothing do to with inheritance):

```
float g=3.0f;
int n = g as int; // error - only works with inheritance

if((int)f==f)
  print("f ends with point-0"); // sneaky way to check for point-0
```

When you assign to a variable, like in `Dog d1=a1 as Dog;`, you need the same type at both ends. Here's an example to try to confuse you about that:

```
Animal a1 = new Cat(); // animal variable, pointing to real cat
Dog d1 = a1 as Cat; // syntax error -- can't assign cat to a dog
Dog d2 = a1 as Dog; // legal (will be null)
```

Since we know it's a Cat, `a1 as Cat` seems right. But the point of those lines is to check whether it's a Dog. That means we have to try to turn it into a Dog using `a1 as Dog`.

Then here are some notes that might help, but aren't anything you need to know:

- `a1 as Cat` is called dynamic casting since it's casting with inheritance logic. Whenever something looks through an inheritance chain to see what to do, we often call it dynamic.

  The casting part is much simpler than we usually do. Normal casting, like `(int)f` returns a changed value. But for this, you either get what you started with, or `null`. It's technically casting since we get an upgrade from `Animal` to `Cat`.

- `c1=a1;` is an error for the regular reason – we can't assign an `Animal` to a `Cat`. The types aren't an exact match. The special polymorphism rule only goes one way. To review the special rules are:
  `a1=c1;` is special, and always works. And `c1 = a1 as Cat;` is special, always legal, but might be `null`.

- Checking for `null` isn't required. Like everything else, you don't have to check if you know it's safe. `(a1 as Cat).cuteness=5;` is legal. But if it wasn't a cat, `a1 as Cat` would be null. Then dot-cuteness gives the usual null reference exception.

35.5 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: the one in it, and the one it got 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() + " " + c1.cStats(); // both look like normal Cat functions
```

Things work the same inside other `Cat` functions. We call "our" functions and the ones we inherit the same way:

```
class Cat : Animal {
  ...
  public string allStats() {
    string s1 = aStats(); // inherited from Animal
    string s2 = cStats(); // defined in Cat
    string result=s1+" "+s2;
    if(wt>10) result+=" heavy";  // use inherited wt variable like normal
    return result;
  }
}
```

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

One thing to note is that inheritance only goes one way. Functions in Cat can use variables in Animal. But functions in Animal are completely normal. Animal by itself never "knows" about things that inherit from it.

There is one new rule for inheritance and functions. You're allowed to re-use a function name in the subclass: you can have a `stats()` in `Animal` and another in `Cat`.

The rules for how that works are: 1) the original is hidden from people outside the class, and 2) inside the class, you can use them both. Use yours normally, and the covered-up one by adding `base.` (base-dot).

Here's a teaching example. Goat and Blob inherit from Animal. There's a `stats` function in Animal, and another in Goat covering it up:

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

```
class Goat : Animal {
  ...
  // same name. Covers up stats in Animal:
  public stats() { return "baa "+name+" baa"; }
}

class Blob {
  ...
  // not writing a stats() here. We get the one from Animal, like normal
}
```

An outside user sees one `stats` function for each class. Blob inherits the one from Animal. Goat didn't like that one, so used the cover-up rule to hide it with a better one. Outside the class works like this:

```
a1.stats(); // the one from Animal
b1.stats(); // the one from Animal
g1.stats(); // baa - the one from Goat
```

The plan is that some animals will like the basic `stats` function. They can do nothing and get it for free. Others will want their own version, so can overwrite it. The end-user only sees what they need to, and doesn't need to know how it got that way.

This example shows the second rule, how to use both versions from inside the class:

```
class Cat : Animal {
  ...
  public string stats() {
    string w=base.stats(); // run the one from Animal
    return w+", cute: "+cuteness;
  }
}
```

This is pretty common. The one covering it up uses the first one as a helper function. It's typical inheritance thinking. We use `stats` in Animal to do the basic work, and the one in Cat adds on the extra work.

The word `base` in front comes from how Animal is the base class. If you remember, we also call it the super class. Some languages use the word super instead of base, to reach back like this.

Here's one more with the same idea, but using numbers:

```
class Animal {
  ...
  public float cost() { return age*5+wt*2; } // made-up cost formula
}
```

```
class Cat : Animal {
  // cats cost 25% less than most animals:
  public float cost() { return base.cost()*0.75f; }
}
```

I think `base.cost()*0.75f` is easy to read, and says what it means.

This is the same idea as with `stats`. Maybe some types of Animals are happy with the `cost` they inherit. Maybe they all cover it up and use `base.cost()` as a helper function. Maybe all Mice are 50 cents, so they cover up the one in Animal with just `float cost() { return 0.5f; }`.

Inside the class, we're allowed to use a covered-up function from anywhere, not just the function covering it up. In a real program we rarely want to, but we can:

```
class Duck : Animal {
  ...
  public string stats() { return "quack"; }

  public string stats2() {
    // call ours and the one in Animal:
    string w1 = stats() + base.stats();
    return w1;
  }
}
```

I think it's easy to see which is which in `stats()+base.stats()`.

One cool error, if you try to call the function you're covering-up, but forget the `base`-dot, you get an infinite recursive loop:

```
  // oops!! Meant to call base.stats(). Runs forever:
  public string stats() { return stats() + " mieo"; }
```

Constructors and base constructors

Constructors don't cover each other up, but they use the `base` rule in mostly the same way. I'd normally leave this as a detail you can look-up later, but it's a nice example of using `base` to reach back into your base class:

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

class Cat : Animal {
```

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

The syntax is a little funny. It's just colon-`base` in that spot. It looks like the inheritance rule, but they're re-using the colon to mean "run the base constructor first." You don't have to use it, but you can.

35.6 dynamic dispatch

Suppose we have an Animal pointer aimed at a Cat and call `a1.cost();`. It's not completely clear if we should call the `cost` function in Animal, or the one hiding it in Cat:

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

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

The question is, should we key off of the pointer type, or the real type? Both ways make some sense.

Keying off the pointer type follows the rules. `a1` is an Animal, so can only run Animal functions. It can't even see the `cost` function in Cat.

The problem is, doing it that way gives us the wrong price for the Cat. We purposely covered up the Animal cost function with a better one for Cats. The entire point is to always use it.

Some languages only use the good rule: functions are always based on the real type: if you're a Cat, you always get the Cat `cost` function, even if you call it using an Animal pointer.

The C# family of languages let you chose. You decide when you write the class. If you do nothing special, you get the bad rule, to key off the pointer type. To follows the real type, add two keywords, `virtual` and `override` (details later):

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

Now, when we run `a1.stats()`, it checks the real thing `a1` points to. It runs the Animal version for real animals, and the Cat version for real cats. If we left out those two words, then `a1.stats()` would always call the Animal `stats` function.

Here's an actual example showing all these rules working together. It takes another array of Animals, which we know will be an array of Cats, or Dogs, or possibly mixed Cats and Dogs; and finds the total cost:

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

It only works if we write `cost` using `virtual` and `override`. Then `A[i].cost()` would compute the correct cost for that Dog or Cat.

If we have several covered-up functions, we have to put `virtual` in front of every one of them in the base class, and `override` in front of every covering-up subclass function. This is a pain. Here's an example where `stats` and `cost` are both virtual:

```
class Animal {
  ...
  virtual public string stats() { return "animal"; }
  virtual public float cost() { return wt*5+age*2; }
}
class Dog : Animal {
  ...
  override public string stats() { return "dog"; }
}
class Cat : Animal {
  ...
  override public string stats() { return "cat"; }
  override public float cost() { return base.cost()*0.75f; } // cats on sale
}
```

Notice how `Dog` didn't write its own `cost` function. As usual, that's fine. If you don't cover a function up, this new rule doesn't even matter. Dogs want to use the basic `cost` in Animal, which is simple and not a problem.

Also notice how the Cat `cost` is still using the one in Animal as a helper. That's still perfectly fine, and works like it did before.