Chapter 35

More Inheritance

The last section was all the basic ideas about inheritance and polymorphism. This section has a few examples, reasoning, and little rules to tweak things.

35.1 A story about the terms

The particular rules and terms C# uses are strange, and probably not worth knowing at first. But if you want to know why things are the way they are, which you don't need to, here's a story:

In the old days we didn't have built-in inheritance. But we had the idea of it, and hand-hacked it into our programs. You made one class, with an `int` for which type it counted as. Then you jammed the class full of every variable for every type of Animal. Then you used `if`s to call member functions for the particular animal you thought you were. Like this:

```
class Animal {
  public int aType; // 0=cat, 1=dog, otherwise generic animal

  // common stats, same as normal:
  public string name;
  public int age;

  // stats for every animal:
  public int cuteness; // only for cats
  public string chewToy; // only dogs

  public float cost() {
    if(aType==0) return catCost();
    else if(aType==1) return dogCost();
    else return animalCost();
  }
```

```
  // add cost function for each animal here:
}
```

Inheritance is just a way to build this trick into the language. Back then, it made total sense to say "you know all of those `if`s you write to call a function based on the real animal type you are? We're going to call that a virtual function and have it be automatic."

That's why call-by-pointer-type is the basic way, because if you know how things really work inside the computer, it is the simplest way. To the computer, that's just a standard function call. Call-by-real type (virtual functions) really are extra work for the computer. It made sense to have the extra `virtual`s and `override`s there to remind us of that.

That's the old detail-oriented way of doing things, giving you every option and making you think about how the computer sees things. C++ does that.

But the point of a computer language is to hide and simplify things you don't need to know. For example, Java (a newer language) gets rid of all those words. Everything is virtual, but you don't have to type anything extra, or even know that word. When you cover up one function with another, it just automatically works the right way.

They could have added a special word to put before functions to mean non-virtual. But they didn't even do that - they figured no one would ever use it. It would just clutter things up trying to explain it.

C# is based on Java and C++. They decided to start with the C++ way of inheritance, because, well, they just did.

35.2 Misc examples

There are several different tricks inheritance can let us do. Unity has some examples using a mix of tricks:

Unity's `Component` is a base class used to cram different things into an array. If you inherit from it, you don't get much except "I count as a Component." This is pretty much what `GetComponent<Rigidbody>()` does to find one:

```
Rigidbody getRBfromArray(Component[] C) {
  for(int i=0;i<C.Length;i++) {
    Rigidbody rb = C[i] as Rigidbody; // dynamic casting test
    if(rb!=null) return rb;
  }
  return null;
}
```

It's also like a label to help you. When you see `class Collider : Component` it's telling you that Colliders, because they're components, go on gameObjects.

Unity uses the `MaskableGraphic` base class as a way to double-up on two real classes that have a lot in common. The two ways to put a 2D image on the screen are `Image` and `RawImage`. They both have a color and material, so we made `MaskableGraphic` for them to inherit from.

Then we can do tricks using what they share:

```
public MaskableGraphic catPicture; // drag in Image or RawImage

  catPicture.color=Color.White; // works if we're an Image or RawImage

// works on Image or RawImage:
void turnRed(MaskableGraphic mg, Color cc) { mg.Color=cc; }
```

The `Monobehaviour` class is for a few things. It inherits from `Component`, so it's using that trick to put your scripts in a gameObject (it goes in the Component list, along with everything else.) It also acts as a label – Unity can loop over the component list using `C[i] as Monobehaviour` to grab all your scripts.

It's a way to give all your scripts a free `transform` and `gameObject` link, which is like giving all of your Animals a name and age.

And then, Start, Update and OnCollision act like virtual functions. They obviously aren't (since they don't have `public override` in front,) but they use another trick that acts that way.

A non-Unity semi-practical example. I want to make a list of Buttons, labels, drop downs and other things which can mostly be clicked. I'll make a base class `menuItem` which has a do-nothing virtual `click` function. Each subclass overrides `click` (or just keeps the base do-nothing click, if you can't click it):

Again, I'm just listing the important parts:

```
class MenuItem {
  public GameObject myGO;
  public virtual void click() {} // override this if your click does something
}

class Label : MenuItem {
  // labels are fine with doing nothing on clicks
}

class Toggle : MenuItem {
  // toggle make clicks flip between true and false:
  public bool value;
  override void click() { value = !value; }
}

class CycleButton : MenuItem {
```

```
  // clicks cycle 0,1,2, 0,1,2:
  int v, max;
  public override void click() { v++; if(v>max) v=0; }
}
```

`MenuItem` as a base class lets us use a `MenuItem[] M;` array for any of these. It also gives a common `myGO` variable. And, of course, `M[i].click()` depends on `click` being a virtual function.

35.3 Cutesy extra rules

Like everything else, inheritance is just a few real ideas (polymorphism and virtual functions) and then a lot of little tweaks and options. There's nothing wrong with these, but they're just a distraction until you know the basics.

35.3.1 Privacy

If you recall, `private` just is a way to help organize a big program. It's a nice way to hide variables from outsiders who shouldn't be using them anyway. So far, the one privacy rule is about what people outside the class can see.

With inheritance, we can make another privacy rule: when you inherit from a class, can you see its variables? We'll grow the privacy rules just a little for that option:

Here are the expended rules:

- `private` means only you. Things that inherit from you can't see your private stuff. They have them, but they can't use them.
- `public` still means everyone can see it – completely outside the class, or other classes that inherit from it.
- A new keyword `protected` means "public if you inherit, private to everyone else." The same as private, it can be used on variables or functions.

Just like the regular `public` rules, these do nothing. The only reason to ever use `protected` is if you think "I sure wish this Animal variable would be in the pop-up for Cat functions, but not for outside." If you never think that, never use `protected` and you'll be fine.

35.3.2 abstract base class

In the Animal, Cat, Dog example, I never use `new Animal()`. I could have, but it wouldn't make any sense to create just a generic Animal. A lot of inheritance is like that, and it's not a problem. But it wouldn't hurt to have an official word for it.

`abstract` in front of the class means you can never use `new` to create one. It does nothing useful except cause errors if you do. It doesn't prevent you from

declaring those variables. `Animal a1=new Cat();` is still legal.

An example:

```
abstract class Animal { ... }

class Cat : Animal { ... } // no change in Cat

Animal a1 = new Cat(); // legal
a1 = new Animal(); // ERROR
```

35.3.3 Inheritance chain, Multiple Inheritance

You're allowed to inherit from a class that inherits from something else. You don't have to list them all – you automatically get everything inherited by the one you inherit. FlyingCats has everything from Cats and Animals:

```
class Cat : Animal { ... }

class FlyingCat : Cat {
  public float airSpeed;
}
```

As usual, everything a FlyingCat inherits looks the same, like `fc1.name`, even though `name` came from way down in Animal.

To use virtual functions, you still put `virtual` in the top one, and `override` in all the ones covering it up. `protected` is unchanged – FlyingCats can see `protected` variables all the way down in Animal.

The other way to inherit from more than one class is to directly inherit from both, which is called multiple inheritance. You list them with commas (the order doesn't matter):

```
class SaleItem { public float price; public int numForSale; }

class Cat : Animal, SaleItem { ... }
class Eagle : Animal { ... } // eagles are not for sale
class Car : Vehicle, SaleItem { ... }
```

This is more flexible than an inheritance chain since we can mix and match. But C# doesn't allow it (trying to is an error.)

35.3.4 Misc

C# has two class keywords you can ignore: `sealed` and `new` (which is a totally different use than `new Cat()`.)

`sealed` adds a restriction that no one can inherit from your class. Like `private`, it doesn't do anything – just makes errors.

`new` is optional. Whenever you could put `override` but don't want to, you can put `new`. It means call-by-pointer-type, which is the one you don't want, and what happens if you just leave it blank.

35.4 Interfaces

Suppose we wrote a base class that had only do-nothing virtual functions, for example:

```
class public Breakable {
  public virtual void use(int times) {}
  public virtual bool isBroken() {return false; }
}
```

The idea is that `use(1)` has a chance to break the item, and `isBroken` is how you can check. But this class gives zero help in how they would work.

We can inherit from it, but we have to write everything from scratch. Here's a sample Hammer and Pen which break in different ways:

```
class Hammer : Breakable {
  // Hammers have a 2% chance to break each use:
  bool broke=false;
  public override use(int times) {
    for(int i=0;i<times;i++) if(Random.value<0.02f) broke=true;
  }
  public override isBroken() {  return broke; }
}

class Pen : Breakable {
  // Pens have 20 uses:
  int usesLeft=20;
  public override use(int times) { usesLeft-=times; }
  public override isBroken() { return usesLeft<=0; }
}
```

That useless base class still lets us use polymorphism tricks. This somewhat silly function will stress-test a Hammer or Pen, since it takes a Breakable:

```
int usesUntilBroken(Breakable b) {
  int timesUsed=0;
  // use until it breaks, or 1000 times (since some things never break):
  while(b.isBroken()==false && timesUsed<1000) {
    b.use(1); timesUsed++;
```

```
  }
  return timesUsed;
}
```

We can improve it just a little by making `Breakable` be `abstract`. There's an extra rule for abstract classes: putting `abstract` in front of a function means you can skip the body, everyone else has to write it, and it counts as virtual:

```
abstract class Breakable {
  public abstract void use(int times);
  public abstract bool isBroken();
}
```

That's really the purest form of the idea. If you inherit from this, you get nothing and are forced to write those functions. In return, you gain the ability for `Breakable b1;` to point to you and use `b1.use(1)` and `b1.isBroken()`.

This concept, just a list of virtual functions you will have, is often called an *interface*.

It's semi-common to inherit from one class for real (you get variables and useful functions,) then tack on an interface. For example, here we use regular inheritance to make Car and Shredder from Machine, and Bird and TurtleElder from Animal, tacking on the Movable interface where needed (this is almost legal C#):

```
// using as an interface:
abstract class Movable {
  public abstract void setTarget(Vector3 pos);
  public abstract void move();
  ...
}

class Car : Machine , Movable { ... }
class Shredder : Machine { ... }
class Bird : Animal, Movable { ... }
class TurtleElder : Animal { ... }
```

Notice how `Movable` can be added to one Machine, but not the other, and then also added to just one Animal out of the two. It's like we're taking a fresh look at all four, not caring if or what base class they used, and just adding Movable to the ones we'll want to move.

Of course, we get no help making them move. We have to hand-write `move` and `setTarget` for Car and Bird to make them really be Movables. But then we can aim `Movable m1;` at either, and call `m1.move();` on Cars and Birds.

Official Interfaces

Traditionally, interfaces are just a way of thinking – if you make a class with only do-nothing virtual functions, we say you're using the interface idea. But C# has an official thing called `interface`. On the one hand, this is just a detail, but I think it also shows the interface idea, so is worth seeing.

Here's Breakable rewritten as an official C# interface. Both versions are the same:

```
// old way:
abstract class Breakable {
  public abstract void use(int times);
  public abstract bool isBroken();
}

// new way:
interface Breakable {
  void use(int times);
  bool isBroken();
}
```

Since you told the computer it was an `interface`, the computer adds the rest (for example, the functions always count as public.)

You "inherit" from these the usual way, and are required to write the functions. You can even declare variables for interfaces, like `Breakable b1;`. That seems funny, but it's the same idea as abstract classes. `Breakable b1;` is for pointing to things that count as Breakables.

The exact C# rules are messy, but you can look up examples pretty easily. For example, you sometimes have to use the interface name as a namespace. Here I'll write `use` that way, but `isBroken` the normal way:

```
class Hammer : Tool , Breakable {
  ...
  int usesLeft=20;
  void Breakable.use() { usesLeft--; }
  public bool isBroken() { return usedLeft<=0; }
 }
```

You really do leave out the `public` the first way, but have to write it the second way; and you can't write `override` but it counts that way. It's a mess. But the error messages are pretty clear, and there's only one way it works. As long as you understand the "I will write these virtual functions, then will count as your type" idea, you'll get it right after a few tries.

Built-in interfaces

We already know how to sort an array using function passing. If you remember, if looks like `Sort(C, youngerCatCompFunc);`, where the second input is a 2-item compare function.

But C# has another way to sort using interfaces. It's not as good, but it makes a nice example.

The first two parts are built-in to C#. There's an interface holding a compare function. It looks funny - the `T` stands for whatever type we are, like a Cat. It only has one input, but it really has two since it's a member function (more about that later):

```
interface IComparable<T> {
  int CompareTo(T t);
}
```

It should return -1/0/+1, which is standard compare function use.

The built-in Sort takes an array of these, and uses the `CompareTo` function to figure out the order. Here's the part we care about:

```
void Sort(IComparable<T>[] A) {
  ...
    // Compare A[i] and A[j]:
    if(A[i].CompareTo(A[j])==1) // out of order
  ...
}
```

The trick is we'll really being giving it an array of Cats or Dogs, or anything that counts as `IComparable` (they don't, yet, but we can make them.) Sort doesn't need to care about exactly what they are, all that matters is they have `CompareTo` written.

Comparing using a member function looks funny - we'd rather use a regular 2-input function, like `Compare(A[i],A[j])`. But interfaces only give us member functions.

We can use those built-ins to sort an array of Cats. Write the `CompareTo` function in Cat, and put `IComparable` in your inherit list. This makes a Cat sortable by age (figuring out the extra Cat in angle-brackets and the other stuff took some trial&error):

```
class Cat : Animal, IComparable<Cat> { // <= added IComparable
  ...
  int System.IComparable<Cat> CompareTo(Cat c2) {
    if(age<c2.age) return -1;
    if(age==c2.age) return 0;
```

```
    return +1;
  }
}
```

Now `System.Array.Sort(Cats);` is legal, and sorts our Cat array. Notice how there isn't an option how to sort. It always uses the one we wrote in the class, which compares by age.

This is a semi-common trick. But if you hate this example, sorting by passing in a compare function is better anyway.

35.5 General inheritance advice

Try to use inheritance for a specific purpose. The main reason for it is when you need a pointer that can go to either, or an array that can hold either.

Suppose you have health pickups, ammo pickups and so on. If you write a simple `Pickup` base class, you can do things like:

```
public Pickup nearestPickup; // health pack or ammo

public Pickup[] AllPickups; // all health and ammo, together
```

Then you might naturally move variables like `timeOut` into the base class. That lets you use `if(nearestPickup.timeout<5)` easily without having to check what kind it is.

A secondary reason for inheritance is you just notice two classes have a lot in common. This was my first Cat/Dog example. Don't think too hard about this type, or pretty soon you'll be making Animal, Mammel, Feline classes that don't help you much.

If several classes have a lot in common, it makes sense to put all that into a common class. Then inherit it, or just declare `public Animal A;` in your class. If you aren't using polymorphism tricks, it won't really matter.

For very simple stuff, a type field is fine. In the Cat/Dog example, suppose we want to distinguish a Cat from a Dog, but we don't need cuteness or barkVolume. In that case we wouldn't need sub-classes. Just the Animal class with `public int animalType;` would be fine (or make an `enum`.)

You can have a base class and be a MonoBehaviour. In the Health pickup example, you may have started with `class HealthPickup : MonoBehaviour`. If you want to add `Pickup` as a base class, do this:

```
class Pickup : MonoBehaviour {
  public float timeout;
  ...
```

```
}

class HealthPickup : Pickup {
  ...
}
```

That's just a standard inheritance chain. HealthPickup inherits MonoBehaviour from Pickup, so it can be dragged into a GameObject, and can use `transform` and `Update()`.

Don't assume a subclass has to be "bigger" than the base class. Sometimes you have a class that does more than what you need, and it's easy to inherit from it and add restrictions, sort of.

In this example, I've got a button that flips through options as you click it, like 0-4 or 0-15. I can re-use it for a simple 0/1 toggle:

```
class MultiButton {
  public int val, maxVal; // value goes from 0 to maxVal
  public override void click() { val++; if(val>maxVal) val=0;
}

class Toggle : MultiButton {
  public MultiButton() { val=0; maxVal=1; }
}
```

`Toggle` is really just a shortcut for making a MultiButton and setting maxVal to 1. But there's nothing wrong with good shortcuts.

Lots of `a1 as Cats` mean you might want virtual functions. The job of a virtual function is to automatically figure out which class you are and do the right thing. So if you can, let it make your job easier.

An example. This hand-checks the real type and decides what to do:

```
HealthPickup h = i1 as HealthPickup;
if(h!=null) gainHealth(h.healAmount);
AmmoPickup a = i1 as AmmoPickup;
if(a!=null) addAmmo(a.ammoType, a.bullets);
 ...
```

You could rewrite that as a virtual `applyPickup` function:

```
abstract class Pickup {
  public virtual void applyPickup(Player p) {}
  ...
}

class HealthPickup : Pickup {
```

```
public int healAmount;
public override void applyPickup(player p) { p.gainHealth(healAmount); }
...
```

Then you can call `i1.applyPickup(thePlayer)` and have it automatically check which type it was and do the right thing.

In theory, besides being shorter code, it's also nicer to have all the healthPack stuff in the HealthPack script where you can find it.

Be a little careful of inheritance fads. Inheritance used to be really hot, as in you got a promotion if you used it enough, even if you didn't need to. You can find inheritance stuff on-line that seems to just be about turning one class into three. Tthere's a good chance the goal really is just to have more classes.