Chapter 40

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.

40.1 A story about dynamic dispatch

This section doesn't have any rules or tricks. You may have noticed the rules for making `a1.cost()` work properly on a Cat or Dog are very odd. This section explains why. It if wasn't bugging you before, skipping this part is safe.

First, to review: if we call `a1.cost()`, and `a1` is pointing to a Cat, we should call the Cat `cost` function. For a second, it might seem as if it should always call the Animal version, because of `a1`, but we quickly realize that would give wrong answers.

So why does `C#` try so hard to make us do it the wrong way?

The two things to know are some languages are very interested in being just a tiny bit faster. And inheritance was one of the last computer tricks we invented.

Compilers like to do as much work as possible ahead of time. It turns out that if `a1.cost()` keys off how `a1` is an Animal, it can preload exactly where to jump for the call. In fact, if `cost()` is short, it might copy the code directly into your function (that's called in-lining.)

Looking up the real type is a lot more work, and you can never use inlining. The computer really has to look up the real type, look that up in a table, check for override's ... before it knows where to jump.

In an old-style language where this was just invented, like C++, we couldn't just change the commands around. We needed to make up keyword `virtual` to enable it, and we called this new way a virtual function.

We also liked it since speed is important to C++. The general rule is to make you work a little harder to do things that run slower.

In contrast, Java is a much newer language, emphasizing safe and easy over speed. They simplified things so covered-up functions always run the correct way. There aren't any extra keywords or terms, and you don't need to learn the difference.

The way `C#` does it is a neat story about how languages get designed. They decided that since `C#` is more of a general-purpose language, it should have the option for non-virtual functions. That might make some programs run a tiny bit faster.

They should have made virtual the normal way, and have you add extra words to run the bad way. But `C#` was new and they didn't want to scare people away. C++ was the best-known language with both options, so they copied the funny "bad way, unless you add the words to make it good" rules.

40.2 Privacy

If you recall, `private` is a way to help organize things – a 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 need more: when you inherit from a class, what can you see from it? We'll grow the privacy rules for that:

- `private` means only you. Things that inherit from you can't see your private stuff. As usual, they have them. But they can't see or directly use them.
- `public` is unchanged – everyone, anywhere can see it.
- A new keyword `protected` means "public if you inherit, private to everyone else." As usual, it can be used on variables or functions.

For an example of each, imagine Animas have a standard cost multiplier based on royalty. But not all animals use it all the time. Our plan is to give Animal a "helper" function computing it. Real animal sub-classes can use it as needed.

And then, we'll give our helper a helper function. Here's what they look like. Note the `protected` and `private` in front:

```
class Animal {
  // helper to compute the cost:
  protected float _getNameCostFactor() {
    if(_nobleCheck("Lord",true)) return 2.0f; // starts with
    if(_nobleCheck("Sir",true)) return 1.4f;
    if(_nobleCheck("Esq",false)) return 1.1f; // ends with
    return 1.0f
```

```
  }

  // helper for the function above:
  private bool _nobleCheck(string w, bool checkAtStart) {
    if(checkAtStart) return w.StartsWith(w));
    else return w.EndsWith(w));
  }
}
```

If we type `a1`-dot or `c1`-dot, neither of these will be in the pop-up. That's good. They only exist to help compute `cost()`, which is the function outsiders are suppose to call.

Inside of a `Goat` we can't see the second one, which is also good, since it's only a helper for the first. But we can use the first, in the normal way:

```
class Goat : Animal {
  public override cost() {
    float rm=1.0f;
    if(age>=4) rm=_getNameCostFactor(); // <= using helper from Animal
    // NOTE: the other one, _nobleCheck, wasn't in the pop-up!!
    ... // do more cost stuff
  }
}
```

As usual, if you're not sure, making everything `public` is safe. The only bad thing is your pop-ups will be more cluttered.

The interesting thing about `protected` is how we're thinking of ways to keep the program as smaller parts, where each part can only see what it needs to.

40.2.1 abstract base class

In the Animal, Cat, Dog example, I only made Cats and Dogs. I could have used `new Animal()`, but it wouldn't make any sense (a generic Animal?)

A base class you never plan to create is common. Adding `abstract` in front of the name makes it illegal to create one. It's another one of those rules that does nothing except limit us, and make the program easier to read:

```
abstract class Animal { ... } // new Animal() is now an error

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

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

The way the rule works is tricky. We can still declare `Animal a1;` which will point to a Cat or Dog. We can write functions with Animal inputs, which

for real always take Cats or Dogs. We can still have `List<Animal>`, full of Cats and Dogs.

The end result is if we use `new Animal()` by mistake, and that's always a mistake, we get a helpful error. It's also a nice note for someone reading the program.

Abstract base classes are common. It seems like you should always do it. But here's a slightly fake example when you wouldn't:

```
class Hammer { // not abstract. Since we could have normal hammers
  // add hammer stats
}

class ElectricHammer : Hammer {
  // add stats about plug-ins and batteries needed
}
```

In this case, we could have `Hammer h1;` which could point to an actual Hammer or an Electric one. A function like `poundNails(Hammer h)` could take either type.

40.3 Inheritance chains

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 it gets. In this example, FlyingCats has everything from Cats and Animals:

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

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

As usual, everything inside of you looks the same. A FlyingCat has name, age, cuteness and airSpeed, all looking like normal member vars. Likewise it has every function from Animal and Cat (except `private` ones.)

Everything else works the same. FlyingCat's can override functions from Cat. They can override functions in Animal that Cat didn't override. `Animal a1;` can point to a FlyingCat.

One new thing: `Cat c1;` can now also point to a FlyingCat. But we probably won't use that trick.

40.4 Interfaces

Suppose we wrote a base class that had only do-nothing functions. For example, this is for something that might break each time you use it:

```
public class Breakable {
  public virtual checkForUseBreak() {} // call after each use
  public virtual bool isBroken() { return true; } // dummy answer
}
```

Clearly, just this is useless. But suppose a few classes inherit from it and override those two functions. They all count as Breakables and can share Breakable-using functions. For example, `n=getActualUses(popper, 10)` checks whether you can use it 10 times, or if it breaks early:

```
int getActualUses(Breakable b, int timesWanted) {
  int useAmt=0;
  while(!b.isBroken() && useAmt<timesWanted) {
    b.checkForUseBreak();
    useAmt++;
  }
  return useAmt;
}
```

First, note that we could run this normal function with an actual Breakable input. It would always say 0, since the useless `isBroken()` function always says true. But it would run.

Next remember this is our old dynamic dispatch trick. The purpose of `getActualUses` is to run with subclasses of Breakable, using *their* 2 functions.

For example Hammer and Pen could be Breakables:

```
class Hammer : Breakable {
  // Hammers have a 2% chance to break each use:
  bool broke=false;
  public override checkForUseBreak() {
    if(Random.value<0.02f) broke=true;
  }
  public override isBroken() { return broke; }
  // rest of hammer stuff goes here
}

class Pen : Breakable {
  // Pens have ink for 20 uses:
  int usesLeft=20;
  public override checkForUseBreak() { usesLeft--; }
  public override isBroken() { return usesLeft<=0; }
  // rest of pen stuff goes here
}
```

Something functionally useless we inherit, which only "registers" some of our functions, is often called a *contract*. Bt inheriting you promise to write those functions, and in return you now count as a Breakable.

Since it's just a set of functions, we sometimes say you're inheriting another *interface*.

Some languages have an official way to write a a contract-only class. `C#`'s way is pretty typical. You use the word `interface` instead of class, you're not allowed to declare any variables or write bodies for any functions, but you don't have to write abstract, public or virtual:

```
interface Breakable { // counts as abstract (new Breakable() is an error)
  void checkForUseBreak(); // counts as public and virtual
  bool isBroken(); // sub-classes must write these
}
```

Now, when you write `class Hammer : Breakable` you really are promising to write those functions. You get an error if you don't. But otherwise it works the same way. You can declare Breakable variables, or write functions taking a Breakable, and it's understood they'll take sub-classes of Breakable.

Let's jump back a bit. What if we tossed all of this stuff? Say a function with input `q` used `q.age` and called `q.isAdult()`. How about we let you call it with any class that has those two things?

That works – some languages allow it (it's sometimes called *duck typing*, after the joke "it it walks and talks like a duck, it's a duck.") The drawback is it can be hard to know what things a function needs you to have. Using interfaces, a function taking a Breakable is really just saying "you need these two functions to run me."

Back to using interfaces, the main point is being able to have several of them. Suppose we have another interface for things that can move:

```
interface Movable {
  void setTarget(Vector3 pos);
  void move();
  // more move-related funtions
}
```

To make it more interesting, suppose we have a Machine base class, that works about the same way as Animal. We could have real classes that inherit from Animal or Machine, and also get Breakable and Movable interfaces:

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

Say we have `List<Movable> M;` of things wandering around the map. We'd be able to add Cars, Birds and Turtles. We could call `M[i].setTarget(v);` and `M[i].move();`.

if we really needed to we could check whether any of them break:

```
Breakable bb = M[i] as Breakable;
if(bb!=null) {
  if(bb.isBroken()) { ... } // crash? remove from list?
}
```

This stuff takes practice. It's confusing since a bunch of things blur together. Base classes and interfaces are different ideas, but we inherit from both and use variables the same way for both. The "sub-class counts as" rule is the same for both.

It's one of those things that isn't useful until you have a big, messy program and you're thinking "these 3 classes are similar, but different. I wish I had a way to tell the computer how they all have this certain part in common."

40.5 Built-in interfaces

`C#` has a built in interface for sorting that might help explain the idea.

You might remember that we can run a built-in sort by passing in a 2-item compare function. This is a different one.

The interface requires you to write one compare member function:

```
interface IComparable<T> {
  int CompareTo(T t); // Ex: n = a1.CompareTo(a2);
}
```

If you inherit from this, you promise to provide a `CompareTo` function. Here's the Cat class using it:

```
class Cat : Animal, IComparable<Cat> { // <= added IComparable
  // the required function:
  int IComparable<Cat>.CompareTo(Cat c2) { // yikes!
    // sort by age, using standard -1/0/+1 rules:
    if(age<c2.age) return -1;
    if(age==c2.age) return 0;
    return +1;
  }
  // rest of regular Cat stuff here
}
```

The important thing about this is we now officially have `c1.CompareTo(c2);` that will compare two Cats. We count as an IComparable.

The other built-in sort function uses it. It looks like `Sort(IComparable<T>[] A)`. That looks horrible, but means an array of any class with a `CompareTo` function. Which our Cats now have.

`System.Array.Sort(Cats);` now sorts cats by age.

Some notes:

- I left out lots of specific C# details. For example figuring out `IComparable<Cat>.CompareTo` was the way it wanted me to write it. Ouch. But if you understand the basic idea, you can look up those details.
- To repeat, the Sort that takes a second compare-function input is completely different. It can sort any way you want, but you always have to tell it how.
- This way is nice if there's one main way you'd want to sort. Writing it is a pain, but then anyone can use `Sort(Cats)` without knowing anything about how it works.
- This is also an example of overloading magic. `Sort(Cats)` uses the interface method, while `Sort(Cats, catNameLongestFirstComp)` uses the completely different function-pointer method. But it feels like one Sort function, with options.

40.6 Misc examples

40.6.1 `Component` class

Unity3D uses a semi-typical system for making things: everything is a GameObject, and you add sub-parts to make it act the way you want.

The sub-parts are all different, but we want to put them all in one list. To do that, we use the base-class trick. Everything inherits from `Component`, and gameObject's have one List of Component's.

Component holds what everything has in common, which seems like it would be nothing. But everything in the list needs a link to what it's in. So:

```
class Component { // lots of things inherit from this
  public GameObject gameObject; // link to what's it's on
  public Transform transform; // ditto
}
```

This is actually the way our scripts have those two variables (more, later).

If you remember, `Rigidbody` lets something move by itself. It inherits from `Component`:

```
class Rigidbody : Component { // <- inherits
  public Vector3 velocity;
   ... // lots more
}
```

Nothing special here. Rigidbodies get 2 extra variables for free and count as Components.

The same way a List of Animals can hold Cats and Dogs, a GameObject's `List<Component>` can hold Rigidbodies.

Here's a function to find the first Rigidbody if there is one, in a List of Components:

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

40.6.2 Scripts and inheritance

The first line of a script looks like `class testScriptA : Monobehaviour {`. All of our scripts inherit from that class. But `Monobehaviour` inherits from `Behaviour` and that inherit's from `Component`:

```
yourScript -> Monobehaviour -> Behaviour -> Component
```

That's just simple chain inheritance. It means all scripts are Components. They can go into the list.

The `Behaviour` subclass only has `bool enabled;` in it. That's not much, but it's a real example of how just one thing can be fine – don't be stingy with subclasses if you need them.

Other things beside scripts can be disabled, so they also inherit from `Behaviour`. We could do this to skip disabled scripts, lights and so on:

```
// handle all Components
for(int i=0; i<C.Count; i++) {
  // check for a disabled behaviour and skip:
  Behaviour bb = C[i] as Behaviour;
  if(bb!=null && bb.enabled==false) continue;
  // do stuff with C[i]:
}
```

The class Monobehaviour is mostly a tag to let you know it's a script. It doesn't have anything really useful in it, but we can use `C[i] as Monobehaviour` to check for scripts.

So you know, `Mono` is the name of the framework running scripts.

40.6.3 Image subtree example

Unity has two classes to put a 2D picture in the screen: `RawImage` is basic, and `Image` has options to stretch parts of it.

We'd like to make these as interchangable as possible, and they both have a color and a material (what picture they use.) So Unity sets them up with a common base class (which eventually inherits from Component):

```
                   Image (extra stats about resizing percents)
                  /
MaskableGraphic --
   - Color         \
   - Material       RawImage (basic picture)
```

When your scripts needs a link to any picture, do this:

```
public MaskableGraphic catPicture; // really an Image or RawImage

  catPicture.color=Color.White; // legal for both types

  // Using base class trick to work on either:
  void turnRed(MaskableGraphic mg, Color cc) { mg.Color=cc; }
```

We can easily swap in either type, and change the color and picture. We can't easily change the resizing values from the more complex Image type, but we usually don't want to.

40.7 General inheritance advice

One of the things people tend to do is, for no reason, is make subclasses that look like a classification system from school. For example:

```
class Tool { ... }
class HandleTool : Tool { ... }
class SqueezeTool : Tool { ... }
class Wrench : SqueezeTool { ... }
class Hammer : HandleTool { ... }
```

```
                / Hammer
    / HandleTool - Screwdriver
Tool
    \ SqueezeTool - Wrench
                \ Scissors
```

But will we ever want `Tool t1;` that can point to any tool, and nothing else? It seems more likely we'll want a variable like `HandItem rightHand;` that

can point to any weapon or tool. Or we'll want `List<InvetoryItem>` that can hold every tool, potion, spellbook ....

Will we ever want to call `t1.apply(target)` and have to work for tools, but nothing else? It might make more sense to have `apply()` work for everything, so you can try to apply a potion, or some food (or applying a weapon would try to smash it?)

And, the same way, what use is `SqueezeTool`? Do we need `SqueezeTool s1;` that can only point to a SqueezeTool? Will we ever write a function that only takes a `SqueezeTool` as input?

And then, are there really any common stats that all tools have? They probably have a weight, price and size, but every item in the game has that.

As a counter-example, it might be useful to make a base Pickup class that types of Pickups inherit from:

```
class Pickup {
  public float timeout; // how long until it vanishes
  Vector3 location;

  // everyone will override this:
  virtual public bool use() { return false; }
}

class HealthPickup : Pickup {
  public int healthGain;

  // try to give health to player:
  public override bool use() {
    int needed=100-player.health;
    if(healthGain>needed) // apply some, leave the rest
     ...
  }
}
```

We can use the `List<Pickup> P;` trick to hold all types of Pickups. When we grab one we can use virtual functions to make `P[i].use()` do whatever that type of pickup does.

For many simple things, we don't need inheritance at all – variables are fine. For example, what if we have types of animals, but otherwise they all act the same:

```
class Animal {
  public float tailLen; // -99 means no tail
  public float cuteness; // -99 means no cuteness
```

```
  public enum aType {cat, dog, rabbit, snake}
  public aType theType;
}
```

Maybe you can see how this is pushing against the edge. Each animal has a few useless stats (spiders always have tail=-99). Subclasses let us add only what we need. Maybe we need a few special `if`'s for certain animals – maybe lizard tails grow as they age.

Inheritance was actually invented this way. Things like this slowly grow more complicated as you add animals: you get more special-use variables and more `if`'s for one animal type. Sub-classes are a nice way to split those out.

Sometimes you have lots of sub-parts that various things will need in combinations. Inheritance is terrible at that. But that's fine. You can make classes for the sub-parts and use them the normal way:

```
class ElectricHammer {
  public InventoryItem itm; // cost, weight ...
  public UseRestrictions; // what level you have to be to use it
  public PowerSupply ps; // info on batteries and plug-ins
  public PowerSupply ps_backup; // nil = no back-up

  public float secsToPoundNail; // <- finally stats about this hammer
}
```

Notice how there are 2 possible power supplies. Inheritance can't even do that.

Later, you might want these in an inventory list, and want a link to things you hold in your hands. So `ElectricHammer` might inherit from a class you made `HandItem` which inherits from `InventoryItem`. But the other parts would still be just regular member variables.

Another common bit of inheritance advice is how using `as` a lot often means you should write a virtual function. From the Pickup example, our first try might look like this:

```
HealthPickup hlt = p1 as HealthPickup;
if(hlt!=null) hlt.addHealth();
AmmoPickup amo = p1 as AmmoPickup;
if(amo!=null) amo.AddAmmo();
```

This is exactly what having `use()` in the base Pickup class, and then overriding it, is for. Then those lines become simply `p1.use()`.