

Object Oriented Programming

Dr Robert Harle

IA NST CS and CST
Lent 2009/10
Handout 1

OO Programming

- This is a new course this year that absorbs what was “Programming Methods” and provides a more formal look at Object Oriented programming with an emphasis on Java
- Four Parts
 - Computer Fundamentals
 - Object-Oriented Concepts
 - The Java Platform
 - Design Patterns and OOP design examples

If we teach Java in isolation, there's a good chance that students don't manage to mentally separate the object-oriented concepts from

Java's implementation of them. Understanding the underlying principles of OOP allows you to transition quickly to a new OOP language. Because Java is the chosen teaching language here, the vast majority of what I do will be in Java, but with the occasional other language thrown in to make a point.

Actually learning to program is best done practically. That's why you have your practicals. I don't want to concentrate on the minutiae of programming, but rather the larger principles. It might be that some of this stuff will make more sense when you've done your practicals; I hope that this material will help set you up to complete your practicals quickly and efficiently. But we'll see.

Java Ticks

- This course is meant to *complement* your practicals in Java
 - Some material appears only here
 - Some material appears only in the practicals
 - Some material appears in both: deliberately!
- A total of 7 workbooks to work through
 - *Everyone should attend every week*
 - CST: Collect 7 ticks
 - NST: Collect at least 5 ticks

Books and Resources

- OOP Concepts
 - Look for books for those learning to first program in an OOP language (Java, C++, Python)
 - *Java: How to Program* by Deitel & Deitel (also C++)
 - *Thinking in Java* by Eckels
 - *Java in a Nutshell* (O' Reilly) if you already know another OOP language
 - Lots of good resources on the web
- Design Patterns
 - *Design Patterns* by Gamma et al.
 - Lots of good resources on the web

Books and Resources

- Also check the course web page
 - Updated notes (with annotations where possible)
 - Code from the lectures
 - Sample tripos questions

<http://www.cl.cam.ac.uk/teaching/0910/OOProg/>

There really is no shortage of books and websites describing the basics of object oriented programming. The concepts themselves

are quite abstract, but most texts will use a specific language to demonstrate them. The books I've given favour Java (because that's the primary language you learn this term). You shouldn't see that as a dis-recommendation for other books. In terms of websites, SUN produce a series of tutorials for Java, which cover OOP:

<http://java.sun.com/docs/books/tutorial/>

but you'll find lots of other good resources if you search. And don't forget your practical workbooks, which will aim *not* to assume anything from these lectures.

Chapter 1

Computer Fundamentals

What can Computers Do?

- The computability problem
 - Given infinite computing 'power' what can we do? How do we do it? What can't we do?
 - Option 1: Forget any notion of a physical machine and do it all in maths
 - Leads to an abstract mathematical programming approach that uses functions
 - Gets us Functional Programming (e.g. ML)
 - Option 2: Build a computer and extrapolate what it can do from how it works
 - Not so abstract. Now the programming language links closely to the hardware
 - This leads naturally to imperative programming (and on to object-oriented)



What can Computers Do?

- The computability problem
 - Both very different (and valid) approaches to understanding computer and computers
 - Turns out that they are equivalent
 - Useful for the functional programmers since if it didn't, you couldn't put functional programs on real machines...

WWII spurred an interest in machinery that could compute. During the war, this interest was stoked by a need to break codes, but also to compute relatively mundane quantities such as the trajectory of an artillery shell. After the war, interest continued in the abstract notion of 'computability'. Brilliant minds (Alan Turing, etc) began to wonder what was and wasn't 'computable'. They defined this in an abstract way: given an infinite computing power (whatever that is), could they solve anything? Are there things that can't be solved by machines?

Roughly two approaches to answering these questions appeared:

Maths, maths, maths. Ignore the mechanics of any real machine and imagine a hypothetical machine that had infinite computation power, then write some sort of 'programming language' for it. This language turned out to be a form of mathematics known as 'Lambda calculus'. It was based around the notion of *functions* in a mathematical sense.

Build it and they will come. Understand how to build and use a real computing machine. This resulted in the von Neumann architecture and the notion of a Turing machine (basically what we would call a computer).

It turned out that both approaches were useful for answering the fundamental questions. In fact, they can be proven to be the same thing now!

Imperative

- This term you transition from functional (ML) to imperative (Java)
- Most people find imperative more natural, but each has its own strengths and weaknesses
- Because imperative is a bit closer to the hardware, it does help to have a good understanding of the basics of computers.
 - All the CST Students have this
 - All the NST students don't... yet.

All this is very nice, but why do you care? Well, there's a legacy from these approaches, in the form of two *programming paradigms*:

Functional Languages. These are very mathematically oriented and have the notion of functions as the building blocks of computation.

Imperative or Procedural Languages. These have variables (state) and procedures as the main building blocks¹. Such languages are well known—e.g. C—and include what we call *object-oriented* languages such as C++ and Java.

Note that I have pluralised “Language” in the above sentences. Terms like “Object-oriented” are really a set of ideas and concepts that various languages implement to varying degrees.

Last term you toured around Computer Science (in FoCS) and used a particular language to do it (ML). Although predominantly a functional programming language, ML has acquired a few imperative ‘features’ so you shouldn’t consider it as a pure functional language. (For example, the ML reference types you looked at are not functional!).

This term you will shift attention to an *object-oriented* language in the form of Java. What we will be doing in this course is looking at the paradigm of object-oriented programming itself so you can better understand the underlying ideas and separate the Java from the paradigm.

1.1 Hardware Fundamentals

As I said before, the imperative style of programming maps quite easily to the underlying computer hardware. A good understanding of how computers work can greatly improve your programming capabilities with an imperative language. Thing is, only around a half of you have been taught the fundamentals of a computer (the

¹A *procedure* is a function-like chunk of code that takes inputs and gives outputs. However, unlike a formal function it can have side effects i.e. can make non-local changes to something other than its inputs and declared outputs.

CST half). Those people will undoubtedly be rather bored by what follows in the next few pages, but it's necessary so that everyone is at the same point when we start to delve deeper into object-oriented programming.

Computers do lots of very simple things very fast. Over the years we have found optimisation after optimisation to make the simple processes that little bit quicker, but really the fundamentals involve some memory to store information and a CPU to perform simple actions on small chunks of it.

We use lots of different types of memory, but conceptually only two are of interest here. System memory is a very large pool of memory (the 2GB or so you get when you buy a machine). Then there are some really fast, but small chunks of memory called registers. These are built into the CPU itself.

The CPU acts only on the chunks in the registers so the computer is constantly *loading* chunks of data from system memory into registers and operating on the register values.

The example that follows loads in two numbers and adds them. Those doing the Operating Systems course will realise this is just the *fetch-execute* cycle. Basically, there is a special register called the program counter (marked P) that tells the computer where to look to get its next instruction. Here I've made up some operations:

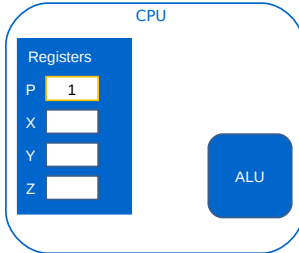
LAM. LOAD the value in memory slot A into register M

AMNO. ADD the values in registers M and N and put the result in register O.

SMA. STORE the value in register M into memory slot A

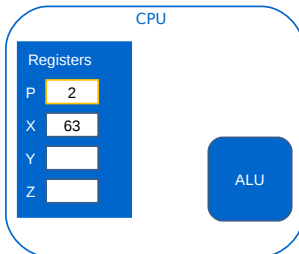
Dumb Model of a Computer

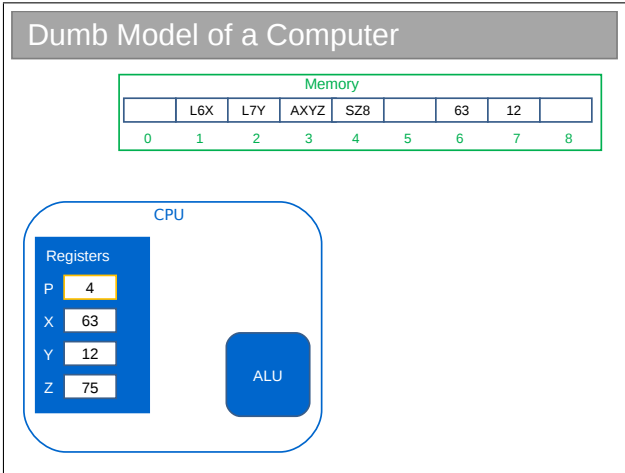
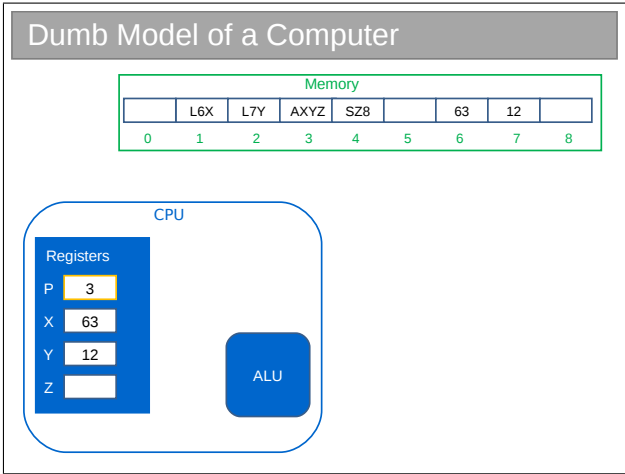
Memory								
	L6X	L7Y	AXYZ	SZ8		63	12	
0	1	2	3	4	5	6	7	8



Dumb Model of a Computer

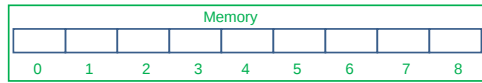
Memory								
	L6X	L7Y	AXYZ	SZ8		63	12	
0	1	2	3	4	5	6	7	8





All a computer does is execute instruction after instruction. Never forget this when programming!

Memory (RAM)



- You probably noticed we view memory as a series of slots
 - Each slot has a set size (1 byte or 8 bits)
 - Each slot has a unique *address*
- Each address is a set length of n bits
 - Mostly $n=32$ or $n=64$ in today's world
 - Because of this there is obviously a maximum number of addresses available for any given system, which means a maximum amount of installable memory

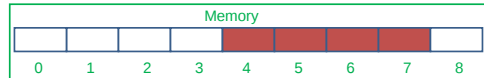
We slice system memory up into 8-bit (1 byte) chunks and give each one an *address* (i.e. a slot number).

Each of the registers is a small, fixed size. On today's machines, they are usually just 32 or 64 bits. Since we have to be able to squeeze a memory address into a single register, the size of the registers dictates how much of the system memory we can use.

Q1. How much system memory can we use if we have 8, 32 or 64 bit registers?

Big Numbers

- So what happens if we can't fit the data into 8 bits e.g. the number 512?
- We end up distributing the data across (consecutive) slots



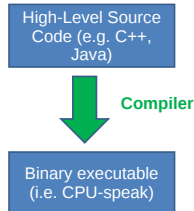
- Now, if we want to act on the number as a whole, we have to process each slot individually and then combine the result
- Perfectly possible, but who wants to do that every time you need an operation?

If I have two lots of 128-bits of data, how do I add them together on a 32 bit machine? I can't squeeze 128 bits into 32 bits. I have to chop the 128 bits up into 32 bit chunks, add corresponding chunks together and then put those new chunks back together again to get a 128-bit answer.

Can you imagine having to do this every time you needed to do an addition? Nasty.

High Level Languages

- Instead we write in high-level languages that are human readable (well, compsci-readable anyway)



You already know the solution from FoCS and ML. We build up layers of abstractions and write tools to drill down from one layer to the next.

We program in high level code and leave it to the *compiler* to figure out that when we say

```
c=a+b;
```

we actually mean “chop up a and b, load them into registers, add them, look out for overflow, then put all the pieces back together in memory somewhere called c”.

1.2 Functional and Imperative Revisited

We said imperative programming developed from consideration of the hardware. Given the hardware we've just designed, the logical thing to do is to represent data using *state*. i.e. explicit chunks of memory used to store data values. A program can then be viewed as running a series of commands that alter ('*mutate*') that state (hopefully in a known way) to give the final result.

So $c=a+b$ is referring to three different chunks of memory that represent numbers. Unlike in ML, it is perfectly valid to say $a=a+b$, which computes the sum and then overwrites the state stored in a .

Functional programs, on the other hand, can be viewed as independent of the machine. Strictly speaking, this means they can't have state in the same way since there's simply nowhere to put it (no notion of memory). Instead of starting with some data that gets *mutated* to give a result, we only have an input and an output for each function.

Of course, ML runs on real machines and therefore makes use of real hardware components such as memory. But that's because someone has mapped the ideal mathematical processes in functional code to machine operations (this must be possible because I said both the mathematical view and the system view are equivalent). The key point is that, although functional code runs on real hardware, it is independent of it since it is really a mathematical construct.

Most people find the imperative approach easier to understand and, in part, that is why it has gained such a stronghold within modern programming languages.

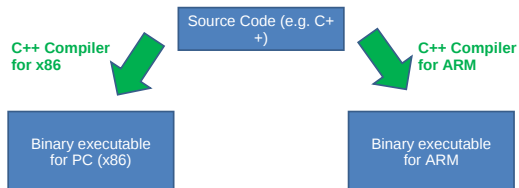
1.3 Machine Architectures

Machine Architectures

- Actually, there's no reason for e.g ARM and Intel to use the same instructions for anything - and they don't!
- The result? Any set of instructions for one processor only works on another processor if they happen to use the same instruction set...
 - i.e. The binary executable produced by the compiler is CPU-specific
- We say that each set of processors that support a given set of instructions is a different *architecture*
 - E.g. *x86, MIPS, SPARC, etc.*
- But what if you want to run on different architectures??

Compilation

- So what do we have? We need to write code specifically for each family of processors... Aarrgh!
- The 'solution':



Remember those weird commands I made up when we did the fetch-execute cycle? Well, not all CPUs are equivalent: newer ones have cool new instructions to do things better. All that means is that a set of instructions for CPU X won't mean anything to CPU Y unless the manufacturers have agreed they will.

Q2. Why do you think CPU manufacturers haven't all agreed on a single set of instructions?

There is one very popular set of machine instructions known as x86. Basically, these were what IBM came up with for the original PC. Now the term 'PC' has come to mean a machine that supports at least the set of x86 instructions. Manufacturers like Apple traditionally chose not to do so, meaning that Mac applications couldn't run on PC hardware and vice-versa.

Q3. Apple recently started using Intel processors that support x86 instructions. This means Apple machines can now run Microsoft Windows. However, off-the-shelf PC software (which is compiled for x86) does not run on a Mac that is using the Apple operating system compiled for the Intel processor. Why not?

Enter Java

- Sun Microcomputers came up with a different solution
 - They conceived of a Virtual Machine - a sort of idealised computer.
 - You compile Java source code into a set of instructions for this Virtual Machine ("bytecode")
 - Your real computer runs a program (the "Virtual machine" or VM) that can efficiently translate from bytecode to local machine code.
- Java is also a *Platform*
 - So, for example, creating a window is the same on any platform
 - The VM makes sure that a Java window looks the same on a Windows machine as a Linux machine.
- Sun sells this as "Write Once, Run Anywhere"



Java compiles high-level source code not into CPU-specific instructions but into a generic CPU instruction set called **bytecode**. Initially there was no machine capable of directly using bytecode, but recently they have started to appear in niche areas.

Therefore you usually run bytecode programs inside the **virtual machine** (which has been compiled specifically for your architecture). This converts bytecode instructions into meaningful instructions for your local CPU.

All the CPU-specific code goes into the virtual machine, which you should just think of as a piece of intermediary software that translates commands so the local hardware can understand.

SUN publishes the specification of a Java Virtual Machine (JVM) and anyone can write one, so there are a plenty available if you want to explore. Start here:

<http://java.sun.com/docs/books/jvms/>

1.4 Types

Types and Variables

- We write code like:

```
int x = 512;  
int y = 200;  
int z = x+y;
```

- The high-level language has a series of *primitive* (built-in) types that we use to signify what's in the memory
 - The compiler then knows what to do with them
 - E.g. An "int" is a primitive type in C, C++, Java and many languages. It's usually a 32-bit signed integer
- A variable is a name used in the code to refer to a specific instance of a type
 - x,y,z are variables above
 - They are all of type int

Primitive Types in Java

- "Primitive" types are the built in ones.
 - They are building blocks for more complicated types that we will be looking at soon.
- boolean - 1 bit (true, false)
- char - 16 bits
- byte - 8 bits as a signed integer (-128 to 127)
- short - 16 as a signed integer
- int - 32 bits as a signed integer
- long - 64 bits as a signed integer
- float - 32 bits as a floating point number
- double - 64 bits as a floating point number

[See practicals]

You met the notion of types in FoCS and within ML. They're important because they allow the compiler to keep track of what the data in its memory actually means and stop us from doing dumb things like interpreting a floating point number as an integer.

In ML, you know you can specify the types of variables. But you almost never do: you usually let the compiler infer the type unless there's some ambiguity (e.g. between `int` and `double`). In an imperative language such as Java you *must* explicitly assign the types to the variable.

For any C/C++ programmers out there: yes, Java looks a lot like the C syntax. But watch out for the obvious gotcha — a `char` in C is a byte (an ASCII character), whilst in Java it is two bytes (a Unicode character). If you have an 8-bit number in Java you may want to use a `byte`, but you also need to be aware that a `byte` is *signed*..!

You do lots more work with number representation and primitives in your Java practical course. You do a lot more on floats and doubles in your Floating Point course.

Reference Types (Classes)

- Lets imagine you're creating a program that stores info about someone doing CS
- You might create a set of variables (instances of types) as follows:

```
String forename = "Kay";  
String surname = "Oss";
```

- Now you need to represent two people so you add:

```
String forename2 = "Don";  
String surname2 = "Keigh";
```

- Now you need to add more - this is getting rather messy
- Better if we could create our own type - call it a "Person"
 - Let it contain two values: forename and surname

Reference Types (Classes)

```
public class Person {  
    String forename;  
    String surname;  
}
```

- In Java we create a **class** which acts as a blueprint for a custom type
- A class:
 - Has attributes (things it is assigned e.g. name, age)
 - Has methods (things it can do e.g. walk, think)
- When we create an **instance** of a class we:
 - Assign memory to hold the attributes
 - Assign the attributes
- We call the instance an **object**
 - As in object-oriented programming

You've met things called 'reference types' in ML so hopefully the idea of a reference isn't completely new to you. But if you're wondering

why classes are called reference types in Java, we'll come to that soon.

Definitions

- **Class**
 - A class is a grouping of conceptually-related attributes and methods

- **Object**
 - An object is a specific instance of a class

Once we have compiled our Java source code, we end up with a set of .class files. We can then distribute these files without their source code (.java) counterparts.

In addition to javac you will also find a javap program which allows you to poke inside a class file. For example, you can disassemble a class file to see the raw bytecode using javap -c classfile:

Input:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
}
```

javap output:

```
Compiled from "HelloWorld.java"
```

```
public class HelloWorld extends java.lang.Object{  
    public HelloWorld();
```

```
    Code:
```

```
        0: aload_0
```

```
        1: invokespecial #1; //Method java/lang/Object."<init>":()V
```

```
        4: return
```

```
public static void main(java.lang.String[]);
```

```
    Code:
```

```
        0: getstatic #2; //Field java/lang/System.out:
```

```
            //Ljava/io/PrintStream;
```

```
        3: ldc #3; //String Hello World
```

```
        5: invokevirtual #4; //Method java/io/PrintStream.println:
```

```
            //(Ljava/lang/String;)V
```

```
        8: return
```

```
}
```

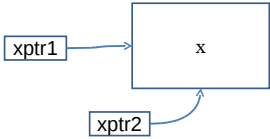
This probably won't make a lot of sense to you right now: that's OK. Just be aware that we can view the bytecode and that sometimes this can be a useful way to figure out *exactly* what the JVM will do with a bit of code. You aren't expected to know bytecode.

1.5 Pointers and References

Pointers

- In some languages we have variables that hold memory addresses.
- These are called *pointers*

```
MyType x;  
MyType *xptr1 = &x;  
MyType *xptr2 = xptr1;  
  
C++
```



- A pointer is just the memory address of the first memory slot used by the object
- The pointer type tells the compiler how many slots the whole object uses

A pointer is just the name given to memory addresses when we handle them in code. They are very powerful concepts to work with, but you need to take a lot of care.

Java doesn't expose pointers to the programmer, so we'll use some basic C examples to make some points in the lectures. There are just a few bits of C you may need to know to follow:

- `int x` declares a variable `x` that is a 32-bit signed integer
- `int *p` declares a pointer `p` that can point to an `int` in memory. It must be manually initialised to point somewhere useful.
- `p=&x` Gets the memory address for the start of variable `x` and stick it in `p` so `p` is a pointer to `x`.
- `int a=*p` declares a new `int` variable called `a` and sets its value to that pointed to by pointer `p`. We say that we are *dereferencing*

p.

If it helps, we can map these operations to ML something like²:

C	ML	Comments
int x	let val xm=0 in...	xm is immutable
int *p	let val pm=ref 0 in...	pm is a reference to a variable initially set to zero
p=&x	val pm = ref xm	
int a=*p	val am = !pm	

The key point to take away is that a pointer is just a number that maps to a particular byte in memory. You can set that number to anything you like, which means there's no guarantee that it points to anything sensible..! And the computer just obeys — if you say that a pointer points to an int, then it doesn't argue.

Working with pointers helps us to avoid needless copies of data and usually speeds up our algorithms (see next term's Algorithms I course). But a significant proportion of program crashes are caused by trying to read or write memory that hasn't been allocated or initialised...

²Now, if all that reference type stuff in ML confused you last term, don't worry about making these mappings just now. Just be aware that what will come naturally in imperative programming has been retrofitted to ML, where it seems a little less obvious! For now, just concentrate on understanding pointers and references from scratch.

Java's Solution?

- You **can't get at the memory directly**.
 - So no pointers, no jumping around in memory, no problem.
 - Great for teaching. ☺
- It does, however, have **references**
 - These are like pointers except they are guaranteed to point to either an object in memory or "null".
 - So if the reference isn't null, it is valid
- In fact, all objects are accessed through references in Java
 - Variables are either primitive types or references!

Remember that classes are called *Reference Types* in Java. This is because they are *only* handled by reference. Code like:

```
MyClass m = new MyClass();
```

creates an instance of `MyClass` somewhere in main memory (that's what goes on on the right of the equals). And then it creates a *reference* to that instance and stores the result in a variable `m`.

A reference is really just a special kind of pointer. Remember that a pointer is just a number (representing a memory slot). We can set a pointer to any number we like and so it is possible that dereferencing it gets us garbage.

A reference, however, *always* has a valid address *or* possibly a special value to signify it is invalid. If you dereference a reference without the special value in it, the memory you read *will* contain something sensible.

In Java, the ‘special value’ is signified using the keyword `null`:

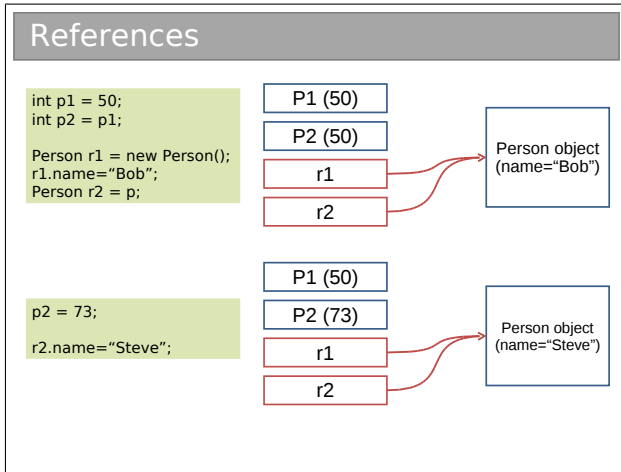
```
MyClass m = null;    // Not initialised
...
m = new MyClass();
```

If you try to do something with a reference set to `null`, Java will bail out on you. To test whether a reference is `null`:³

```
if (m==null) {
    ...
}
```

Q4. Pointers are problematic because they might not point to anything useful. A `null` reference doesn’t point to anything useful. So what is the advantage of using references over pointers?

³Note that the Java/C syntax uses `=` for assignment and `==` for comparison. We’ll discuss this in more detail later.



So we have *primitive* types and *reference* types. A variable for a primitive type can be thought of as being an instance of a type, so each you assign something to it, the same bit of memory gets changed as with P1 and P2 in the above slide.

A variable for a reference type holds the memory address. If you assign something to it, what it used to be assigned to still exists in memory (but your variable doesn't link to it any more). Multiple variables can reference the same object in memory (like `r1` and `r2` above).

Q5. Draw box diagrams like the one above to illustrate what happens in each step of the following Java code in memory:

```
Person p = null;  
Person p2 = new Person();  
p = p2;  
p2 = new Person();  
p=null;
```

Pass By Value and By Reference

```
void myfunction(int x, Person p) {  
    x=x+1;  
    p.name="Alice";  
}  
  
void static main(String[] arguments) {  
    int num=70;  
    Person person = new Person();  
    person.name="Bob";  
  
    myfunction(num, p);  
    System.out.println(num+" "+person.name)  
}
```

A. "70 Bob"
B. "70 Alice"
C. "71 Bob"
D. "71 Alice"

Now things are getting a bit more involved. We have added a function *prototype* into the mix: `void myfunction(int x, Person p)`. This means it returns nothing (codenamed `void`) but takes an `int` and a `Person` object as input.

When the function is run, the computer takes a copy of the inputs and creates local variables `x` and `p` to hold the copies.

Here's the thing: if you copy a primitive type, the value gets copied across (so if we were watching memory we would see two integers in memory, the second with the same value as the first).

But for the reference type, it is the *reference* that gets copied and not the object it points to (i.e. we just create a new reference `p` and point it at the same object).

When we copy a primitive value we say that it is *pass by value*. When we send a reference to the object rather than the object, it is *pass by reference*.

In my opinion, this all gets a bit confusing in Java because you have no choice but to use references (unlike in, say C++). So I prefer to think of all variables as representing either primitives or references (rather than primitives and actual objects), and when we pass any variable around its value gets copied. The 'value' of a primitive is its actual value, whilst the 'value' of a reference is a memory address. You may or may not like this way of thinking: that's fine. Just make *certain* you understand the different treatment of primitives and reference types.

Chapter 2

Object-Oriented Concepts

Modularity

- A class is a custom type
- We could just shove all our data into a class
- The real power of OOP is when a class corresponds to a concept
 - E.g. a class might represent a car, or a person
- Note that there might be sub-concepts here
 - A car has wheels: a wheel is a concept that we might want to embody in a separate class itself
- The basic idea is to figure out which concepts are useful, build and test a class for each one and then put them all together to make a program
 - The really nice thing is that, if we've done a good job, we can easily re-use the classes we have specified again in other projects that have the same concepts.
 - “Modularity”

Modularity is extremely important in OOP. It's the usual CS trick: break big problems down into chunks and solve each chunk. In this case, we have large programs, meaning scope for lots of coding bugs. We split the program into modules. The goal is:

- Modules are conceptually easier to handle
- Different people can work on different modules simultaneously
- Modules may be re-used in other software projects
- Modules can be individually tested as we go (unit testing)

The module 'unit' in OOP is a class.

State and Behaviour

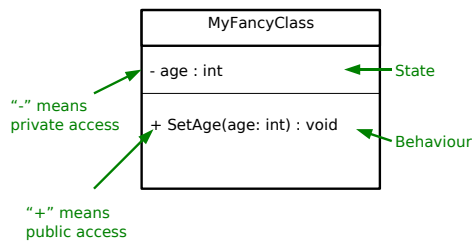
- An object/class has:
 - **State**
 - Properties that describe that specific instance
 - E.g. colour, maximum speed, value
 - **Behaviour/Functionality**
 - Things that it can do
 - These often *mutate* the state
 - E.g. accelerate, brake, turn

Identifying Classes

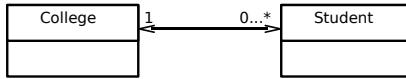
- We want our class to be a **grouping of conceptually-related state and behaviour**
- One popular way to group is using English grammar
 - **Noun → Object**
 - **Verb → Method**

“The footballer kicked the ball”

Representing a Class Graphically (UML)



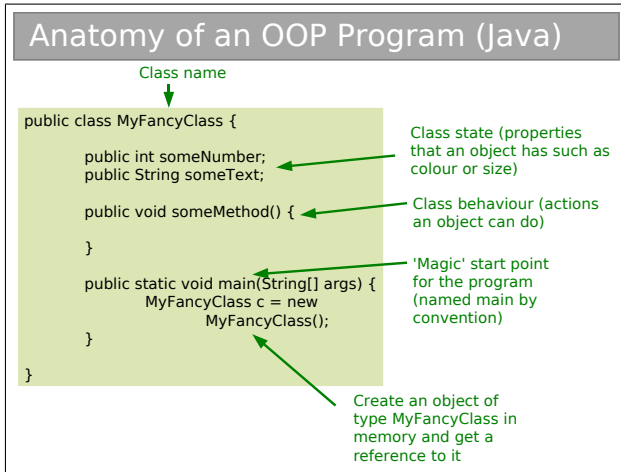
The “has-a” Association



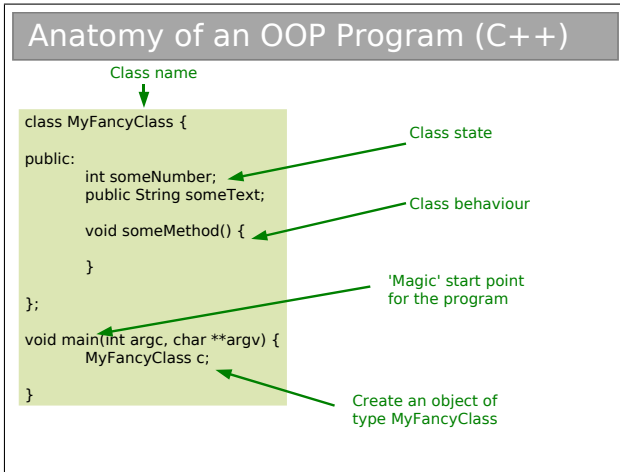
- Arrow going left to right says “a College has zero or more students”
- Arrow going right to left says “a Student has exactly 1 College”
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable that references a College object.
- Note that we are only linking *classes*: we don't start drawing arrows to primitive types.

The graphical notation used here is part of UML (Unified Modeling Language). UML is basically a standardised set of diagrams that can be used to describe software independently of any programming language used to create it.

UML contains many different diagrams (touched on in the Software Design course). Here we just use the *UML class diagram* such as the one in the slide.



There are a couple of interesting things to note for later discussion. Firstly, the word **public** is used liberally. Secondly, the **main** function is declared *inside* the class itself and as **static**. Finally there is the notation **String[]** which represents an array of String objects in Java. You will see arrays in the practicals.



This is here just so you can compare. The Java syntax is based on C/C++ so it's no surprise that there's a lot of similarities. This certainly eases the transition from Java to C++ (or vice-versa), but there are a lot of pitfalls to bear in mind (mostly related to memory management).

Let's Build up Java Ourselves

- We'll start with a simple language that looks like Java and evolve it towards real Java
- Use the same primitives and Java and the similar syntax. E.g.

```
class MyFancyClass {  
    int someNumber;  
    String someText;  
  
    void someMethod() {  
    }  
}  
  
void main() {  
    MyFancyClass c = new  
        MyFancyClass();  
}
```

A red box on the code box means it is not valid Java.

2.1 Encapsulation

Encapsulation

```
class Student {
    int age;
}

void main() {
    Student s = new Student();
    s.age = 21;

    Student s2 = new Student();
    s2.age=-1;

    Student s3 = new Student();
    s3.age=10055;
}
```

- Here we create 3 Student objects when our program runs
- Problem is obvious: nothing stops us (*or anyone using our Student class*) from putting in garbage as the age
- Let's add an *access modifier* that means nothing outside the class can change the age

Encapsulation

```
class Student {
    private int age;

    boolean SetAge(int a) {
        if (a>=0 && a<130) {
            age=a;
            return true;
        }
        return false;
    }

    int GetAge() {return age;}
}

void main() {
    Student s = new Student();
    s.SetAge(21);
}
```

- Now nothing outside the class can access the *age* variable directly
- Have to add a new method to the class that allows *age* to be set (but only if it is a sensible value)
- Also needed a GetAge() method so external objects can find out the age.

Encapsulation

- We hid the state implementation to the outside world (no one can tell we store the age as an int without seeing the code), but provided mutator methods to... errr, mutate the state
- This is *data encapsulation*
 - We define interfaces to our objects without committing long term to a particular implementation
- *Advantages*
 - We can change the internal implementation whenever we like so long as we don't change the interface other than to add to it (E.g. we could decide to store the age as a float and add GetAgeFloat())
 - Encourages us to write clean interfaces for things to interact with our objects

Another name for encapsulation is *information hiding*. The basic idea is that a class should expose a clean interface that allows interaction, but nothing about its internal state. So the general rule is that all state should start out as `private` and only have that access relaxed if there is a very, very good reason.

Encapsulation helps to minimise *coupling* between classes. High coupling between two class, A and B, implies that a change in A is likely to ripple through to B. In a large software project, you really don't want a change in one class to mean you have to go and fix up the other 200 classes! So we strive for low coupling.

It's also related to *cohesion*. A highly cohesive class contains only a set of strongly related functions rather than being a hotch-potch of functionality. We strive for high cohesion.

2.2 Inheritance

Inheritance

```
class Student {  
    public int age;  
    public String name;  
    public int grade;  
}
```

```
class Lecturer {  
    public int age;  
    public String name;  
    public int salary;  
}
```

- There is a lot of duplication here
- Conceptually there is a hierarchy that we're not really representing
- Both Lecturers and Students are people (no, really).
- We can view each as a kind of specialisation of a general person
 - They have all the properties of a person
 - But they also have some extra stuff specific to them

(I should not have used public variables here, but I did it to keep things simple)

Inheritance

```
class Person {  
    public int age;  
    Public String name;  
}
```

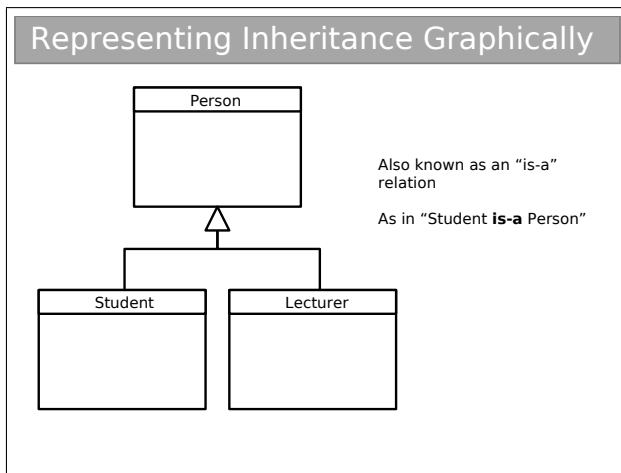
```
class Student extends Person {  
    public int grade;  
}
```

```
class Lecturer extends Person {  
    public int salary;  
}
```

- We create a *base class* (Person) and add a new notion: classes can *inherit* properties from it
 - Both state and functionality
- We say:
 - Person is the *superclass* of Lecturer and Student
 - Lecturer and Student *subclass* Person

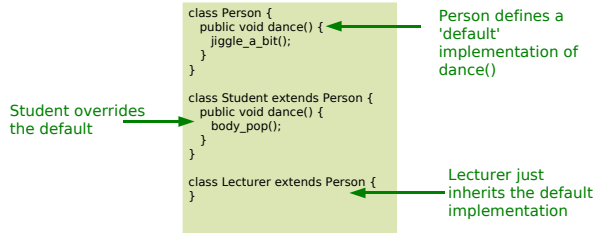
Unfortunately, we tend to use an array of different names for things in an inheritance tree. For A extends B, you might get any of:

- A is the superclass of B
- A is the parent of B
- A is the base class of B
- B is a subclass of A
- B is the child of A
- B derives from A
- B inherits from A
- B subclasses A



Overriding Functionality

- We might want to require that every Person can dance. But the way a Lecturer dances is not likely to be the same as the way a Student dances...



2.3 Abstract Classes

Abstract Methods

```
class Person {
    public void dance();
}

class Student extends Person {
    public void dance() {
        body_pop();
    }
}

class Lecturer extends Person {
    public void dance() {
        jiggle_a_bit();
    }
}
```

- There are times when we have a definite concept but we expect every specialism of it to have a different implementation. We want to enforce that without providing a default
- E.g. We want to enforce that all objects that are Persons support a dance() method
 - But we don't now think that there's a default dance()
- We specify an **abstract** dance method in the Person class
 - i.e. we don't fill in any implementation (code) at all in Person.

Abstract Classes

- Before we could write `Person p = new Person()`
- But now `p.dance()` is undefined
- Therefore we have implicitly made the class abstract i.e. It cannot be directly instantiated to an object
- Languages require some way to tell them that the class is meant to be abstract and it wasn't a mistake:

```
public abstract class Person {  
    public abstract void dance();  
}
```

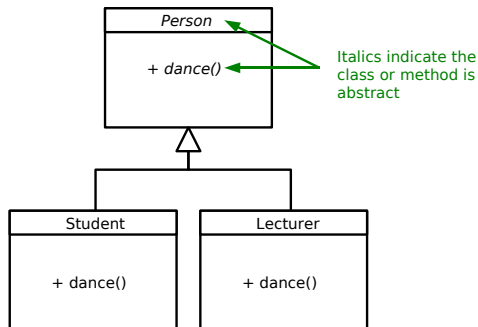
Java

```
class Person {  
    public:  
        virtual void dance()=0;  
}
```

C++

- Note that an abstract class can contain state variables that get inherited as normal
- Note also that, in Java, we can declare a class as abstract despite not specifying an abstract method in it!!

Representing Abstract Classes



2.4 Polymorphic Functions

(Subtype) Polymorphism

- As we descend our inheritance *tree* we specialise by adding more detail (a salary variable here, a dance() method there)
- So, in some sense, a Student object has all the information we need to make a Person (and some extra).
- It turns out to be quite useful to group things by their common ancestry in the inheritance tree
- We can do that semantically by expressions like:

```
Student s = new Student();  
Person p = (Person)s;
```

This is a *widening* conversion (we move up the tree, increasing generality: always OK)

```
Person p = new Person();  
Student s = (Student)p;
```

X

This would be a *narrowing* conversion (we try to move down the tree, but it's not allowed here because the real object doesn't have all the info to be a Student)

When we write `(Person)s` we say we are *casting* the Student object to a Person object.

It might look a bit odd, but just remember that going down an inheritance tree adds stuff. Every object has an intrinsic type when we create it and it can't be cast to anything below that type (because it doesn't contain the 'extra' stuff).

(Subtype) Polymorphism

```
Student s = new Student();  
Person p = (Person)s;  
p.dance();
```

- Assuming Person has a default dance() method, what should happen here??

- Option 1**

- Compiler says “p is of type Person”
- So p.dance() should do the default dance() action in Person

- Option 2**

- Compiler says “The object in memory is really a Student”
- So p.dance() should run the Student dance() method

Polymorphic behaviour

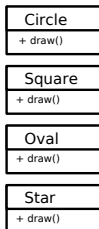


The Canonical Example

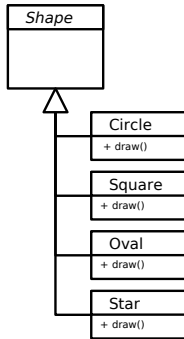
- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects

- Option 1**

- Keep a list of Circle objects, a list of Square objects,...
- Iterate over each list drawing each object in turn
- What has to change if we want to add a new shape?



The Canonical Example



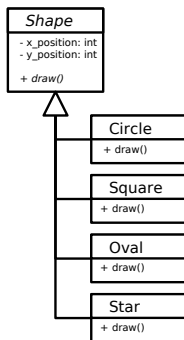
Option 2

- Keep a single list of Shape references
- Figure out what each object really is, narrow the reference and then `draw()`

```
For every Shape s in myShapeList
If (s is really a Circle)
  Circle c = (Circle)s;
  c.draw();
Else if (s is really a Square)
  Square sq = (Square)s;
  sq.draw();
Else if...
```

- What if we want to add a new shape?

The Canonical Example



Option 3 (Polymorphic)

- Keep a single list of Shape references
- Let the compiler figure out what to do with each Shape reference

```
For every Shape s in myShapeList
  s.draw();
```

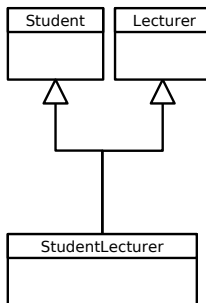
- What if we want to add a new shape?

Implementations

- Java
 - All methods are polymorphic. Full stop.
- C++
 - Only functions marked *virtual* are polymorphic
- Polymorphism is an extremely important concept that you need to make sure you understand...

2.5 Interfaces

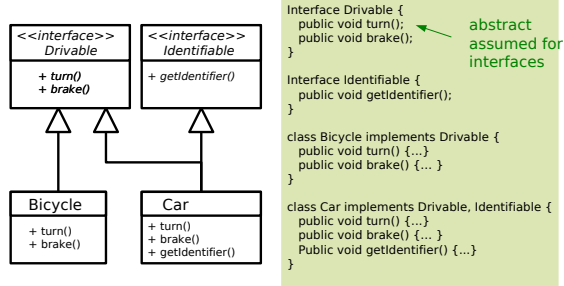
Multiple Inheritance



- What if we have a Lecturer who studies for another degree?
- If we do as shown, we have a bit of a problem
 - StudentLecturer inherits two different dance() methods
 - So which one should it use if we instruct a StudentLecturer to dance()?
- The Java designers felt that this kind of problem mostly occurs when you have designed your class hierarchy badly
- Their solution? **You can only extend (inherit) from one class in Java**
 - (which may itself inherit from another...)
 - This is a Java oddity (C++ allows multiple class inheritance)

Interfaces (Java only)

- Java has the notion of an **interface** which is like a class except:
 - There is no state whatsoever
 - All methods are abstract
- For an interface, there can then be no clashes of methods or variables to worry about, so we can allow multiple inheritance



Q6. Explain the differences between a class, an abstract class and an interface.

Q7. * Imagine you have two classes: `Employee` (which represents being an employee) and `Ninja` (which represents being a Ninja). An `Employee` has both state and behaviour; a `Ninja` has only behaviour.

You need to represent an employee who is also a ninja. By creating only one interface and only one class (`NinjaEmployee`), show how you can do this without having to copy method implementation code from either of the original classes.

Recap

- Important OOP concepts you need to understand:
 - Modularity (classes, objects)
 - Data Encapsulation
 - Inheritance
 - Abstraction
 - Polymorphism

2.6 Lifecycle of an Object

Constructors

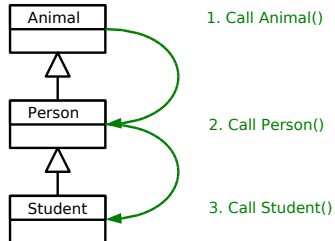
```
MyObject m = new MyObject();
```

- You will have noticed that the RHS looks rather like a function call
- Every Class has a **constructor**, which is a function that gets run automatically whenever you create an object
 - Allows you to initialise the object
 - No constructor specified? You'll get the *default constructor* (which is empty).
 - Multiple constructors specified?
 - Each constructor must have a different signature (i.e. a different set of arguments)
 - The compiler looks at the arguments you give in your **new** command and runs the appropriate one.

Constructor Chaining

- When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence

```
Student s = new Student();
```



Q8. Write a Java program that demonstrates the notion of constructor chaining.

Destructors

- Most OO languages have a notion of a destructor too
 - Gets run when the object is destroyed
 - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

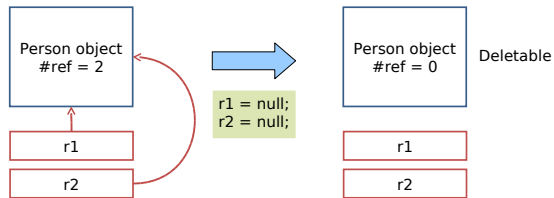
```
class FileReader {
public:
    FileReader() {
        f = fopen("myfile", "r");
    }
    ~FileReader() {
        fclose(f);
    }
private:
    FILE *file;
}
```

Cleaning Up

- A typical program creates lots of objects, not all of which need to stick around all the time
- **Approach 1:**
 - Allow the programmer to specify when objects should be deleted from memory
 - Lots of control, but what if they forget to delete an object?
- **Approach 2:**
 - Delete the objects automatically (**Garbage collection**)
 - But how do you know when an object is finished with?

Cleaning Up (Java)

- Java *reference counts*. i.e. it keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted



Cleaning Up (Java)

- **Good:**
 - System cleans up after us
- **Bad:**
 - It has to keep searching for objects with no references. This requires effort on the part of the CPU so it degrades performance.
 - We can't easily predict when an object will be deleted

Cleaning Up (Java)

- So we can't tell when a destructor would run - so Java doesn't have them!!
- It does have the notion of a **finalizer** that gets run when an object is garbage collected
 - BUT there's no guarantee an object will ever get garbage collected in Java...
 - Garbage Collection != Destruction

2.7 Class-Level Data

Class-Level Data and Functionality

```
public class ShopItem {
    private float price;
    private float VATRate = 0.175;

    public float GetSalesPrice() {
        return price*(1.0+VATRate);
    }

    public void SetVATRate(float rate) {
        VATRate=rate;
    }
}
```

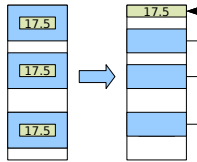
- Imagine we have a class ShopItem. Every ShopItem has an individual core price to which we need to add VAT
- Two issues here:
 1. If the VAT rate changes, we need to find every ShopItem object and run SetVATRate(...) on it. We could end up with different items having different VAT rates when they shouldn't...
 2. It is inefficient. Every time we create a new ShopItem object, we allocate another 32 bits of memory just to store exactly the same number!

- What we have is a piece of information that is class-level not object level
 - Each individual object has the same value at all times
- We throw in the **static** keyword:

```
public class ShopItem {
    private float price;
    private static float VATRate;
    ....
}
```

Variable created only once and has the lifetime of the program, not the object

Class-Level Data and Functionality



- We now have one place to update
- More efficient memory usage

- Can also make methods **static** too
 - A static method must be instance independent i.e. it can't rely on member variables in any way
- Sometimes this is obviously needed. For example:

```
public class Whatever {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Must be able to run this function without creating an object of type Whatever (which we would have to do in the main()...!)

Why use other static functions?

- A static function is like a function in ML - it can depend only on its arguments
 - Easier to debug (not dependent on any state)
 - Self documenting
 - Allows us to group related methods in a Class, but not require us to create an object to run them
 - The compiler can produce more efficient code since no specific object is involved

```
public class Math {  
    public float sqrt(float x) {...}  
    public double sin(float x) {...}  
    public double cos(float x) {...}  
}
```

```
public class Math {  
    public static float sqrt(float x) {...}  
    public static float sin(float x) {...}  
    public static float cos(float x) {...}  
}
```

vs

```
...  
Math mathobject = new Math();  
mathobject.sqrt(9.0);  
...
```

```
...  
Math.sqrt(9.0);  
...
```

2.8 Examples

One of the examples we will develop in class is a representation of a two dimensional vector (x, y) . This is obviously a very simple class, but it brings us to an interesting question of *mutability*. An *immutable* class cannot have its state changed after it has been created (you're familiar with this from ML, where everything is immutable). A *mutable* class can be altered somehow (usually as a side effect of calling a method).

To make a class immutable:

- Make sure all state is **private**.
- Consider making state **final** (this just tells the compiler that the value never changes once constructed).
- Make sure no method tries to change any internal state.

Some advantages of immutability:

- Simpler to construct, test and use
- Automatically thread safe (don't worry if this means nothing to you yet).
- Allows lazy instantiation of objects.

In fact, to quote *Effective Java* by Joshua Bloch:

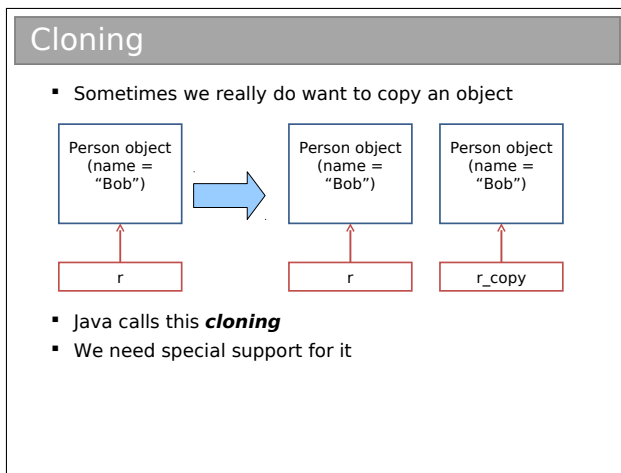
“Classes should be immutable unless there's a very good reason to make them mutable... If a class cannot be made immutable, limit its mutability as much as possible.”

- Q9.** Modify the Vector2D class to handle subtraction and scalar product.
- Q10.** Is the Vector2D class developed in lectures mutable or immutable?
- Q11.** Vector2D provides a method to add two vectors. Contrast the following approaches to providing an add method assuming Vector2D is (i) mutable, and (ii) immutable.
- (a) `public void add(Vector2D v)`
 - (b) `public Vector2D add(Vector2D v)`
 - (c) `public Vector2D add(Vector2D v1, Vector2D v2)`
 - (d) `public static Vector2D add(Vector2D v1,
Vector2D v2)`
- Q12.** A student wishes to create a class for a 3D vector and chooses to derive from the Vector2D class (i.e. `public void Vector3D extends Vector2D`). The argument is that a 3D vector is a “2D vector with some stuff added”. Explain the conceptual misunderstanding here.
- Q13.** Explain how to write a Java program that is ‘functional’ in nature i.e. no modifiable variables, methods are mathematical functions without side effects. Remember the `final` keyword.

Chapter 3

Java Miscellany

3.1 Copying or Cloning Java Objects

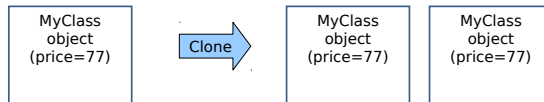


Cloning

- Every class in Java ultimately inherits from the **Object** class
 - The **Object** class contains a clone() method
 - So just call this to clone an object, right?
 - Wrong!
- Surprisingly, the problem is defining what copy actually means

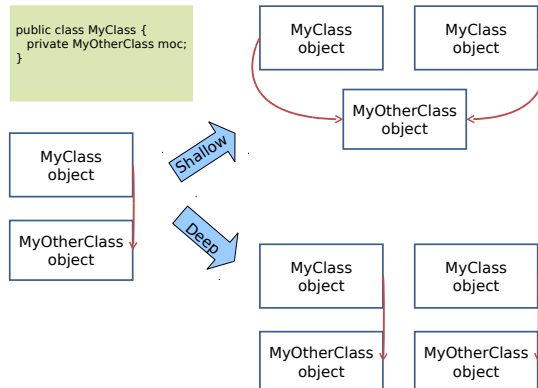
Cloning

```
public class MyClass {  
    private float price = 77;  
}
```



Shallow and Deep Copies

```
public class MyClass {  
    private MyOtherClass moc;  
}
```



Java Cloning

- So do you want shallow or deep?
 - The default implementation of `clone()` performs a shallow copy
 - But Java developers were worried that this might not be appropriate: they decided they wanted to know for sure that we'd thought about whether this was appropriate
- Java has a **Cloneable** interface
 - If you call `clone` on anything that doesn't extend this interface, it fails

Marker Interfaces

- If you go and look at what's in the Cloneable interface, you'll find it's empty!! What's going on?
- Well, the clone() method is already inherited from **Object** so it doesn't need to specify it
- This is an example of a **Marker Interface**
 - A marker interface is an empty interface that is used to label classes
 - This approach is found occasionally in the Java libraries

You might also see these marker interfaces referred to as *tag interfaces*. They are simply a way to label or tag a class. They can be very useful, but equally they can be a pain (you can't dynamically tag a class, nor can you prevent a tag being inherited by all subclasses).

Q14. An alternative strategy to clone()-ing an object is to provide a *copy constructor*. This is a constructor that takes the enclosing class as an argument and copies everything manually:

```
public class MyClass {
    private String mName;
    private int[] mData;

    // Copy constructor
    public MyClass(MyClass toCopy) {
        this.mName = toCopy.mName;
        // TODO
    }
    ...
}
```

- (a) Complete the copy constructor.
- (b) Make MyClass clone()-able (you should do a deep copy).
- (c) * Why might the Java designers have disliked copy constructors? [Hint: What happens if you want to copy an object that is being referenced using its parent type?].
- (d) * Under what circumstances is a copy constructor a good solution?

Q15. Consider the class below. What difficulty is there in providing a deep clone() method for it?

```
public class CloneTest {
    private final int[] mData = new int[100];
}
```

3.2 Distributing Java Classes

Distributing Classes

- So you've written some great classes that might be useful to others. You release the code. What if you've named your class the same as someone else?
 - E.g. There are probably 100s of "Vector" classes out there..!
- Most languages define some way that you can keep your descriptive class name without getting it confused with others.
- Java uses **packages**. A class belongs to a package
 - A nameless 'default' package unless you specify otherwise
 - You're supposed to choose a package name that is unique.
 - Sun decided you should choose your domain name
 - You do have your own domain name, right? ;)

Distributing Classes

```
package uk.cam.ac.rkh23; ← Class Whatever is part of this package
import uk.cam.ac.abc21.*; ← Import all the Classes from some
                           other package
Class Whatever {
...
}
```

- You get to do lots more about this in your practicals

Access Modifiers Revisited

- Most Languages:
 - **public** - everyone can access directly
 - **protected** - only subclasses can access directly
 - **private** - nothing can access directly
- Java adds:
 - **package** - anything in the same package can access directly

3.3 Java Class Libraries

Java Class Library

- Java the platform contains around **4,000** classes/interfaces
 - Data Structures
 - Networking, Files
 - Graphical User Interfaces
 - Security and Encryption
 - Image Processing
 - Multimedia authoring/playback
 - And more...
- All neatly(ish) arranged into packages (see API docs)

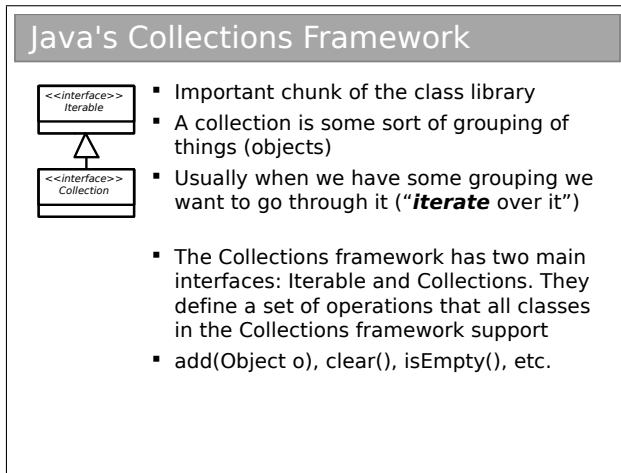
Remember Java is a *platform*, not just a programming language. It ships with a huge *class library*: that is to say that Java itself contains a big set of built-in classes for doing all sorts of useful things like:

- Complex data structures and algorithms
- I/O (input/output: reading and writing files, etc)
- Networking
- Graphical interfaces

Of course, most programming languages have built-in classes, but Java has a big advantage. Because Java code runs on a virtual machine, the underlying platform is abstracted away. For C++, for example, the compiler ships with a fair few data structures, but things like I/O and graphical interfaces are completely different for each platform (windows, OSX, Linux, whatever). This means you usually end up using lots of third-party libraries to get such ‘extras’ — not so in Java.

There is, then, good reason to take a look at the Java class library to see how it is structured. In fact, we’ll see a lot of design patterns used in the library...

3.3.1 Collections and Generics

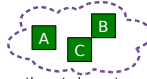


The Java Collections framework is a set of interfaces and classes that handles groupings of objects and allows us to implement various algorithms invisibly to the user (you'll learn about the algorithms themselves next term).

Major Collections Interfaces

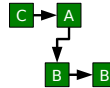
- **<<interface>> Set**

- Like a mathematical set in DM 1
- A collection of elements with no duplicates
- Various concrete classes like TreeSet (which keeps the set elements sorted)



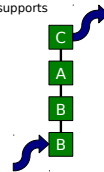
- **<<interface>> List**

- An ordered collection of elements that may contain duplicates
- ArrayList, Vector, LinkedList, etc.



- **<<interface>> Queue**

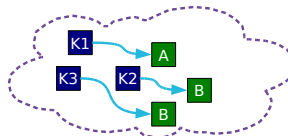
- An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
- PriorityQueue, LinkedList, etc.



Major Collections Interfaces

- **<<interface>> Map**

- Like relations in DM 1
- Maps key objects to value objects
- Keys must be unique
- Values can be duplicated and (sometimes) null.



There are other interfaces in the Collections class, and you may want to poke around in the API documentation. In day-to-day program-

ming, however, these are likely to be the interfaces you use.

Obviously, you can't use the interfaces directly. So Java includes a few implementations that implement sensible things. Again, you will find them in the API docs, but as an example for **Set**:

TreeSet. A **Set** that keeps the elements in sorted order so that when you iterate over them, they come out in order.

HashSet. A **Set** that uses a technique called hashing (don't worry — you're not meant to know about this yet) that happens to make certain operations (add, remove, etc) very efficient. However, the order the elements iterate over is neither obvious nor constant.

Now, don't worry about what's going on behind the scenes (that comes in the Algorithms course), just recognise that there are a series of implementations in the class library that you can use, and that each has different properties.

Generics

- The original Collections framework just dealt with collections of **Objects**
 - Everything in Java “is-a” **Object** so that way our collections framework will apply to any class we like without any special modification.
 - It gets messy when we get something from our collection though: it is returned as an **Object** and we have to do a narrowing conversion to make use of it:

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));

// Loop through
Iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

Generics

- It gets worse when you realise that the add() method doesn't stop us from throwing in random objects:

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

Going to fail for the second element!
(But it will compile: the error will be at runtime)

Generics

- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// Make a TreeSet of Integers
TreeSet<Integer> ts = new TreeSet<Integer>();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));

// Loop through
iterator<Integer> it = ts.iterator();
while(it.hasNext()) {
    Integer i = it.next();
}
```

Won't even compile

No need to cast :-)

Notation in Java API

- Set<E>
- List<E>
- Queue<E>
- Map<K,V>

Here the letter between the brackets just signifies some class, so you might do:

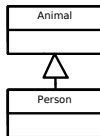
```
TreeSet<Person> ts = new TreeSet<Person>()
```

Polymorphism Revisited

- You might recognise Generics as the “polymorphism” you met in FoCS when using ML.
- Both allow you to write code that works for multiple types
 - (Parametric) Polymorphism [FP] or Generics [OOP]
 - The types are determined at compile-time
 - (Sub-type or ad-hoc) Polymorphism [OOP]
 - The types are determined at run-time
 - Needs an inheritance tree

For the most part, when programmers say “polymorphism”, it’s usually with reference to the sub-type polymorphism.

Generics and SubTyping



```
// Object casting
Person p = new Person();
Animal o = (Animal) p;

// List casting
List<Person> plist = new LinkedList<Person>();
List<Animal> alist = (List<Animal>)plist;
```

So a list of **Persons** is a list of **Animals**, yes?

Q16. Java generics can handle this issue using *wildcards*. Explain how these work. *Yes, some research will be required...*

3.3.2 Comparing Java Objects

Comparing Primitives

>	Greater Than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to

- Clearly compare the value of a primitive
- But what does `(object1==object2)` mean??
 - Same object?
 - Same state ("value") but different object?

The problem is that we deal with references to objects, not objects. So when we compare two things, do we compare the references of the objects they point to? As it turns out, both can be useful so we want to support both.

Option 1: a==b, a!=b

- These compare the *references*

```
Person p1 = new Person("Bob");
Person p2 = new Person("Bob");

(p1==p2);
(p1!=p2);
p1==p1;
```

False (references differ)

True (references differ)

True (references the same)

```
String s = "Hello";
if (s=="Hello") System.out.println("Hello");
else System.out.println("Nope");
```

Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
- Returns boolean, so can only test equality
- Override it if you want it to do something different
- Most (all?) of the core Java classes have properly implemented equals() methods

```
Person p1 = new Person("Bob");
Person p2 = new Person("Bob");

(p1==p2);

String s1 = "Bob";
String s2 = "Bob";

(s1==s2);
```

False (we haven't overridden the equals() method so it just compares references)

True (String has equals() overridden)

I find this mildly irritating: every class you use will support equals() but you'll have to check whether or not it has been overridden to do

something other than ==. Personally, I only use equals() on objects from core Java classes.

Option 3: Comparable<T> Interface

`int compareTo(T obj);`

- Part of the Collections Framework
- Returns an integer, r:
 - r<0 This object is less than obj
 - r==0 This object is equal to obj
 - r>0 This object is greater than obj

Option 3: Comparable<T> Interface

```
public class Point implements Comparable<Point> {
    private final int mX;
    private final int mY;
    public Point (int, int y) { mX=x; mY=y; }

    // sort by y, then x
    public int compareTo(Point p) {
        if ( mY>p.mY) return 1;
        else if (mY<p.mY) return -1;
        else {
            if (mX>p.mX) return 1;
            else if (mX<p.mX) return -1;
            else return 0;
        }
    }
}
```

```
// This will be sorted automatically by y, then x
Set<Point> list = new TreeSet<Point>();
```

Note that the class itself contains the information on how it is to be sorted: we say that it has a *natural ordering*.

Q17. Write a class that represents a 3D point (x,y,z). Give it a natural order such that values are sorted in ascending order by z, then y, then x.

Option 4: Comparator<T> interface

```
int compareTo(T obj1, T obj2)
```

- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections object.

At first glance, it may seem that `Comparator` doesn't add much over `Comparable`. However it's very useful to be able to specify Comparators and apply them dynamically to Collections. If you look in the API, you will find that `Collections` has a *static* method `sort(List l, Comparator, c)`.

So, imagine we have a class `Student` that stores the forename, surname and exam percentage as a `String`, `String`, and a `float` respectively. The natural ordering of the class sorts by surname. We

might then supply two `Comparator` classes: `ForenameComparator` and `ExamScoreComparator` that do as you would expect. Then we could write:

```
List list = new SortedList();

// Populate list
// List will be sorted naturally
...

// Sort list by forename
Collections.sort(list, new ForenameComparator());

// Sort list by exam score
Collections.sort(list, new ExamScoreComparator());
```

- Q18.** (a) Remembering the idea of data encapsulation, write the code for the `Student` class.
- (b) Write the code for the two `Comparators`.
- (c) Write a program that demonstrates that your code works.
- (d) Without generics, the `Comparator` signature has to be `compareTo(Object o1, Object o2)`. Show how you would make this work for your `Student` class. Assume that it only makes sense to compare two `Student` objects.

Q19. The user of the class `Car` below wishes to maintain a collection of `Car` objects such that they can be iterated over in some specific order.

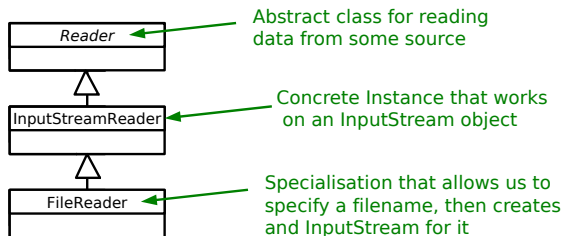
```
public class Car {  
    private String manufacturer;  
    private int age;  
}
```

- (a) Show how to keep the collection sorted alphabetically by the manufacturer without writing a `Comparator`.
- (b) Show how to keep the collection sorted by {manufacturer, age}. i.e. sort first by manufacturer, and sub-sort by age.

3.3.3 Java's I/O Framework

Java's I/O framework

- Support for system input and output (from/to sources such as network, files, etc).

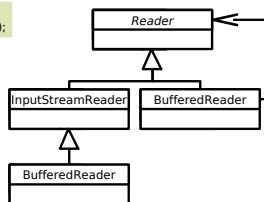


Speeding it up

- In general file I/O is slowwwww
- One trick we can use is that whenever we're asked to read some data in (say one byte) we actually read lots more in (say a kilobyte) and buffer it somewhere on the assumption that it will be wanted eventually and it will just be there in memory, waiting for us. :-)
- Java supports this in the form of a **BufferedReader**

```
FileReader f = new FileReader();
BufferedReader br = new BufferedReader(f);
```

- Whenever we call read() on a BufferedReader it looks in its buffer to see whether it has the data already
- If not it passes the request onto the Reader object
- We'll come back to this...



The reason file I/O is typically so slow is that hard drives take a long time to deliver information. They contain big, spinning disks and the read head has to move to the correct place to read the data from, then wait until the disc has spun around enough to read all the data it wanted (think old 12 inch record players). Contrast this with memory (in the sense of RAM), where you can just jump wherever you like without consequence and with minimal delay.

The `BufferedReader` simply tries to second guess what will happen next. If you asked for the first 50 bytes of data from a file, chances are you'll be asking for the next 50 bytes (or whatever) before long, so it loads that data into a buffer (i.e. into RAM) so that *if* you do turn out to want it, there will be little or no delay. If you don't use it: oh well, we tried.

The key thing is to look at the tree structure: a `BufferedReader` *is-a* `Reader` but also *has-a* `Reader`. The idea is that a `BufferedReader` has all the capabilities of the `Reader` object that it contains, but also adds some extra functionality.

For example, a `Reader` allows you to read text in byte-by-byte using `read()`. If you have a string of text, you have to read it in character by character until you get to the terminating character that marks the end of the string. A `BufferedReader` reads ahead and can read the entire string in one go: it adds a `readLine()` function to do so. But it still supports the `read()` functionality if you want to do it the hard way.

The really nice thing is that we don't have to write a `BufferedReader` for a `Reader` that we create from scratch. I could create a `SerialPortReader` that derives from `Reader` and I could immediately make a `BufferedReader` for it without having to write any more code.

This sort of solution crops up again and again in OOP, and this is one of the "Design Patterns" we'll talk about later in the course. So

you may want to come back to this at the end of the course if you don't fully 'get' it now.

Chapter 4

Design Patterns

4.1 Introduction

Coding anything more complicated than a toy program usually benefits from forethought. After you've coded a few medium-sized pieces of object-oriented software, you'll start to notice the same general problems coming up over and over. And you'll start to automatically use the same solution each time. We need to make sure that set of default solutions is a good one!

In his 1991 PhD thesis, Erich Gamma compared this to the field of architecture, where recurrent problems are tackled by using known good solutions. The follow-on book ([Design Patterns: Elements of Reusable Object-Oriented Software, 1994](#)) identified a series of commonly encountered problems in object-oriented software design and 23 solutions that were deemed elegant or good in some way. Each solution is known as a *Design Pattern*:

A Design Pattern is a general reusable solution to a commonly occurring problem in software design.

The modern list of design patterns is ever-expanding and there is no shortage of literature on them. In this course we will be looking at a few key patterns and how they are used.

4.1.1 So Design Patterns are like coding recipes?

No. Creating software by stitching together a series of Design Patterns is like painting by numbers — it's easy and it probably works, but it doesn't produce a Picasso! Design Patterns are about intelligent solutions to a series of generalised problems that you *may* be able to identify in your software. You might find they don't apply to your problem, or that they need adaptation. You simply can't afford to disengage your brain (sorry!)

4.1.2 Why Bother Studying Them?

Design patterns are useful for a number of things, not least:

1. They encourage us to identify the fundamental aims of given pieces of code
2. They save us time and give us confidence that our solution is sensible
3. They demonstrate the power of object-oriented programming
4. They demonstrate that naïve solutions are bad
5. They give us a common vocabulary to describe our code

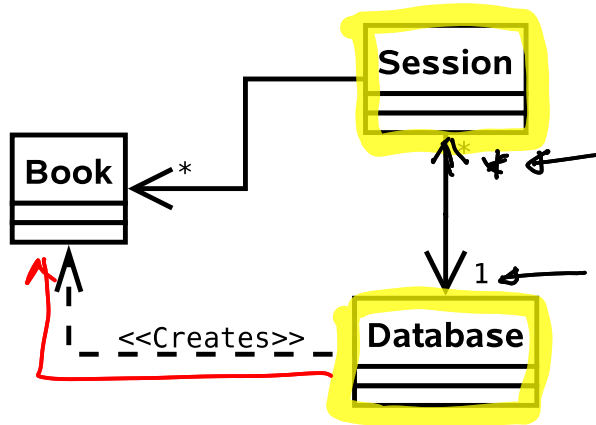
The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments¹ with a single word, the code becomes more readable and maintainable. Furthermore, you can insert the word into the class name itself, making the class self-describing.

¹You are commenting your code liberally, aren't you?

4.2 Design Patterns By Example

We're going to develop a simple example to look at a series of design patterns. Our example is a new online venture selling books. We will be interested in the underlying ("back-end") code — this isn't going to be a web design course!

We start with a very simple setup of classes. For brevity we won't be annotating the classes with all their members and functions. You'll need to use common sense to figure out what each element supports.



Session. This class holds everything about a current browser session (originating IP, user, shopping basket, etc).

Database. This class wraps around our database, hiding away the query syntax (i.e. the SQL statements or similar).

Book. This class holds all the information about a particular book.

4.3 Supporting Multiple Products

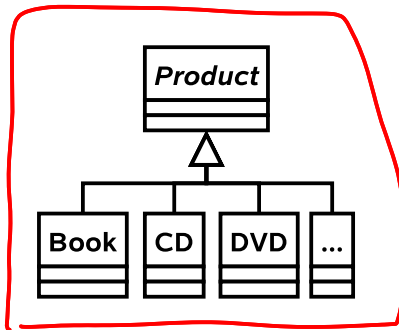
Problem: Selling books is not enough. We need to sell CDs and DVDs too. And maybe electronics. Oh, and sports equipment. And...

Solution 1: Create a new class for every type of item.



- ✓ It works.
- ✗ We end up duplicating a lot of code (all the products have prices, sizes, stock levels, etc).
- ✗ This is difficult to maintain (imagine changing how the VAT is computed...).

Solution 2: Derive from an abstract base class that holds all the common code.



- ✓ “Obvious” object oriented solution

- ✓ If we are smart we would use polymorphism² to avoid constantly checking what type a given **Product** object is in order to get product-specific behaviour.

4.3.1 Generalisation

This isn't really an 'official' pattern, because it's a rather fundamental thing to do in object-oriented programming. However, it's important to understand the power inheritance gives us under these circumstances.

²There are two types of polymorphism. *Ad-hoc* polymorphism (a.k.a. runtime or dynamic polymorphism) is concerned with object inheritance. It is familiar to you from Java, when the computer automatically figures out which version of an inherited method to run. *Parametric* polymorphism (a.k.a. static polymorphism) is where the compiler figures out which version of a type to use *before* the program runs. You are familiar with this in ML, but you also find it in C++ (templates) and Java (look up generics).

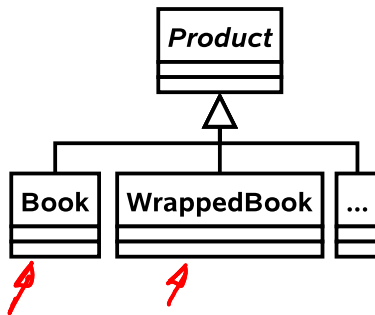
4.4 The Decorator Pattern

Problem: You need to support gift wrapping of products.

Solution 1: Add variables to the **Product** class that describe whether or not the product is to be wrapped and how.

- ✓ It works. In fact, it's a good solution if all we need is a binary flag for wrapped/not wrapped.
- ✗ As soon as we add different wrapping options and prices for different product types, we quickly clutter up **Product**.
- ✗ Clutter makes it harder to maintain.
- ✗ Clutter wastes storage space.

Solution 2: Add **WrappedBook** (etc.) as subclasses of **Product** as shown.

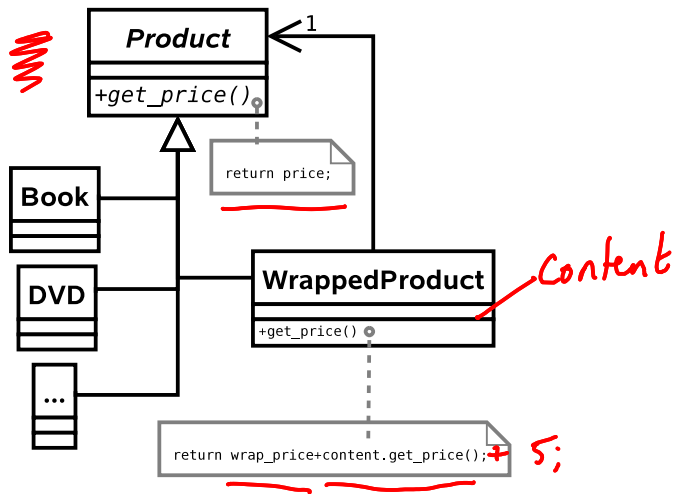


Implementing this solution is a shortcut to the Job centre.

- ✓ We are efficient in storage terms (we only allocate space for wrapping information if it is a wrapped entity).
- ✗ We instantly double the number of classes in our code.

- ✗ If we change **Book** we have to remember to mirror the changes in **WrappedBook**.
- ✗ If we add a new type we must create a wrapped version. This is bad because we can forget to do so.
- ✗ We can't convert from a **Book** to a **WrappedBook** without copying lots of data.

Solution 3: Create a general **WrappedProduct** class that is both a subclass of **Product** and references an instance of one of its siblings. Any state or functionality required of a **WrappedProduct** is 'passed on' to its internal sibling, unless it relates to wrapping.



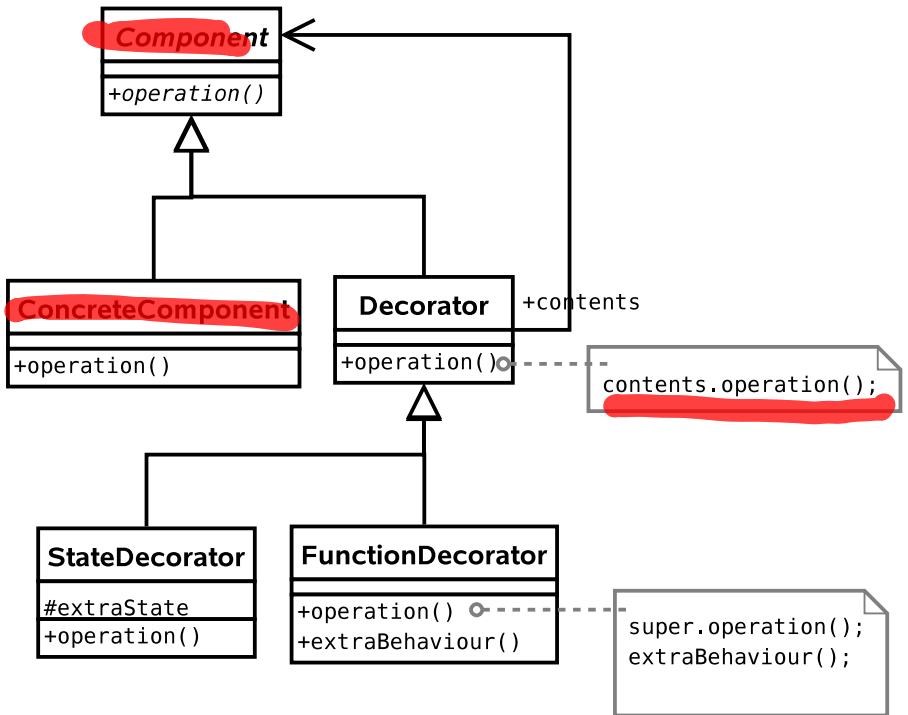
- ✓ We can add new product types and they will be automatically wrappable.
- ✓ We can dynamically convert an established product object into a wrapped product and back again without copying overheads.

- ✗ We can wrap a wrapped product!
- ✗ We could, in principle, end up with lots of chains of little objects in the system

4.4.1 Generalisation

This is the **Decorator** pattern and it allows us to add functionality to a class *dynamically* without changing the base class or having to derive a new subclass. Real world example: humans can be ‘decorated’ with contact lenses to improve their vision.

Java 1/0



Note that we can use the pattern to add state (variables) or functionality (methods), or both if we want. In the diagram above, I have explicitly allowed for both options by deriving **StateDecorator** and **FunctionDecorator**. This is usually unnecessary — in our book seller example we only want to decorate one thing so we might as well just put the code into **Decorator**.

4.5 State Pattern

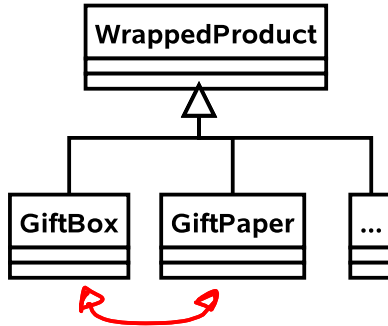
Problem: We need to handle a lot of gift options that the customer may switch between at will (different wrapping papers, bows, gift tags, gift boxes, gift bags, ...).

Solution 1: Take our `WrappedProduct` class and add a lot of if/then statements to the function that does the wrapping — let's call it `initiate_wrapping()`.

```
void initiate_wrapping() {  
    if (wrap.equals("BOX")) {  
        ...  
    }  
    else if (wrap.equals("BOW")) {  
        ...  
    }  
    else if (wrap.equals("BAG")) {  
        ...  
    }  
    else ...  
}
```

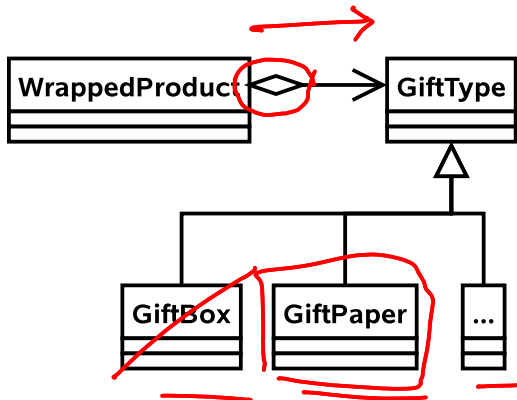
- ✓ It works.
- ✗ The code is far less readable.
- ✗ Adding a new wrapping option is ugly.

Solution 2: We basically have type-dependent behaviour, which is code for “use a class hierarchy”.



- ✓ This is easy to extend.
- ✓ The code is neater and more maintainable.
- ✗ What happens if we need to change the type of the wrapping (from, say, a box to a bag)? We have to construct a new **GiftBag** and copy across all the information from a **GiftBox**. Then we have to make sure every reference to the old object now points to the new one. This is hard!

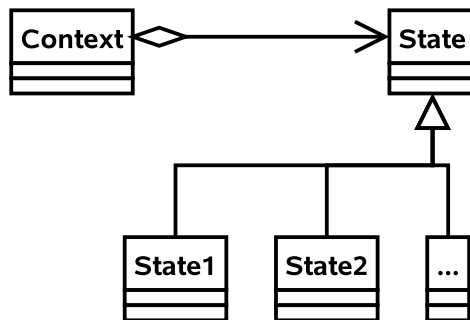
Solution 3: Let's keep our idea of representing states with a class hierarchy, but use a new abstract class as the parent:



Now, every `WrappedProduct` *has-a* `GiftType`. We have retained the advantages of solution 2 but now we can easily change the wrapping type in-situ since we know that only the `WrappedObject` object references the `GiftType` object.

4.5.1 Generalisation

This is the **State** pattern and it is used to permit an object to change its behaviour *at run-time*. A real-world example is how your behaviour may change according to your mood. e.g. if you're angry, you're more likely to behave aggressively.



Q20. Suppose you have an abstract class `TeachingStaff` with two concrete subclasses: `Lecturer` and `Professor`. Problems arise when a lecturer gets promoted because we cannot convert a `Lecturer` object to a `Professor` object. Using the **State** pattern, show how you would redesign the classes to permit promotion.

Students.

4.6 Strategy Pattern

Problem: Part of the ordering process requires the customer to enter a postcode which is then used to determine the address to post the items to. At the moment the computation of address from postcode is very slow. One of your employees proposes a different way of computing the address that should be more efficient. How can you trial the new algorithm?

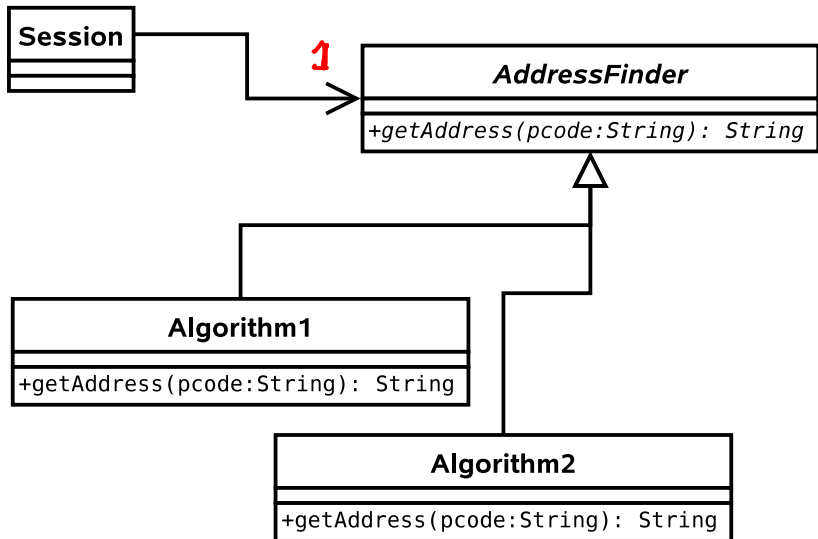
Solution 1: Let there be a class `AddressFinder` with a method `getAddress(String postcode)`. We could add lots of if/then/else statements to the `getAddress()` function.

```
String getAddress(String postcode) {  
    if (algorithm==0) {  
        // Use old approach  
        ...  
    }  
    else if (algorithm==1) {  
        // use new approach  
        ...  
    }  
}
```

- ✗ The `getAddress()` function will be huge, making it difficult to read and maintain.
- ✗ Because we must edit `AddressFinder` to add a new algorithm, we have violated the open/closed principle³.

³This states that a class should be open to extension but closed to modification. So we allow classes to be easily extended to incorporate new behavior without modifying existing code. This makes our designs resilient to change but flexible enough to take on new functionality to meet changing requirements.

Solution 2: Make `AddressFinder` abstract with a single abstract function `getAddress(String pcode)`. Derive a new class for each of our algorithms.

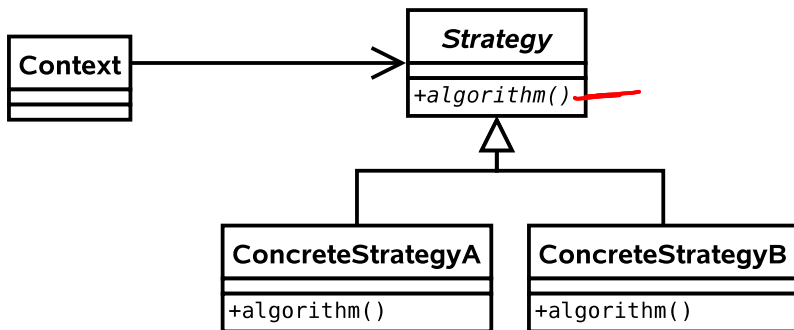


- ✓ We encapsulate each algorithm in a class.
- ✓ Code is clean and readable.
- ✗ More classes kicking around

4.6.1 Generalisation

This is the **Strategy** pattern. It is used when we want to support different algorithms that achieve the same goal. Usually the algorithm is fixed when we run the program, and doesn't change. A real life

example would be two consultancy companies given the same brief. They will hopefully produce the same result, but do so in different ways. i.e. they will adopt different strategies. From the (external) customer's point of view, the result is the same and he is unaware of how it was achieved. One company may achieve the result faster than the other and so would be considered 'better'.



Note that this is essentially the same UML as the **State** pattern! The *intent* of each of the two patterns is quite different however:

- **State** is about encapsulating behaviour that is linked to specific internal state within a class.
- Different states produce different outputs (externally the class behaves differently).
- **State** assumes that the state will continually change at runtime.
- The usage of the **State** pattern is normally invisible to external classes. i.e. there is no `setState(State s)` function.

- **Strategy** is about encapsulating behaviour in a class. This behaviour does not depend on internal variables.

- Different concrete **Strategy**s may produce exactly the same output, but do so in a different way. For example, we might have a new algorithm to compute the standard deviation of some variables. Both the old algorithm and the new one will produce the same output (hopefully), but one may be faster than the other. The **Strategy** pattern lets us compare them cleanly.
- **Strategy** in the strict definition usually assumes the class is selected at compile time and not changed during runtime.
- The usage of the **Strategy** pattern is normally visible to external classes. i.e. there will be a `setStrategy(Strategy s)` function or it will be set in the constructor.

However, the similarities do cause much debate and you will find people who do not differentiate between the two patterns as strongly as I tend to.

Decorator - Add state/functionality dynamically (at run-time)
 ↙ remove

State - Changing behaviour at run time

Strategy - Supporting multiple ways of doing something

Singleton - Ensuring only one object created of a certain type

4.7 Composite Pattern

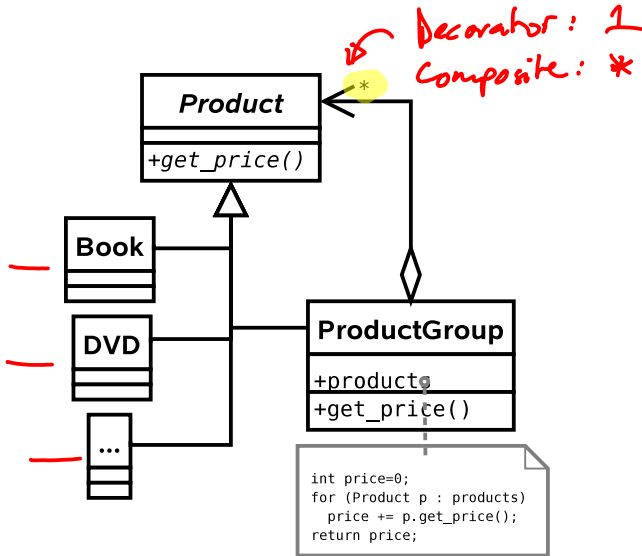
Problem: We want to support entire *groups* of products. e.g. The Lord of the Rings gift set might contain all the DVDs (plus a free cyanide capsule).

Solution 1: Give every **Product** a group ID (just an int). If someone wants to buy the entire group, we search through all the **Products** to find those with the same group ID.



- ✓ Does the basic job.
- ✗ What if a product belongs to no groups (which will be the majority case)? Then we are wasting memory and cluttering up the code. ||
- ✗ What if a product belongs to multiple groups? How many groups should we allow for?

Solution 2: Introduce a new class that encapsulates the notion of groups of products:



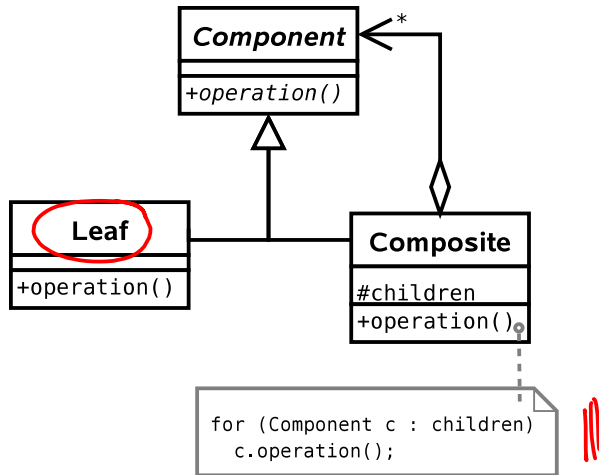
If you're still awake, you may be thinking this is a bit like the **Decorator** pattern, except that the new class supports associations with multiple **Products** (note the * by the arrowhead). Plus the intent is different – we are not adding new functionality but rather supporting the same functionality for groups of **Products**.

- ✓ Very powerful pattern.
- ✗ Could make it difficult to get a list of all the individual objects in the group, should we want to.

4.7.1 Generalisation

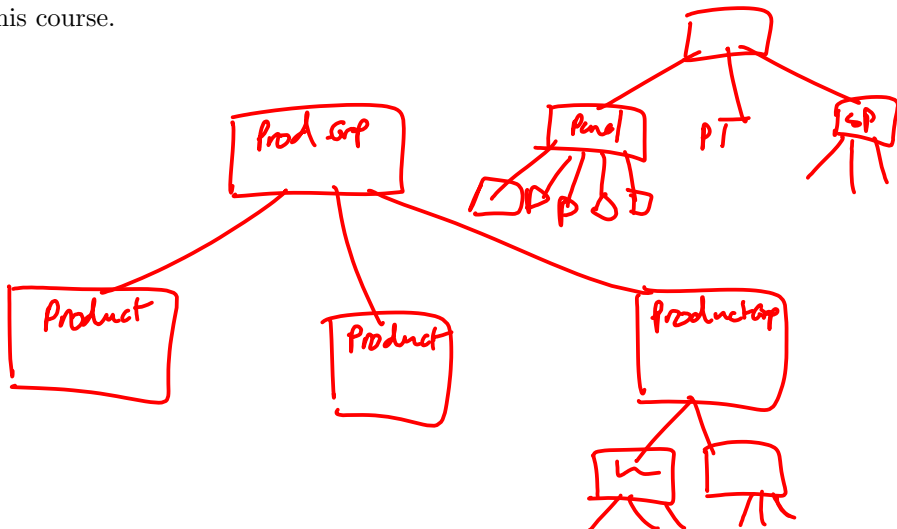
This is the **Composite** pattern and it is used to allow objects and collections of objects to be treated uniformly. Almost any hierarchy uses the **Composite** pattern. e.g. The CEO asks for a progress

report from a manager, who collects progress reports from all those she manages and reports back.



Notice the terminology in the general case: we speak of **Leafs** because we can use the Composite pattern to build a *tree* structure. Each **Composite** object will represent a node in the tree, with children that are either **Composites** or **Leafs**.

This pattern crops up a lot, and we will see it in other contexts later in this course.



Q21. A drawing program has an abstract `Shape` class. Each `Shape` object supports a `draw()` method that draws the relevant shape on the screen (as per the example in lectures). There are a series of concrete subclasses of `Shape`, including `Circle` and `Rectangle`. The drawing program keeps a list of all shapes in a `List<Shape>` object.

- (a) Should `draw()` be an abstract method?
- (b) Write Java code for the function in the main application that draws all the shapes on each screen refresh.
- (c) Show how to use the Composite pattern to allow sets of shapes to be grouped together and treated as a single entity.
- (d) Which design pattern would you use if you wanted to extend the program to draw frames around some of the shapes? Show how this would work.

4.8 Singleton Pattern

Problem: Somewhere in our system we will need a database and the ability to talk to it. Let us assume there is a **Database** class that abstracts the difficult stuff away. We end up with lots of simultaneous user **Sessions**, each wanting to access the database. Each one creates its own **Database** object and connects to the database over the network. The problem is that we end up with a lot of **Database** objects (wasting memory) and a lot of open network connections (bogging down the database).

What we want to do here is to ensure that there is only one **Database** object ever instantiated and every **Session** object uses it. Then the **Database** object can decide how many open connections to have and can queue requests to reduce instantaneous loading on our database (until we buy a half decent one).

Solution 1: Use a global variable of type **Database** that everything can access from everywhere.

- ✗ Global variables are less desirable than David Hasselhoff's greatest hits.
- ✗ Can't do it in Java anyway...

Solution 2: Use a public static variable which everything uses (this is as close to global as we can get in Java).

```
→ public class System {  
    public static Database database;  
}
```



```

...

public static void main(String[]) {
    // Always gets the same object
    Database d = System.database;
}

```

- ✗ This is really just global variables by the back door.
- ✗ Nothing fundamentally prevents us from making multiple **Database** objects!

Solution 3: Create an instance of **Database** at startup, and pass it as a constructor parameter to every **Session** we create, storing a reference in a member variable for later use.

```

public class System {
    public System(Database d) {...}
}

public class Session {
    public Session(Database d) {...}
}

...

public static void main(String[]) {
    Database d = new Database();
    System sys = new System(d);
    Session sesh = new Session(d);
}

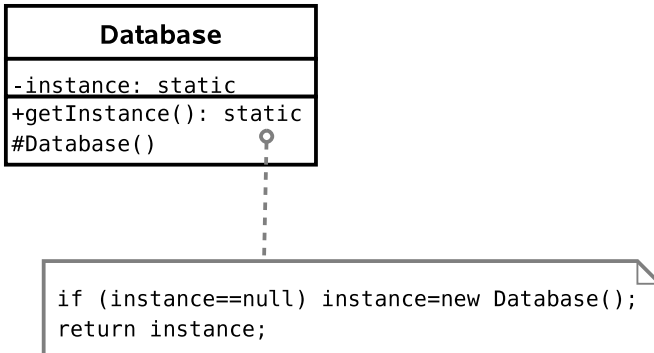
```

}

- ✗ This solution could work, but it doesn't *enforce* that only one **Database** be instantiated – someone could quite easily create a new **Database** object and pass it around.
- ✗ We start to clutter up our constructors.
- ✗ It's not especially intuitive. We can do better.

Solution 4: (Singleton) Let's adapt Solution 2 as follows. We *will* have a single static instance. However we will access it through a static member function. This function, **getInstance()** will either create a new **Database** object (if it's the first call) or return a reference to the previously instantiated object.

Of course, nothing stops a programmer from ignoring the **getInstance()** function and just creating a new **Database** object. So we use a neat trick: we make the constructor *private* or *protected*. This means code like `new Database()` isn't possible from an arbitrary class.

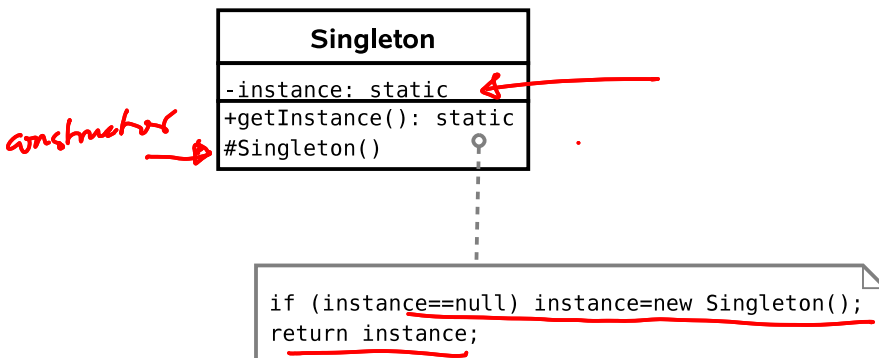


- ✓ *Guarantees* that there will be only one instance.

- ✓ Code to get a Database object is neat and tidy and intuitive to use. e.g. (Database d=Database.getInstance());
- ✓ Avoids clutter in any of our classes.
- ✗ Must take care in Java. Either use a dedicated package or a private constructor (see below).
- ✗ Must remember to disable **clone()**-ing!

4.8.1 Generalisation

This is the **Singleton** pattern. It is used to provide a global point of access to a class that should be instantiated only once.



There is a caveat with Java. If you choose to make the constructor protected (this would be useful if you wanted a singleton base class for multiple applications of the singleton pattern, and is actually the 'official' solution) you have to be careful. ⚠

Protected members are accessible to the class, any subclasses, *and all classes in the same package*. Therefore, any class in the same package



as your base class will be able to instantiate **Singleton** objects at will, using the `new` keyword!

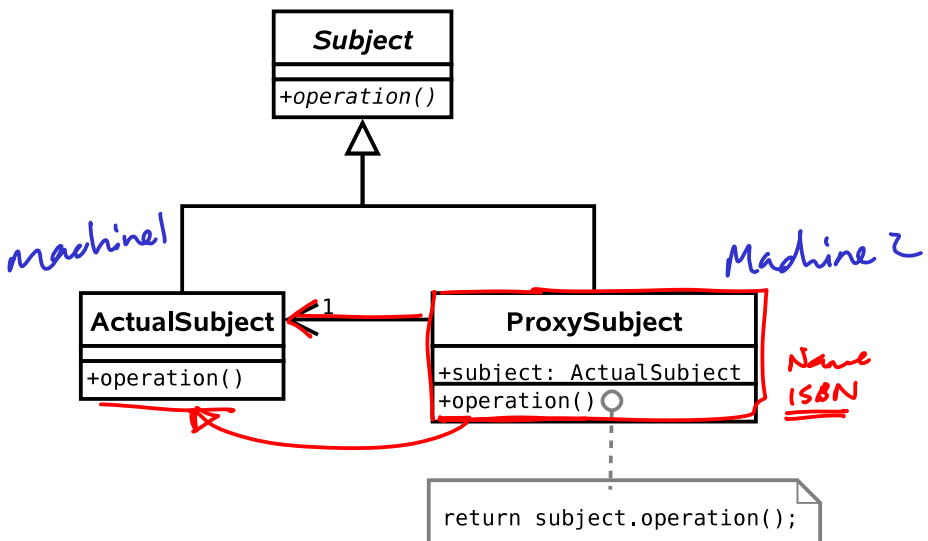
Additionally, we don't want a crafty user to subclass our singleton and implement **Cloneable** on their version. The examples sheet asks you to address this issue.

Q22. One technique to break a **Singleton** object is to extend it and implement **Cloneable**. This allows **Singleton** objects can be cloned, breaking the fundamental goal of a **Singleton**! Write a Java **Singleton** class that is not `final` but still prevents subclasses from being cloned.

4.9 Proxy Pattern(s)

The **Proxy** pattern is a very useful *set* of three patterns: **Virtual Proxy**, **Remote Proxy**, and **Protection Proxy**.

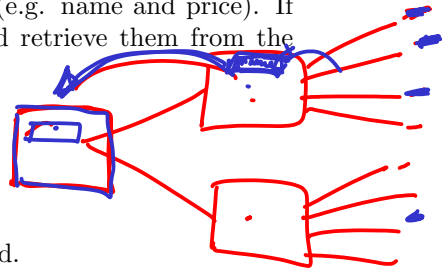
All three are based on the same general idea: we can have a placeholder class that has the same interface as another class, but actually acts as a pass through for some reason.



4.9.1 Virtual Proxy

Problem: Our **Product** subclasses will contain a lot of information, much of which won't be needed since 90% of the products won't be selected for more detail, just listed as search results.

Solution : Here we apply the **Proxy** pattern by only loading part of the full class into the proxy class (e.g. name and price). If someone requests more details, we go and retrieve them from the database.



4.9.2 Remote Proxy

Problem: Our server is getting overloaded.

Solution : We want to run a farm of servers and distribute the load across them. Here a particular object resides on server A, say, whilst servers B and C have proxy objects. Whenever the proxy objects get called, they know to contact server A to do the work. i.e. they act as a pass-through.

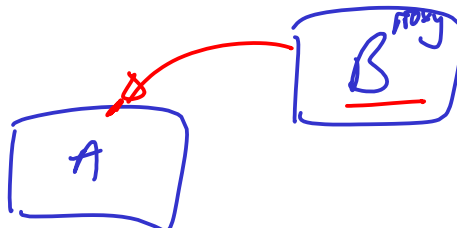
Note that once server B has bothered going to get something via the proxy, it might as well keep the result locally in case it's used again (saving us another network trip to A). This is *caching* and we'll return to it shortly.

4.9.3 Protection Proxy

Problem: We want to keep everything as secure as possible.

Solution : Create a **User** class that encapsulates all the information about a person. Use the **Proxy** pattern to fill a proxy class with public information. Whenever private information is requested of the proxy, it will only return a result if the user has been authenticated.

In this way we avoid having private details in memory unless they have been authorised.



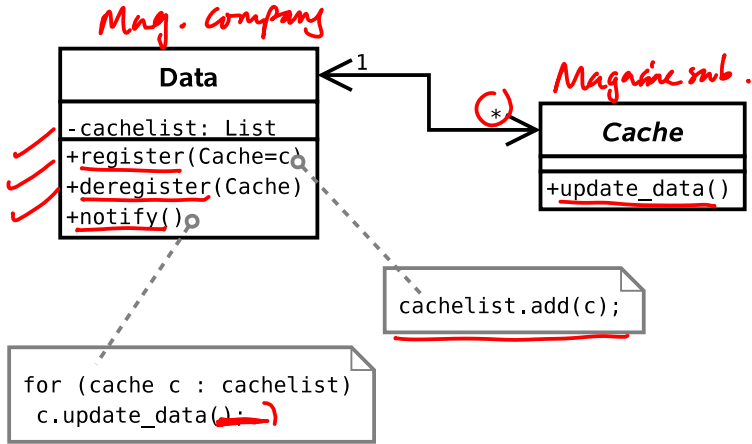
4.10 Observer Pattern

Problem: We use the **Remote Proxy** pattern to distribute our load. For efficiency, proxy objects are set to cache information that they retrieve from other servers. However, the originals could easily change (perhaps a price is updated or the exchange rate moves). We will end up with different results on different servers, dependent on how old the cache is!!

Solution 1: Once a proxy has some data, it keeps polling the authoritative source to see whether there has been a change (c.f. polled I/O).

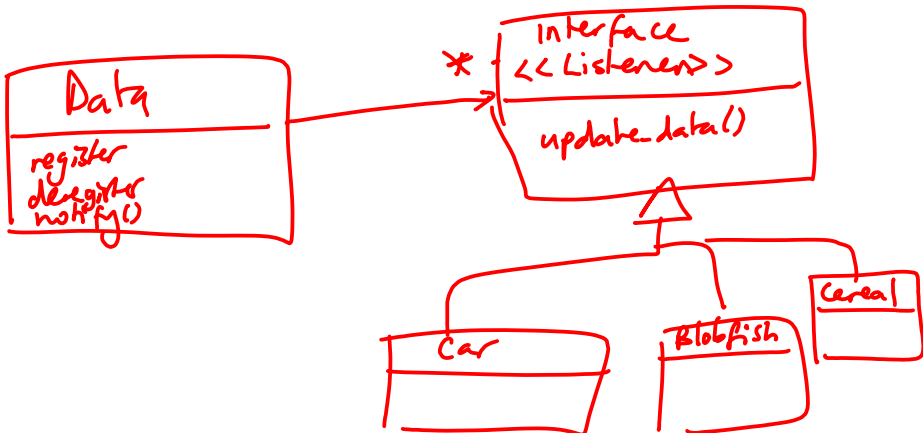
- ✗ How frequently should we poll? Too quickly and we might as well not have cached at all. Too slow and changes will be slow to propagate.

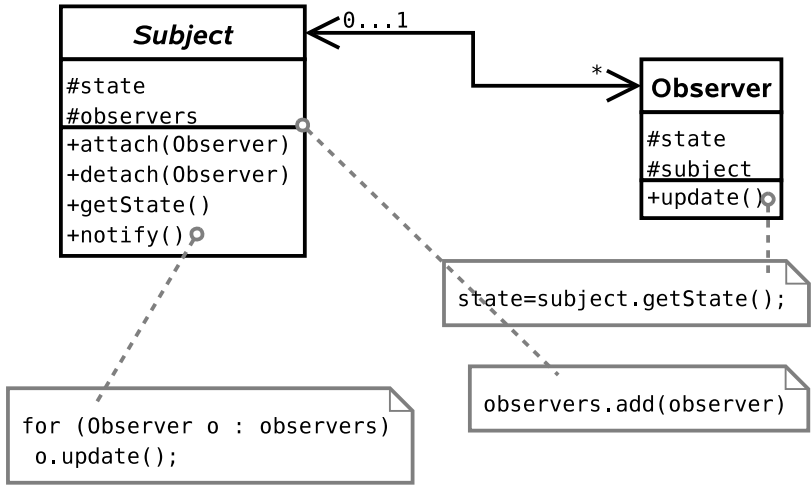
Solution 2: Modify the real object so that the proxy can 'register' with it (i.e. tell it of its existence and the data it is interested in). The proxy then provides a *callback* function that the real object can call when there are any changes.



4.10.1 Generalisation

This is the **Observer** pattern, also referred to as **Publish-Subscribe** when multiple machines are involved. It is useful when changes need to be propagated between objects and we don't want the objects to be tightly coupled. A real life example is a magazine subscription — you register to receive updates (magazine issues) and don't have to keep checking whether a new issue has come out yet. You unsubscribe as soon as you realise that 4GBP for 10 pages of content and 60 pages of advertising isn't good value.





Q23. Assume there is a machine somewhere on the internet that can supply the latest stock price for a given stock. The software it runs is written in Java and implements the interface:

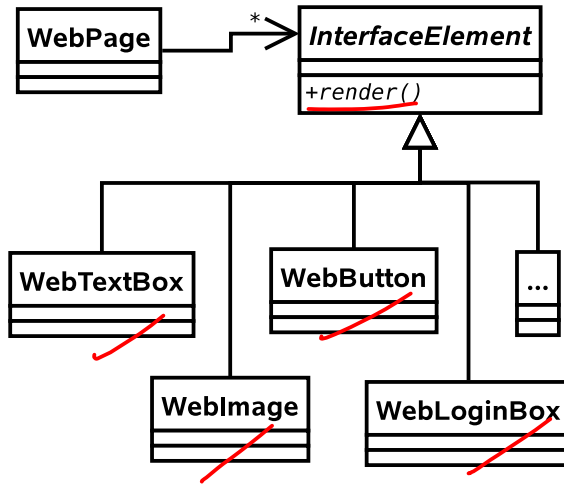
```
public interface StockReporter {  
    public double getStockPrice(String stockid);  
}
```

You are given a Java class `MyStockReporter` that implements this interface for you. When you use a `MyStockReporter` object, your request is automatically passed onto the real machine.

- (a) Identify the design pattern in use here
- (b) Why is this design inefficient?
- (c) Draw a UML class diagram to explain how the Observer pattern could improve efficiency. Give the changes to the interface that would be required.

4.11 Abstract Factory

Assume that the front-end part of our system (i.e. the web interface) is represented internally by a set of classes that represent various entities on a web page:



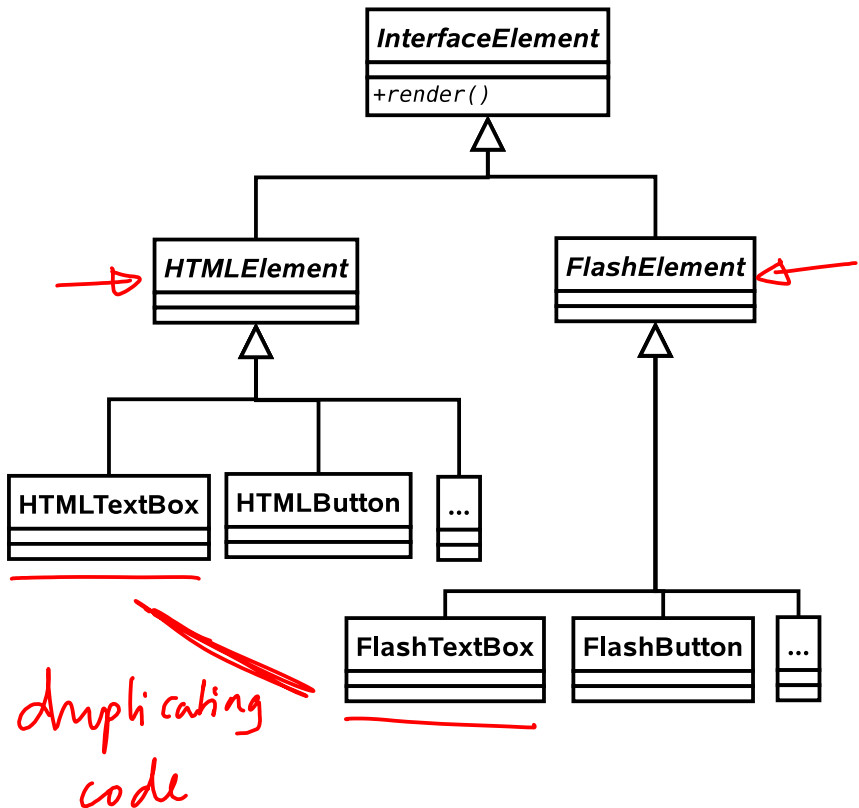
Let's assume that there is a `render()` method that generates some HTML which can then be sent on to web browsers.

Problem: Web technology moves fast. We want to use the latest browsers and plugins to get the best effects, but still have older browsers work. e.g. we might have a Flash site, a SilverLight site, a DHTML site, a low-bandwidth HTML site, etc. How do we handle this?

Solution 1: Store a variable ID in the `InterfaceElement` class, or use the **State** pattern on each of the subclasses.

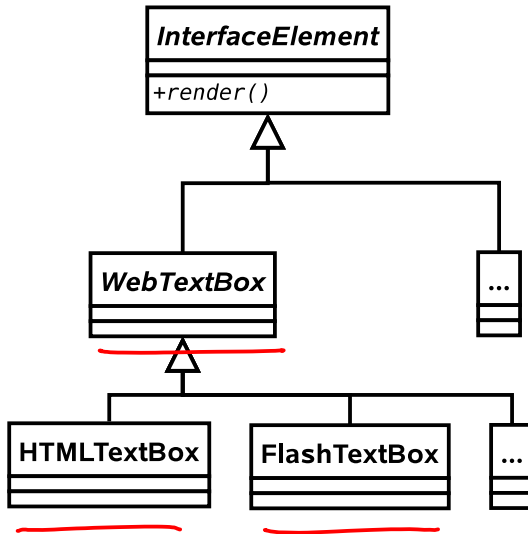
- ✓ Works.
- ✗ The **State** pattern is designed for a single object that regularly changes state. Here we have a family of objects in the same state (Flash, HTML, etc.) that we choose between at compile time.
- ✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

Solution 2: Create specialisations of `InterfaceElement`:



- ✗ Lots of code duplication.
- ✗ Nothing keeps the different `TextBoxes` in sync as far as the interface goes.
- ✗ A lot of work to add a new interface component type.
- ✗ Doesn't stop us from mixing `FlashButton` with `HTMLButton`, etc.

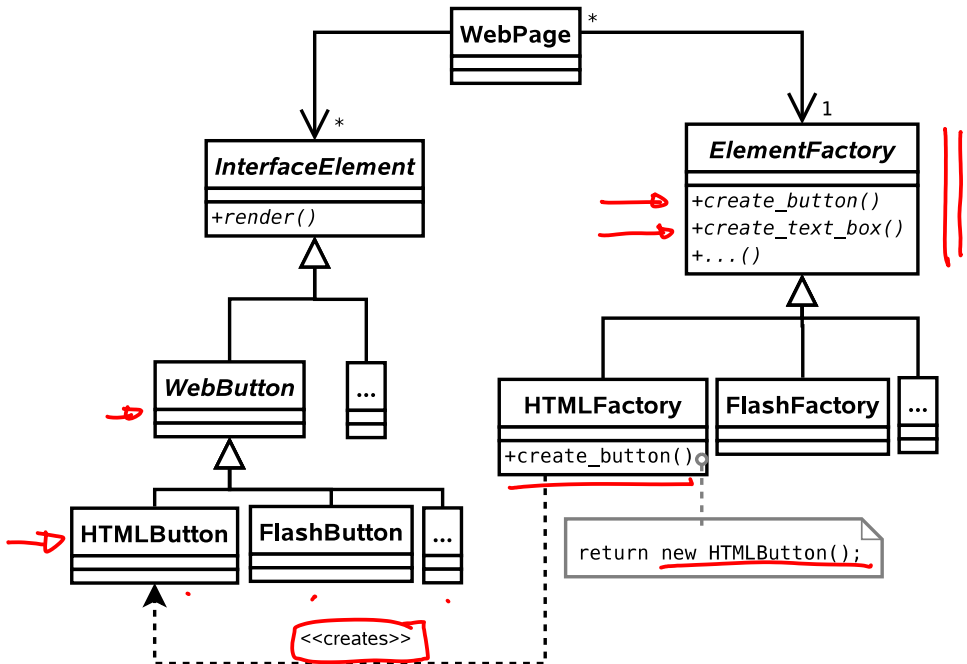
Solution 3: Create specialisations of each `InterfaceElement` subclass:



- ✓ Standardised interface to each element type.
- ✗ Still possible to inadvertently mix element types.

Solution 4: Apply the **Abstract Factory** pattern. Here we associate every `WebPage` with its own ‘factory’ — an object that is

there just to make other objects. The factory is specialised to one output type. i.e. a **FlashFactory** outputs a **FlashButton** when `create_button()` is called, whilst a **HTMLFactory** will return an **HTMLButton()** from the same method.



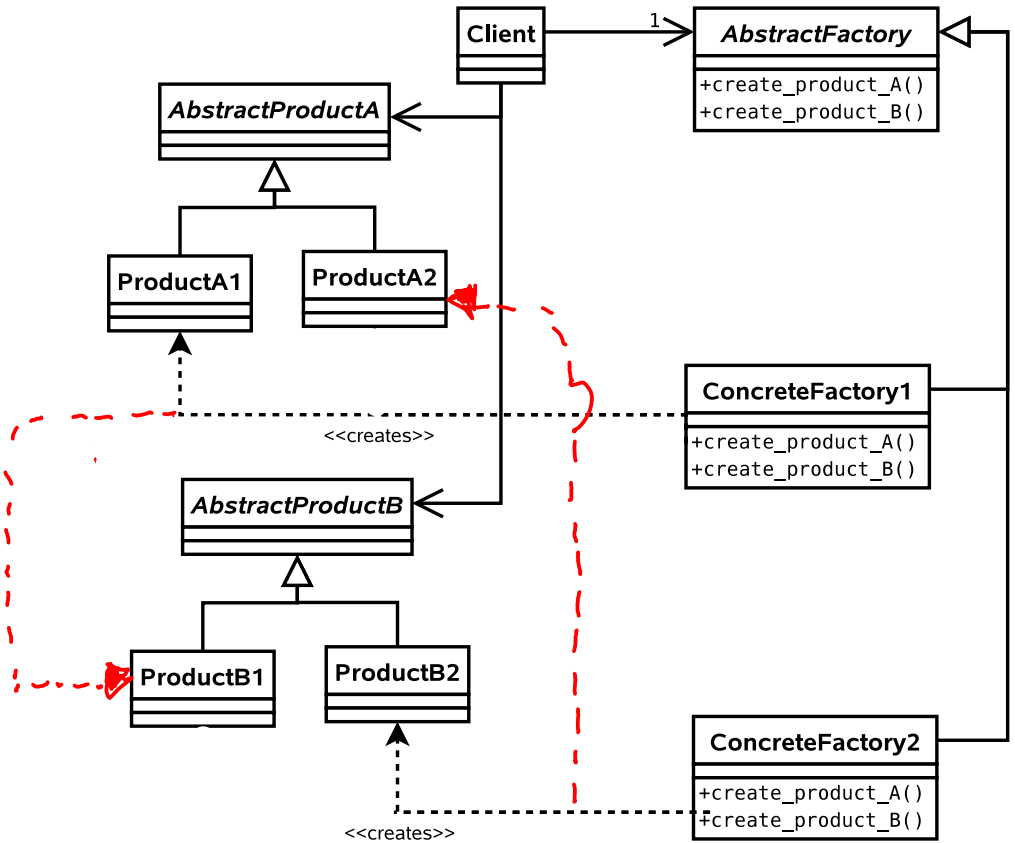
- ✓ Standardised interface to each element type.
- ✓ A given **WebPage** can only generate elements from a single family.
- ✓ Page is completely decoupled from the family so adding a new family of elements is simple.
- ✗ Adding a new element (e.g. **SearchBox**) is difficult.

✗ Still have to create a lot of classes.

4.11.1 Generalisation

This is the **Abstract Factory** pattern. It is used when a system must be configured with a specific family of products that must be used together.

Corrected slide



Note that usually there is no need to make more than one factory for a given family, so we can use the **Singleton** pattern to save memory and time.

Q24. Explain using diagrams how to the Abstract Factory pattern would help in writing an application that must support different languages (english, french, german, etc).

4.12 Summary

From the original Design Patterns book:

Decorator Attach additional responsibilities to an object dynamically. Decorators provide flexible alternatives to subclassing for extending functionality.

State Allow an object to alter its behaviour when its internal state changes.

Strategy Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Composite Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Singleton Ensure a class only has one instance, and provide a global point of access to it.

Proxy Provide a surrogate or placeholder for another object to control access to it.

Observer Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated accordingly.

Abstract Factory Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

4.12.1 Classifying Patterns

Often patterns are classified according to what their intent is or what they achieve. The original book defined three classes:

Creational Patterns . Patterns concerned with the creation of objects (e.g. **Singleton**, **Abstract Factory**).

Structural Patterns . Patterns concerned with the composition of classes or objects (e.g. **Composite**, **Decorator**, **Proxy**).

Behavioural Patterns . Patterns concerned with how classes or objects interact and distribute responsibility (e.g. **Observer**, **State**, **Strategy**).

4.12.2 Other Patterns

You've now met eight Design Patterns. There are plenty more (23 in the original book), but this course will not cover them. What has been presented here should be sufficient to:

- Demonstrate that object-oriented programming is powerful.
- Provide you with (the beginnings of) a vocabulary to describe your solutions.
- Make you look critically at your code and your software architectures.
- Entice you to read further to improve your programming.

Of course, you probably won't get it right first time (if there even is a 'right'). You'll probably end up *refactoring* your code as new situations arise. However, if a Design Pattern *is* appropriate, you should probably use it.

4.12.3 Performance

Note that all of the examples here have concentrated on structuring code to be more readable and maintainable, and to incorporate constraints structurally where possible. At no point have we discussed whether the solutions *perform* better. Many of the solutions exploit runtime polymorphic behaviour, for example, and that carries with it certain overheads.

This is another reason why you can't apply Design Patterns blindly. [This is a good thing since, if it wasn't true, programming wouldn't be interesting, and you wouldn't get jobs!].