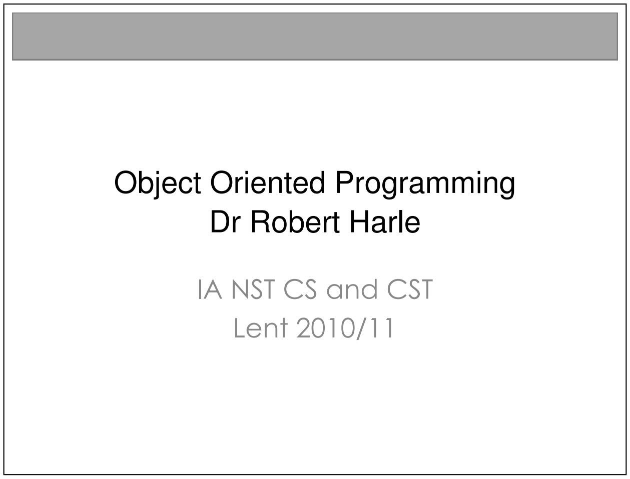
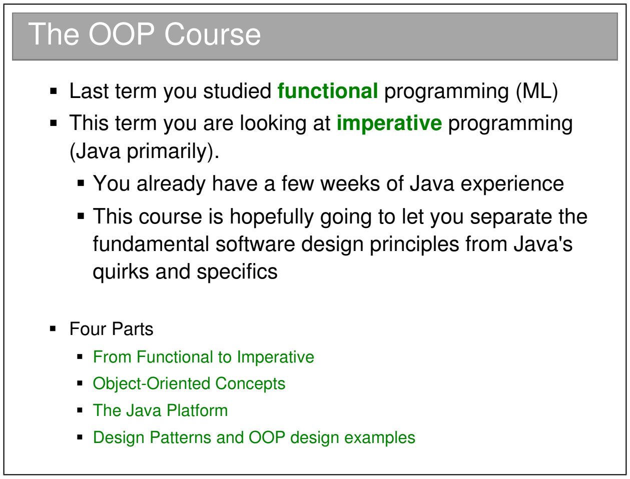


Object Oriented Programming
Dr Robert Harle

IA NST CS and CST
Lent 2010/11



The OOP Course

- Last term you studied **functional** programming (ML)
- This term you are looking at **imperative** programming (Java primarily).
  - You already have a few weeks of Java experience
  - This course is hopefully going to let you separate the fundamental software design principles from Java's quirks and specifics
- Four Parts
  - From Functional to Imperative
  - Object-Oriented Concepts
  - The Java Platform
  - Design Patterns and OOP design examples

Last term you learnt to program using the functional programming language ML. There are many reasons we started with this, chief amongst them being that everything is a well-formed *function*, by which we mean that an output is dependent solely on the inputs (arguments). This generally makes understanding easier. In fact, if you try any other functional programming languages you'll probably discover that its very similar to ML in many respects and translation is very easy.

However, if you have any experience of programming outside this course, you're probably aware that functional programming remains a niche choice. The dominant paradigm is that of *imperative* programming. Jumping from one imperative language to another is quite tricky if you haven't been able to distinguish the underlying programming concepts from the quirks of the specific language you know. So this course serves both to back up you java practicals and to make sure you've thought about the underlying paradigms that you're meeting.

To make specific points, I'll use a few imperative languages throughout the course. But predominantly we'll be using Java and *you won't be expected to program in anything else.*

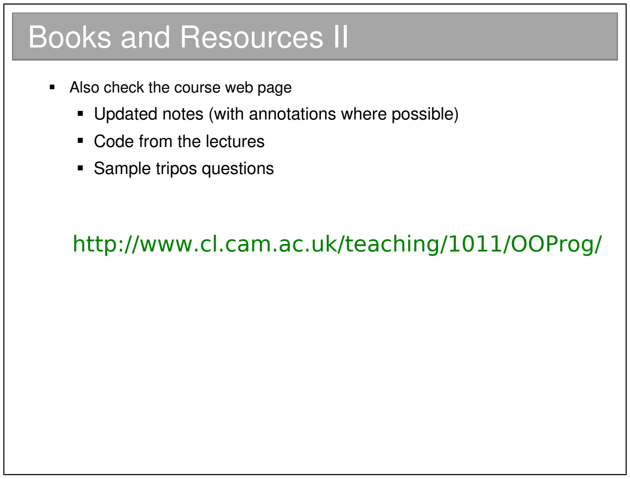
Java Practicals

- 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*!

* Some material may be repeated unintentionally. If so I will claim it was deliberate.

Books and Resources I

- 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 II

- 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/1011/OOProg/

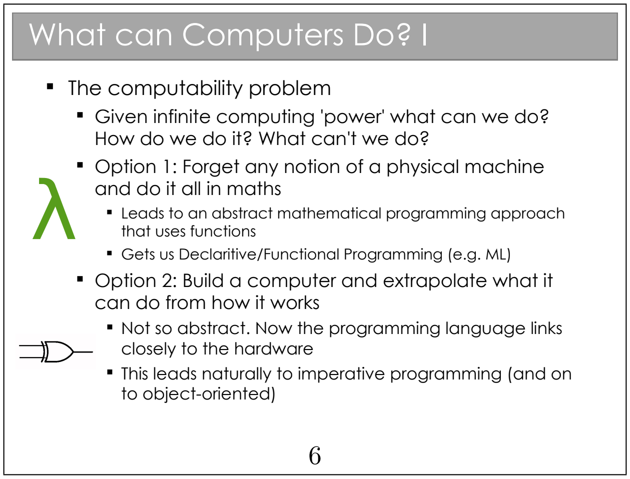
There is no shortage of books and websites describing the basics of object oriented programming. The concepts themselves are quite abstract, but most texts will use a specific language to demonstrate them. The books I've given favour Java (because that's the primary language you learn this term). You 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 aim *not* to assume anything from these lectures.

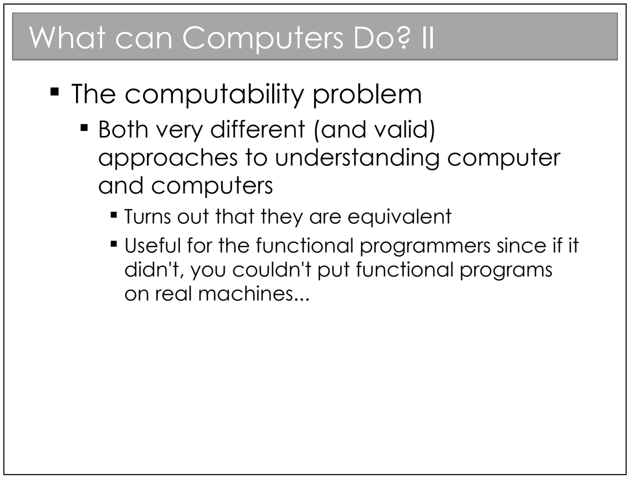
Chapter 1

From Functional to Imperative Programming



What can Computers Do? I

- 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 Declaritive/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? II

- 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...

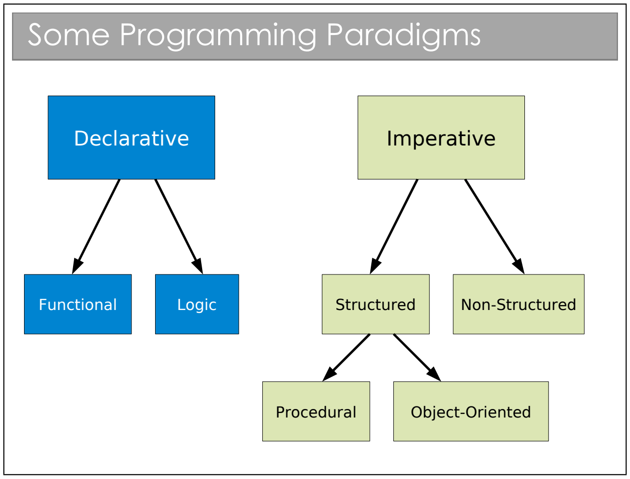
WWII spurred an interest in machinery that could compute. During the war, this interest was stoked by a need to break codes, but also to compute relatively mundane quantities such as the trajectory of an artillery shell. After the war, interest continued in the abstract notion of 'computability'. Brilliant 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.

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! All very nice, but why do you care??



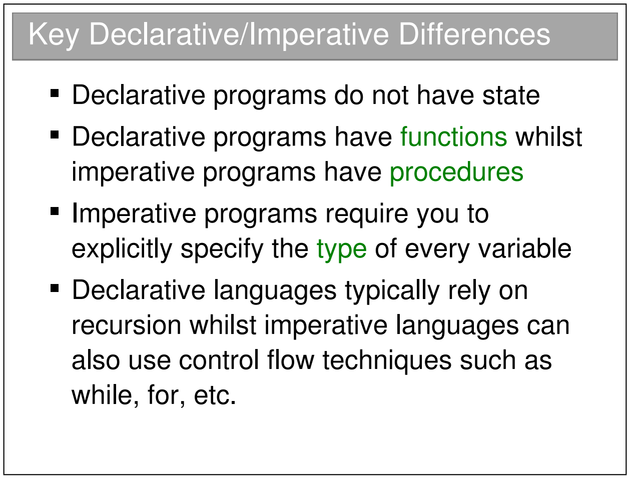
We can classify programming languages in many ways, but one of the most fundamental is as a *declarative* or *imperative* language. These roughly map to the two approaches just mentioned:

Declarative Languages. These allow the programmer to specify *what* should be done, and not *how* it should be done.

Imperative Languages. These specify exactly *how* something should be done

It should be clear to you that (most of) the ML you have done falls in the declarative camp. At no point were you specifying what was happening in memory and you have no idea how your values were represented at such a low level. Functional languages are therefore naturally declarative[1].

Imperative languages come rather naturally from abstraction of machine/assembly code. Assembly is not exactly human-readable (it's barely compsci-readable!). Imperative languages really just provide human-readable abstraction of assembly and as such they are naturally very close to the hardware in the sense you do low level manipulations.



Key Declarative/Imperative Differences

- Declarative programs do not have state
- Declarative programs have functions whilst imperative programs have procedures
- Imperative programs require you to explicitly specify the type of every variable
- Declarative languages typically rely on recursion whilst imperative languages can also use control flow techniques such as while, for, etc.

If you look back to your Foundations of CS notes, you will see there

[1]It turns out that pure functional languages can be a bit limiting, so many 'functional' languages are not actually pure functional. You've already seen this with ML references, which are very definitely imperative in style! When I talk about ML here I will almost always be ignoring the imperative hacks that have invaded the language.

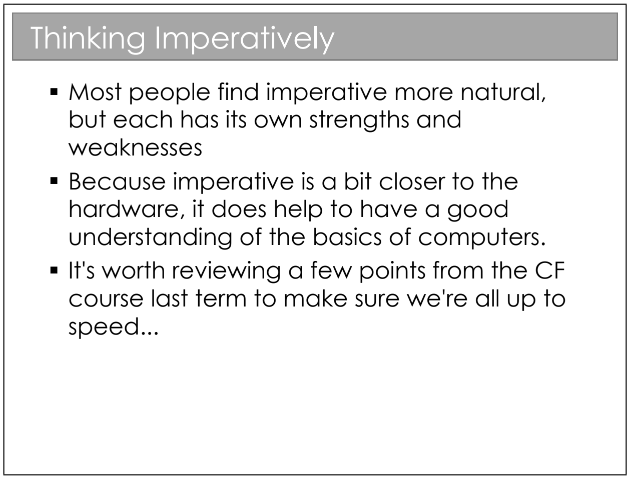
is a distinction drawn between functions and procedures. Strictly speaking, a *function* maps directly to the same notion in mathematics: its output is **solely** dependent on the supplied arguments and there can be no “side effects” of calling it.

In contrast, the output from a *procedure* can depend on program state that is not supplied in the arguments and it can also modify that external state. This is a “side effect” because, given only the procedure name and its arguments, we cannot predict what the state of the system will be after calling it without reading the full procedure definition.

Procedures are much more powerful, but as that awful line in Spiderman goes, “with great power comes great responsibility”. Now, that’s not to say that imperative programming makes you into some superhuman freak who runs around in his pyjamas climbing walls and battling the evil declaratives. It’s just that it introduces a layer of complexity into programming that makes the results better but the job harder.

Health warning: Most languages are imperative and many of them use the word ‘function’ as a synonym for ‘procedure’. To be honest, I bet a lot of programmers couldn’t tell you the difference between a function and a procedure so you will have to use your intelligence when you hear the word. Similarly, many people think of ‘procedural programming’ as a synonym for ‘imperative programming’.

1.1 Mapping Code to Hardware



Thinking Imperatively

- 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.
- It's worth reviewing a few points from the CF course last term to make sure we're all up to speed...

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. What's here is a really basic refresher of some salient points from your CF course:

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.

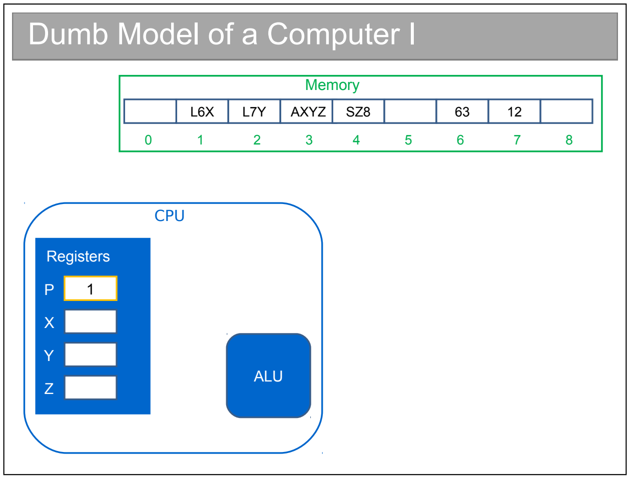
We use lots of different types of memory, but conceptually only two are of interest here. System memory is a very large pool of memory (the 2GB or so you get when you buy a machine). Then there are some really fast, but very small, chunks of memory called registers.

These are built into the CPU itself.

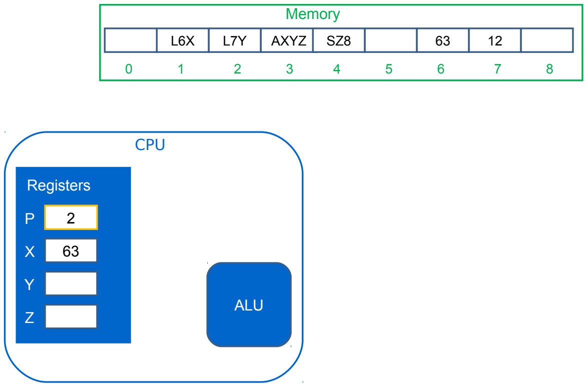
The CPU acts only on the chunks in the registers so the computer is constantly copying chunks of data from system memory into registers, operating on the registers and copying back any changes to system memory.

Let's recap the *fetch-execute cycle*. 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:

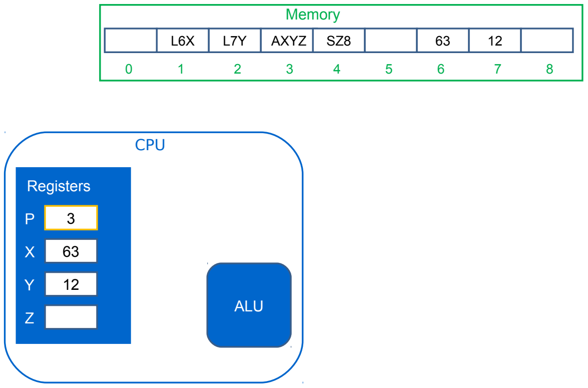
LAM. LOAD the value in memory slot A into register M
AMNO. ADD the values in registers M and N and put the result in register O.
SMA. STORE the value in register M into memory slot A

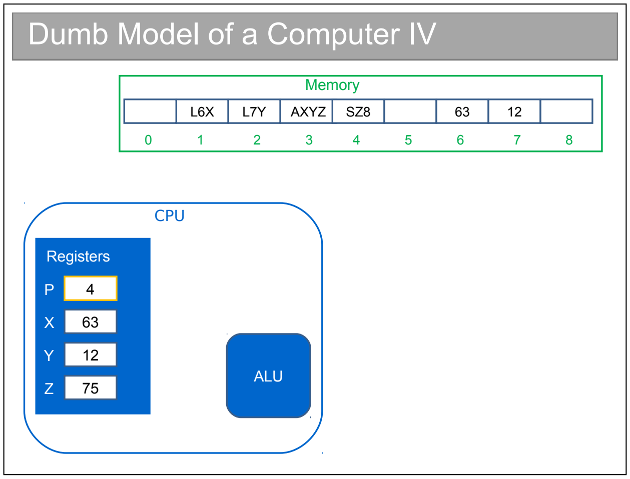


Dumb Model of a Computer II



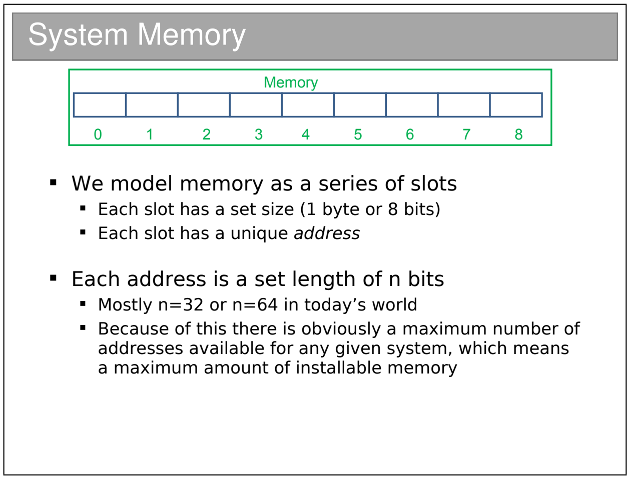
Dumb Model of a Computer III





Dumb Model of a Computer IV

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

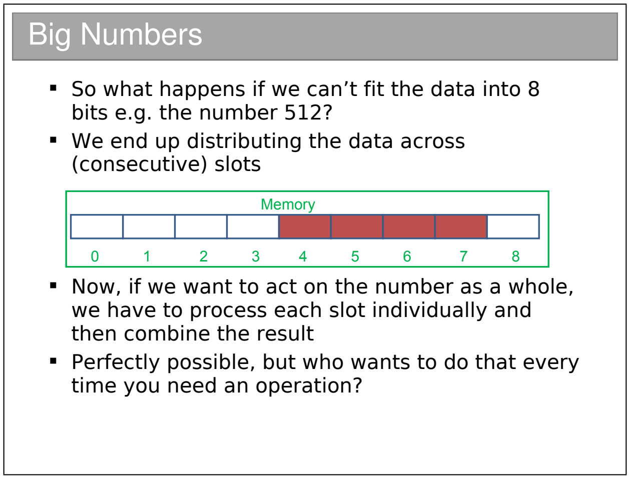


System Memory

Memory
0
1
2
3
4
5
6
7
8

- We model 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 are going to look at manipulating memory at a low level, so we had better have a model for how it works. We slice system memory up into 8-bit (1 byte) chunks and give each one an *address* (i.e. a slot number). Looking back oat our dumb computer model, we have to squeeze an address into one register (of n bits above), so we're immediately limited in the number of addresses available to us.

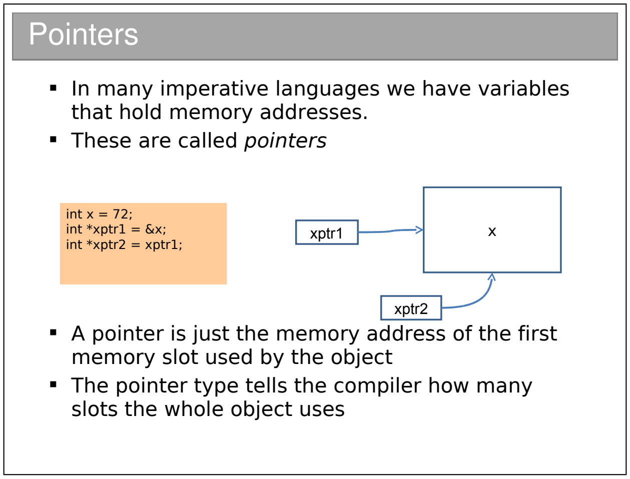


Big Numbers

- 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

Memory
0
1
2
3
4
5
6
7
8

- 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?



Pointers

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

```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```

xptr1
x
xptr2

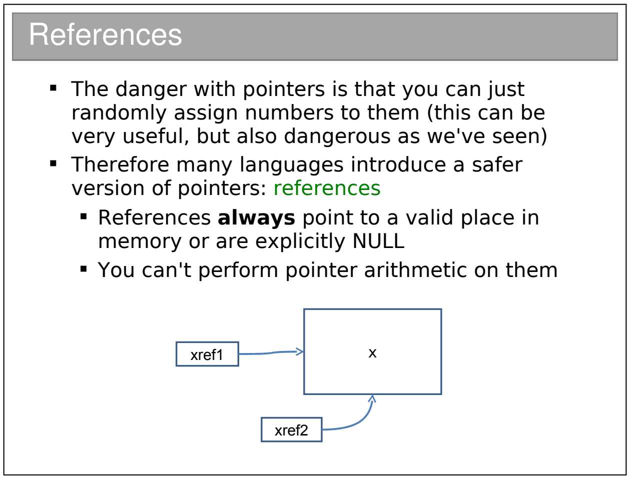
- A pointer is just the memory address of the first memory slot used by the object
- The pointer type tells the compiler how many slots the whole object uses

In your FoCS notes, there is a line that says "...creates references (also called pointers or locations)". Here, we need to be a bit more formal: there is a subtle but important difference between references and pointers.

The slide shows some C code. I know you haven't done any C, but since the Java syntax is based on C, it should look roughly familiar to you. In order, the lines do:

- Create a new variable of type `int` called `x` and set it to the value 72;
- Create a new `int` pointer (a variable that holds a memory address that leads to an `int` in memory) and set it to point to the variable `x`;
- Create another new `int` pointer and make it point to the same thing as the first.

In lectures I will show some examples of how the pointers may be used. The crucial point is that you can set a pointer value to anything you like. This can be both immensely useful and immensely dangerous (if you start reading at random locations, things can go very wrong indeed!).



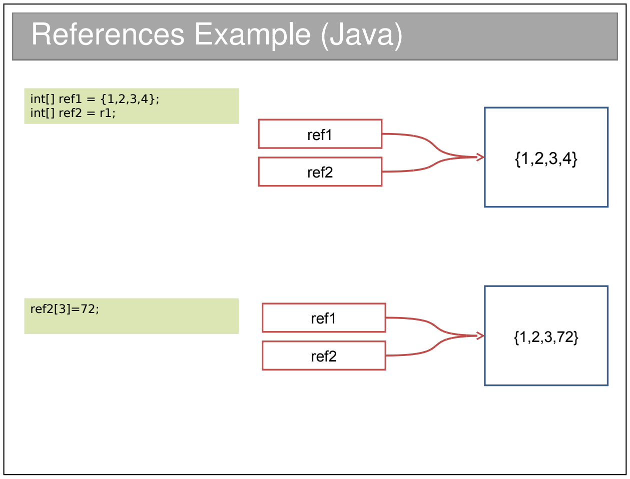
References

- The danger with pointers is that you can just randomly assign numbers to them (this can be very useful, but also dangerous as we've seen)
- Therefore many languages introduce a safer version of pointers: references
  - References **always** point to a valid place in memory or are explicitly NULL
  - You can't perform pointer arithmetic on them

xref1
x
xref2

One fix for pointers is the introduction of references to a language. As per the slide, a reference is a restricted pointer. It's still just a chunk of memory that contains a memory address, but the compiler (*not* the computer) will prevent us from doing certain operations on it to make things safer.

Sun decided that Java would have *only* references and no pointers. Whilst slightly limiting, this makes programming much safer (and it's one of the many reasons we teach with it).



References Example (Java)

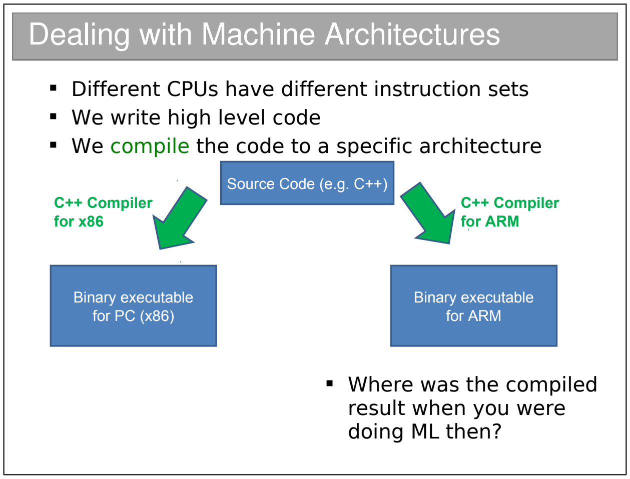
```
int[] ref1 = {1,2,3,4};
int[] ref2 = r1;
```

```
ref2[3]=72;
```

In your practicals you have found that arrays are handled by reference in Java. So in this code we create one array in memory, with four elements. ref1 is a reference to that array; we then set ref2 to have the same value i.e. point to the same memory address.

Thus, when we *dereference* ref2 and make a change, the change will also affect ref2. We will return to this shortly.

1.2 Aside: Dealing with Machine Architectures



Dealing with Machine Architectures

- Different CPUs have different instruction sets
- We write high level code
- We compile the code to a specific architecture

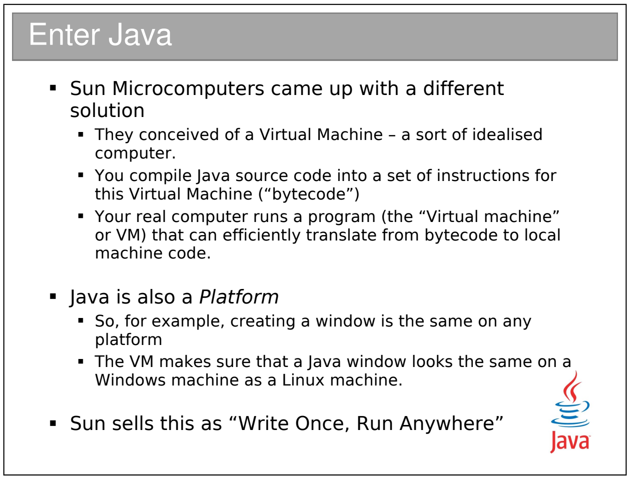
Source Code (e.g. C++)
C++ Compiler for x86
C++ Compiler for ARM
Binary executable for PC (x86)
Binary executable for ARM

- Where was the compiled result when you were doing ML then?

In Computer Fundamentals you learnt that each CPU has its own set of instructions: the so-called *instruction set*. These instructions do not make for easily readable or writable code and so we introduce the notion of layered machines. We write using high level languages such as C++, Java, python, ML, etc. We then need a compiler to convert these to something that makes sense to the CPU (a "binary executable").

The traditional approach is write source code; compile source to binary machine code; run binary program. This paradigm is "compile once, run many on one architecture". If you can guarantee enough machines understand the machine code you've used, you can distribute your software and hopefully make a profit!

An alternative approach is to compile "on-the-fly" i.e. as it is needed. If we do this we consider the compiler to be an *interpreter*. The advantage is the 'program' is then just the source code and it will run on any architecture that has a working interpreter. When you have programmed in ML, you have used an ML interpreter to run the code. In effect this is "write once, run anywhere" code. The downside is that there is an overhead associated with the translation and performance won't be as good as with compiled programs.



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"

Sun Microsystems invented Java as the web started to take off. Suddenly many different devices with many different machine architectures were communicating and they wanted to produce programs that could be run on any machine. They could have sent source code to an interpreter in the browser (a valid approach - it's what javascript does), but they i) wanted to get the best performance they could and ii) realised that there are times when you want to distribute binary files not source code.

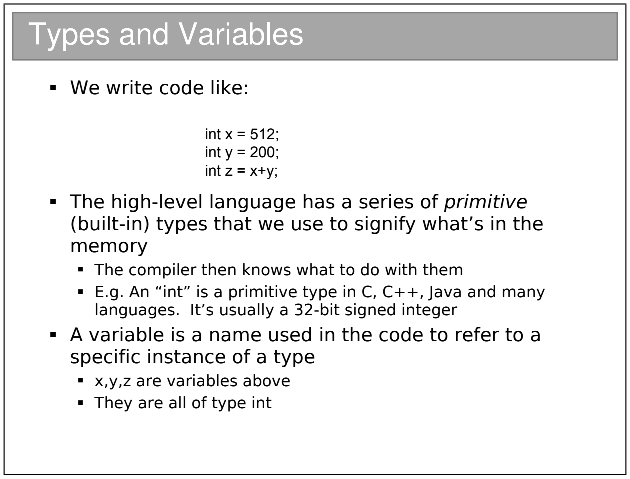
So Java is a bit of a half-way house. It compiles high-level source code into binary files that use a special instruction set called **bytecode**. You can think of this as being machine code for a virtual, generic CPU. Ironically there are now CPUs that use the bytecode instruction set but that wasn't really the intention.

So how do we use a bytecode file? The machine running the program must have a **Virtual Machine (VM)**), which acts as an interpreter for bytecode, translating it to the local CPU's instruction set on the fly. At first glance, this doesn't seem to be worth it—why not just use an interpreter directly? Well, high level languages are made for humans not CPUs; the compilation to bytecode does all of the hard work moving from something that is easy for a human to understand to something that is easy for a VM to understand. The VM is really just converting machine code to machine code. The end result is that the VM interpreter has much less work to do and therefore overall performance is increased when you run the program. As with an interpreter, this is "write once, run anywhere".

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/`

1.3 Handling Typed Variables


Types and Variables

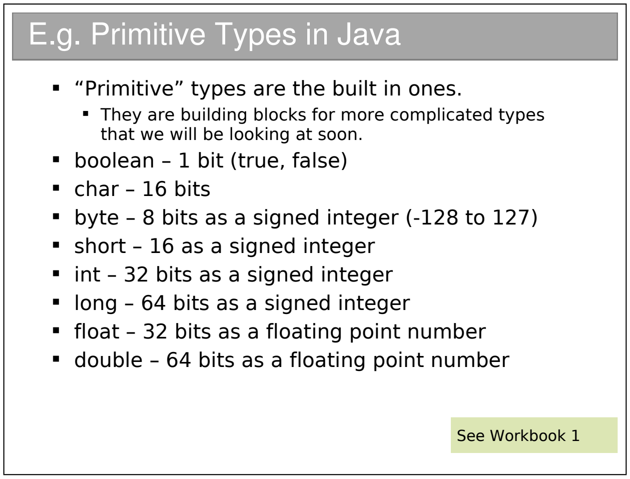
- We write code like:

```
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

By this stage you’ve no doubt had a few headaches dealing with types in ML. When you wrote ML functions you tried hard to avoid specifying the types: occasionally you had to but you knew that if you could keep it general then you could use polymorphism to avoid writing separate functions for integers, reals, etc.

In imperative languages, every variable has a type assigned when it is declared, and every function specifies the type of its output (it’s *return type*) and the types of its arguments. You’ve already seen the primitive (built-in) types available in ML and Java:



E.g. Primitive Types in Java

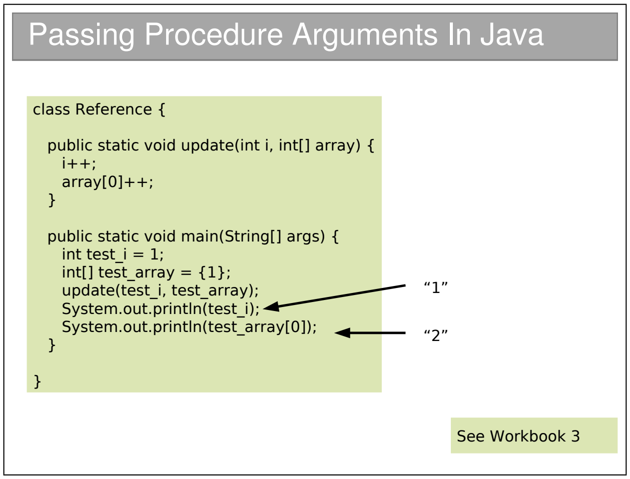
- “Primitive” types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 as a signed integer
- int – 32 bits as a signed integer
-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

See Workbook 1

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.

1.3.1 Passing Procedure Arguments



Passing Procedure Arguments In Java

```
class Reference {

  public static void update(int i, int[] array) {
    i++;
    array[0]++;
  }

  public static void main(String[] args) {
    int test_i = 1;
    int[] test_array = {1};
    update(test_i, test_array);
    System.out.println(test_i);
    System.out.println(test_array[0]);
  }

}
```

See Workbook 3

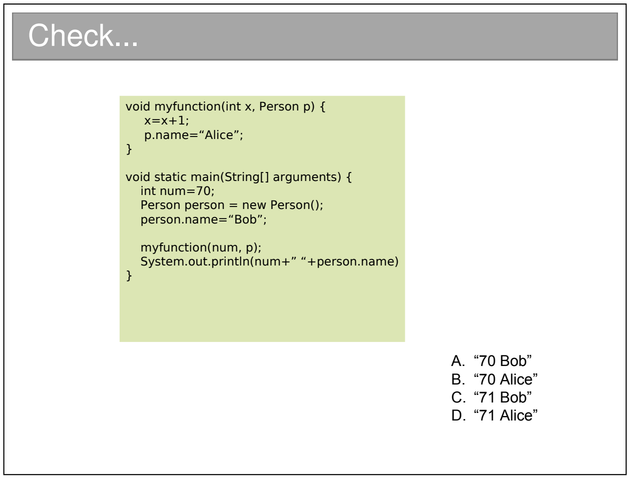
This example is taken from practical workbook 3, where you observed the different behaviour of test_i and test_array—the former being a primitive type and the latter being a reference to an array. We often say that primitive values are "passed by value" and arrays are "passed by reference", but I think this is rather confusing area for Java.

Let's create a model for what happens when we pass a primitive, say an int like test_i. The operating system allocates another int somewhere in memory (called i)and copies the value of the original (test_i=i). All subsequent operations in the procedure occur on the copy (test_i), which is then deleted at the end of the procedure.

Now let's look at what happens to the test_array variable. This is a reference to an array in memory. When passed as an argument, Java makes a copy of the reference (called array above). The reference obviously points to the same array since it contains the copied memory address. i.e. If we dereference the copy we end up at the same place in memory. All subsequent operations in the procedure occur on the copied reference, which is then deleted at the end of the procedure.

If you view the Java world this way, you can see that really passes all arguments by value, it's just that arguments are either primitives or references.

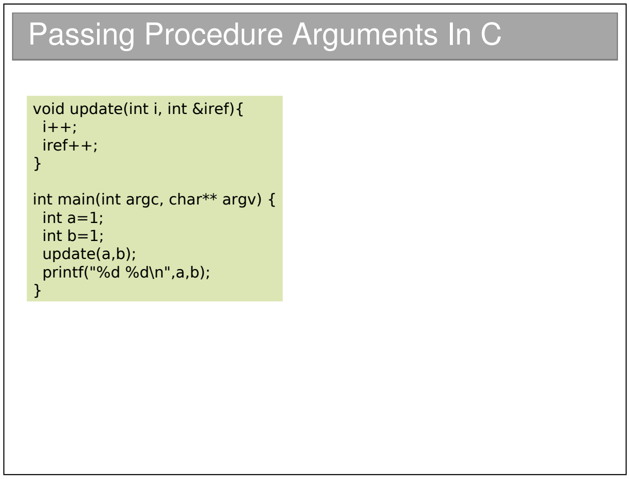
The confusion over this comes from the fact that many people view test_array to *be* the array and not a reference to it. If you think like that, then Java passes it by reference, as many books claim. The examples sheet has a question that explores this further.



Check...

```
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"



Passing Procedure Arguments In C

```
void update(int i, int &iref){
  i++;
  iref++;
}

int main(int argc, char** argv) {
  int a=1;
  int b=1;
  update(a,b);
  printf("%d %d\n",a,b);
}
```

Things are a bit clearer in other languages, such as C. They may allow you to specify how something is passed. In this C example, putting an ampersand ('&') in front of the argument tells the compiler to pass by reference and not by value.

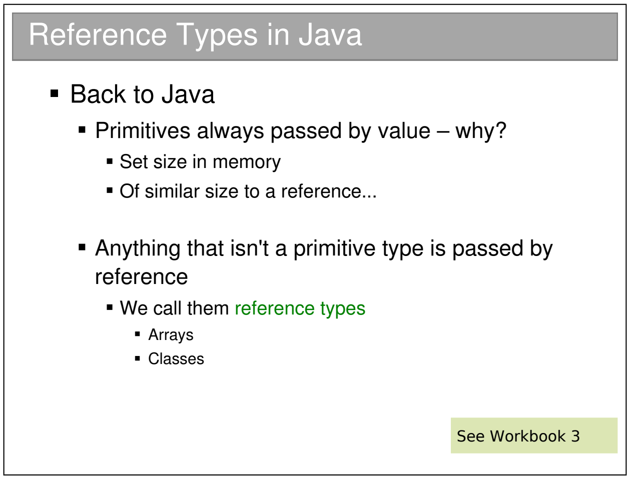
Having the ability to choose how you pass variables can be very powerful, but also problematic. Look at this code:

```
bool testA(HugeInt h) {
  if (h > 1000) return TRUE;
  else return FALSE;
}

bool testB(HugeInt &h) {
  if (h > 1000) return TRUE;
  else return FALSE;
}
```

Here I have made a fictional type HugeInt which is meant to represent something that takes a lot of space in memory. Calling either of these functions will give the same answer, but what happens at a low level is quite different. In the first, the variable is copied (lots of memory copying required—bad) and then destroyed (ditto). Whilst in the second, only a reference is created and destroyed, and that's quick and easy.

So, even though both pieces of code work fine, if you miss that you should pass by reference (just one tiny ampersand's difference) you incur a large overhead and slow your program.

I see this sort of mistake a *lot* in C++ programming and I guess the Java designers did too—they stripped out the need to specify pass by reference or value from Java!



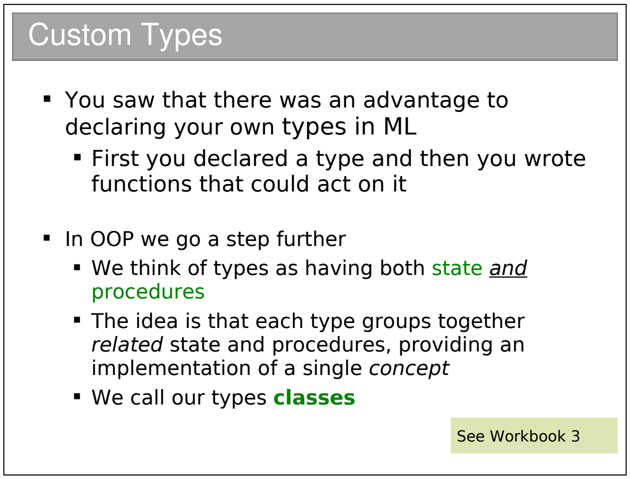
Reference Types in Java

- Back to Java
  - Primitives always passed by value – why?
    - Set size in memory
    - Of similar size to a reference...
  - Anything that isn't a primitive type is passed by reference
    - We call them reference types
      - Arrays
      - Classes

See Workbook 3

Chapter 2

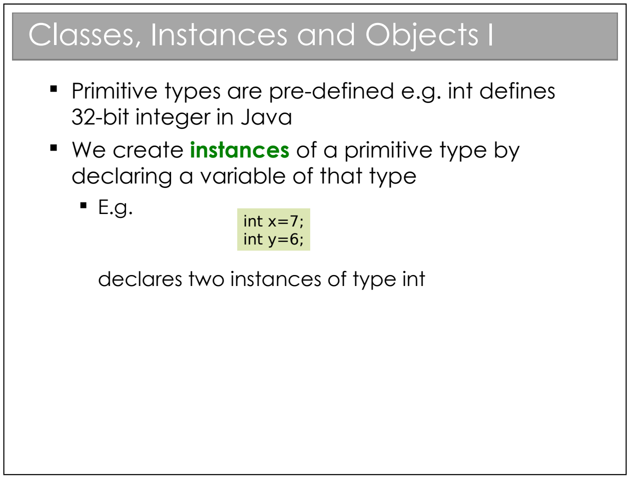
Object-Oriented Programming



Custom Types

- You saw that there was an advantage to declaring your own types in ML
  - First you declared a type and then you wrote functions that could act on it
- In OOP we go a step further
  - We think of types as having both state *and* procedures
  - The idea is that each type groups together *related* state and procedures, providing an implementation of a single *concept*
  - We call our types **classes**

See Workbook 3

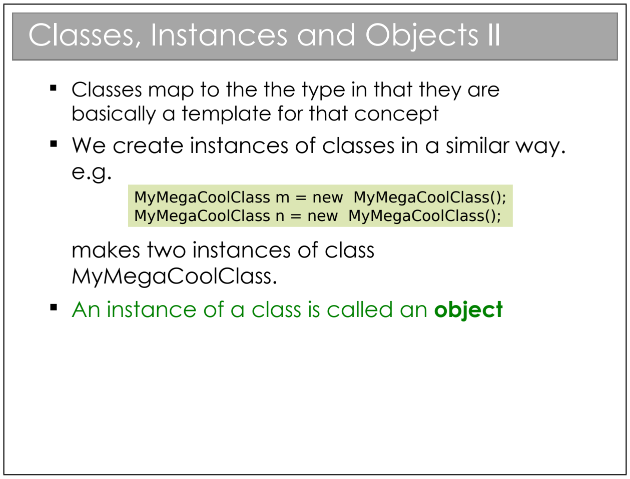


Classes, Instances and Objects I

- Primitive types are pre-defined e.g. int defines 32-bit integer in Java
- We create **instances** of a primitive type by declaring a variable of that type
  - E.g.

```
int x=7;
int y=6;
```

    declares two instances of type int



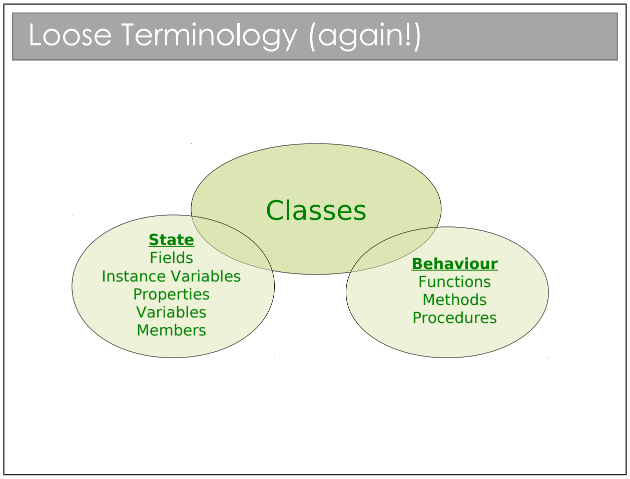
Classes, Instances and Objects II

- Classes map to the the type in that they are basically a template for that concept
- We create instances of classes in a similar way. e.g.

```
MyMegaCoolClass m = new  MyMegaCoolClass();
MyMegaCoolClass n = new  MyMegaCoolClass();
```

  makes two instances of class MyMegaCoolClass.
- An instance of a class is called an **object**

The difference between a class an an object is very simple, but you'd be surprised how much confusion it can cause for novice programmers. Classes define what properties and methods every instance of it should have, whilst each object is a specific implementation with set values. So a Person class might specify that a Person class has a name and an age. Each object of type Person has those properties set.



Loose Terminology (again!)

Now, you remember all that fuss we had about 'function' and 'procedure'? Well, it gets worse: when we're talking about a procedure inside a class, it's called a *method*.

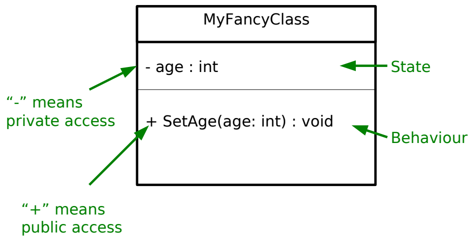
In the real world (which I'm assured does exist), you'll find people use 'function', 'procedure' and 'method' interchangeably. Thankfully you're all smart enough to cope!

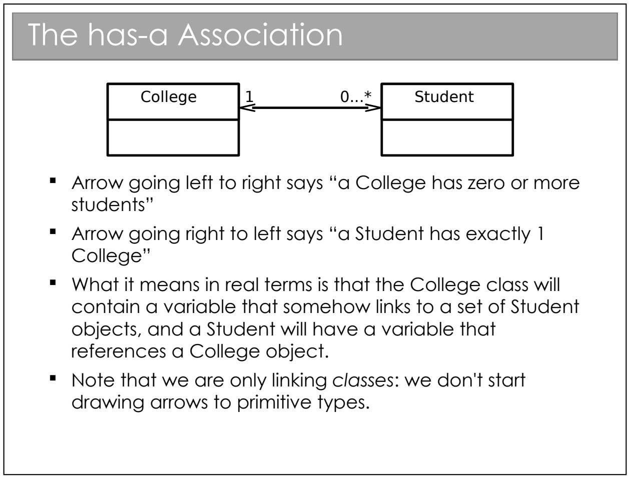
Identifying Classes

- 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

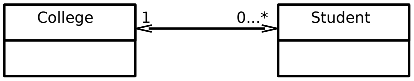
"Write a simulation of the Earth's orbit around the Sun"

Representing a Class Graphically (UML)





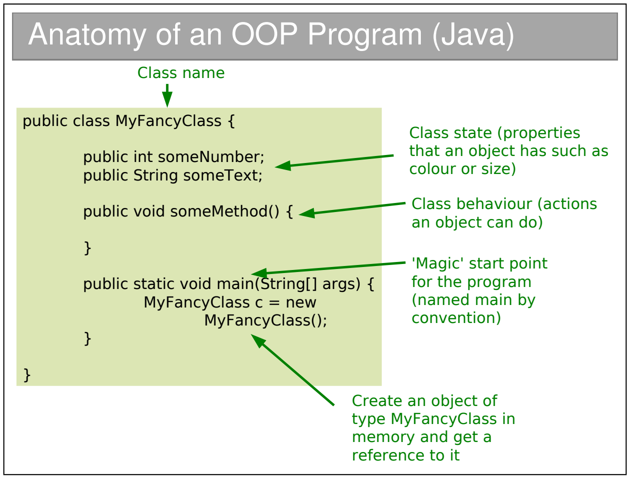
The has-a Association



- 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.

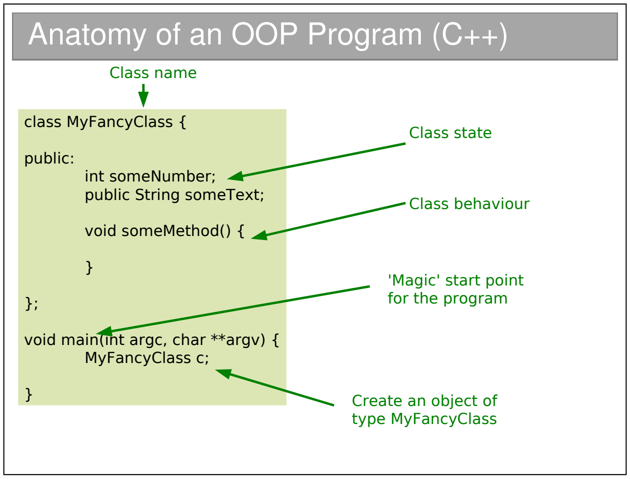


Anatomy of an OOP Program (Java)

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

        public static void main(String[] args) {
                MyFancyClass c = new
                        MyFancyClass();
        }

}
```

Class name
Class state (properties that an object has such as colour or size)
Class behaviour (actions an object can do)
'Magic' start point for the program (named main by convention)
Create an object of type MyFancyClass in memory and get a reference to it

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.



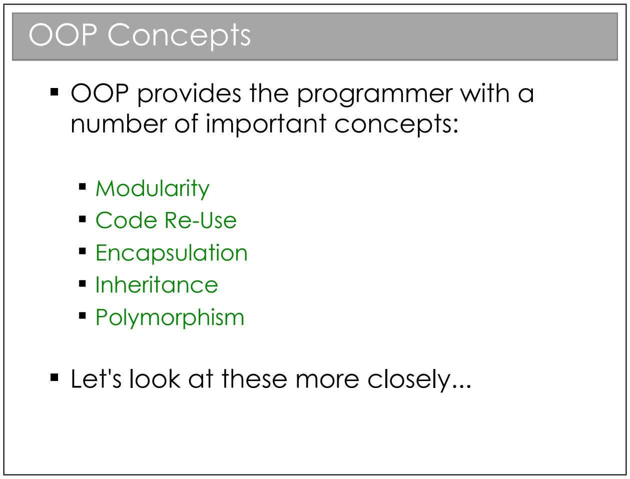
Anatomy of an OOP Program (C++)

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

void main(int argc, char **argv) {
        MyFancyClass c;

}
```

Class name
Class state
Class behaviour
'Magic' start point for the program
Create an object of type MyFancyClass

This is here just so you can compare. The Java syntax is based on C/C++ so it's no surprise that there are 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).

2.1 OOP Concepts

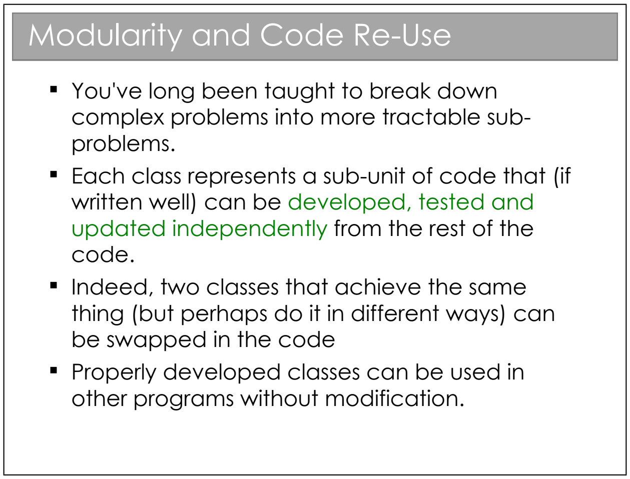


OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance
  - Polymorphism
- Let's look at these more closely...

Let's be clear here: OOP doesn't *enforce* the correct usage of the ideas we're about to look at. Nor are the ideas exclusively found in OOP languages. The main point is that OOP *encourages* the use of these concepts, which is generally good for software design.

2.1.1 Modularity and Code Re-Use



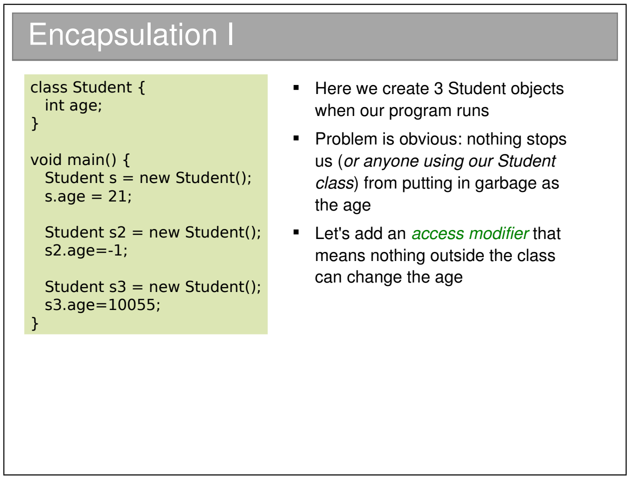
Modularity and Code Re-Use

- You've long been taught to break down complex problems into more tractable sub-problems.
- Each class represents a sub-unit of code that (if written well) can be developed, tested and updated independently from the rest of the code.
- Indeed, two classes that achieve the same thing (but perhaps do it in different ways) can be swapped in the code
- Properly developed classes can be used in other programs without modification.

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. By identifying objects in our problem, we can write classes that represent them. Each class can be developed, tested and maintained independently of the others (assuming we've done a good job).

There is a further advantage to breaking a program down into self-contained objects: those objects can be ripped from the code and put into other programs. So, once you've developed and tested a class that represents a Student, say, you can use it in lots of other programs with minimal effort. Even better, the classes can be distributed to other programmers so they don't have to reinvent the wheel. OOP therefore strongly encourages software *re-use*.

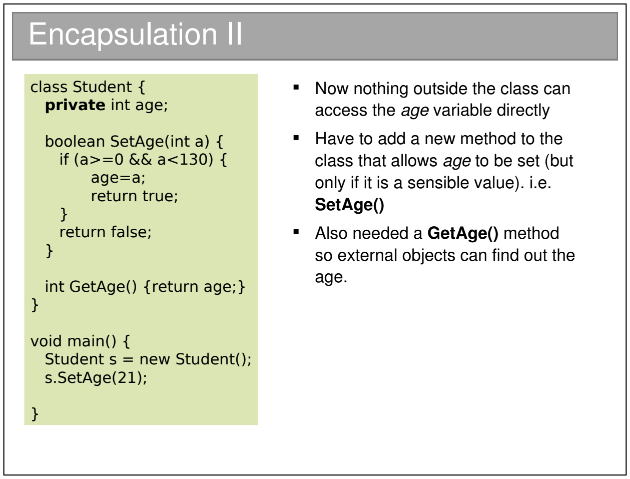
2.1.2 Encapsulation



Encapsulation I

```
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;
}
```

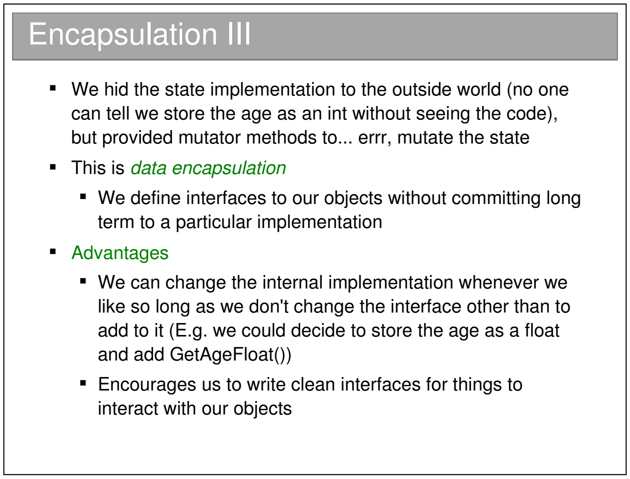
- Here we create 3 Student objects when our program runs
- Problem is obvious: nothing stops us (*or anyone using our Student class*) from putting in garbage as the age
- Let's add an *access modifier* that means nothing outside the class can change the age



Encapsulation II

```
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). i.e. **SetAge()**
- Also needed a **GetAge()** method so external objects can find out the age.



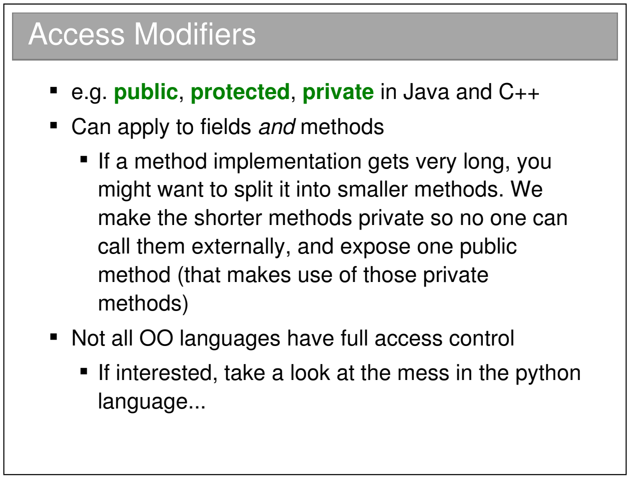
Encapsulation III

- 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* or (as some pedants prefer) *implementation 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 to A is likely to require a change 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! 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**.

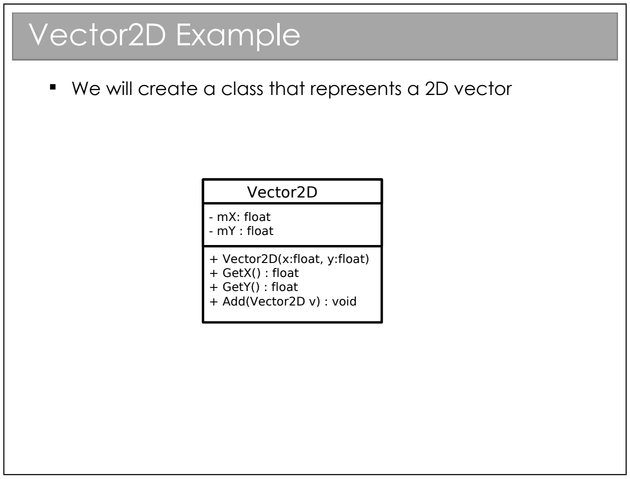


Access Modifiers

- e.g. **public**, **protected**, **private** in Java and C++
- Can apply to fields *and* methods
  - If a method implementation gets very long, you might want to split it into smaller methods. We make the shorter methods private so no one can call them externally, and expose one public method (that makes use of those private methods)
- Not all OO languages have full access control
  - If interested, take a look at the mess in the python language...

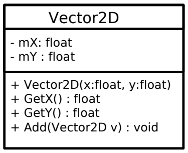
At this stage you should be comfortable with public and private fields of a class. The former allows code outside of the object to read it and alter it; whilst the latter prevents any access to the field from outside the class. protected we will come to shortly....

Java actually throws in one more: package. You've met Java packages in your practicals and now it a way to group classes together. If a field has package access then any code within that package can access it; all other code cannot. So package is to packages what private is to classes. Interestingly, the default access modifier (i.e. the one adopted if you don't specify the access modifier when you declare your field) is package and *not* public as is often thought.



Vector2D Example

- We will create a class that represents a 2D vector



One of the examples we will develop in lectures is a representation of a two dimensional vector (x, y). This class will require a constructor, which you encountered in Workbook 3. We'll talk more about them later.

The class we create is obviously very simple, 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 functional 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.

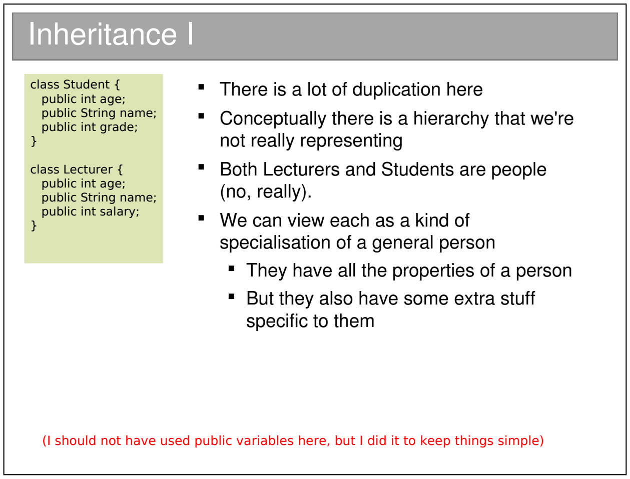
Some advantages of immutability:

- Simpler to contruct, 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."

2.2 Inheritance

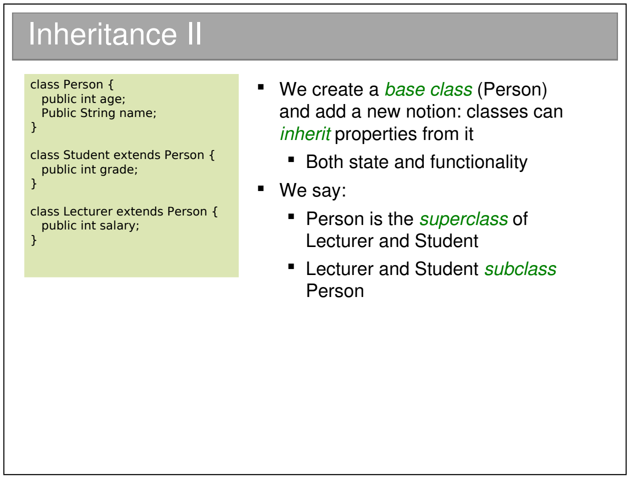


Inheritance I

```
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

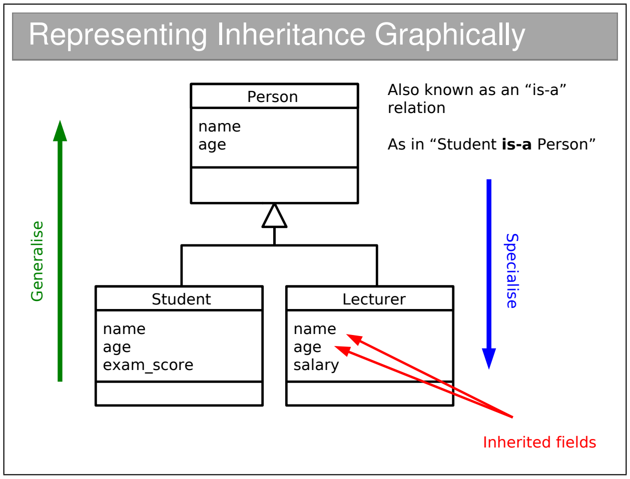
(I should not have used public variables here, but I did it to keep things simple)



Inheritance II

```
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



Representing Inheritance Graphically

Also known as an "is-a" relation

As in "Student **is-a** Person"

Inheritance is an extremely powerful concept that is used extensively in good OOP. We has discussed the "has-a" relation amongst classes;

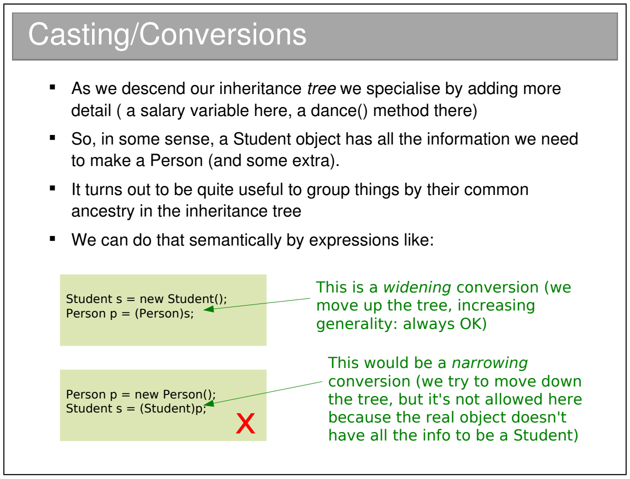
inheritance adds the “is-a” concept. E.g. A car *is a* vehicle that *has a* steering wheel.

We speak of an inheritance *tree* where moving down the tree makes things more specific and up the tree more general.

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

Many students seem to confuse “is-a” and “has-a” arrows in their UML class diagrams: please make sure you don’t!



Casting/Conversions

- 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

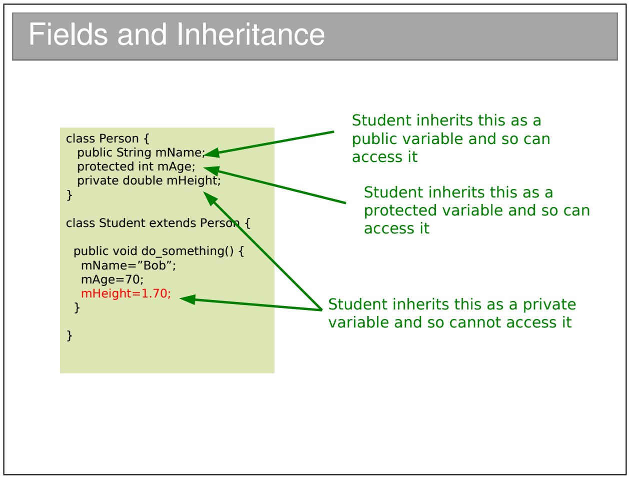
This would be a *narrowing* conversion (we try to move down the tree, but it's not allowed here because the real object doesn't have all the info to be a Student)

One way to think about this is that when we create a new Student the computer allocates a chunk of memory for that object. In that chunk there is space for all of the fields (name, age, exam score, etc) and definitions of the methods it supports. Because a Student is a Person, that chunk will contain everything required for a Person (name, age) and some extras specific to being a Student (exam score).

Now, we work with references to objects (the memory chunks) in Java. The type of the reference tells the computer what to expect when it follows it. If we have a Person reference that points to an object that's really a Student, that's fine—everything it needs for a Person is in the object it finds (plus some extra 'stuff').

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

2.2.1 Inheritance and State



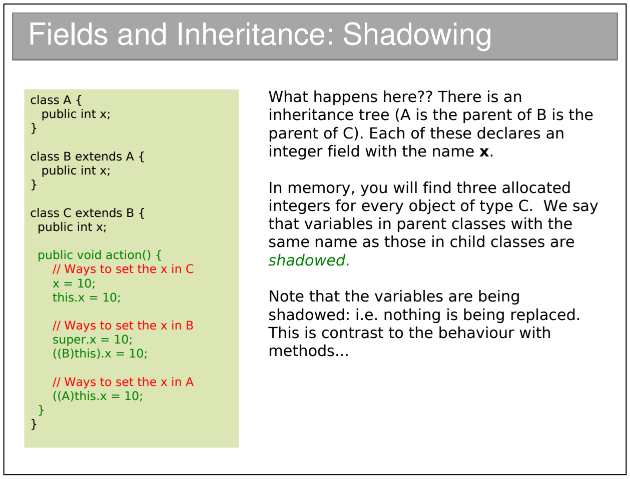
Fields and Inheritance

```
class Person {
   public String mName;
   protected int mAge;
   private double mHeight;
}

class Student extends Person {

  public void do_something() {
    mName="Bob";
    mAge=70;
    mHeight=1.70;
  }

}
```

Student inherits this as a public variable and so can access it
Student inherits this as a protected variable and so can access it
Student inherits this as a private variable and so cannot access it

You will see that the protected access modifier can now be explained. A protected variable is exposed for read and write within a class, and *within all subclasses of that class*. Code outside the class or its subclasses can't touch it directly[1].

[1] At least, that's how it in most languages. Java actually allows any class in the same Java package to access protected variables.



Fields and Inheritance: Shadowing

```
class A {
   public int x;
}

class B extends A {
   public int x;
}

class C extends B {
  public int x;

  public void action() {
     // Ways to set the x in C
     x = 10;
     this.x = 10;

     // Ways to set the x in B
     super.x = 10;
     ((B)this).x = 10;

     // Ways to set the x in A
     ((A)this.x = 10;
  }
}
```

What happens here?? There is an inheritance tree (A is the parent of B is the parent of C). Each of these declares an integer field with the name **x**.

In memory, you will find three allocated integers for every object of type C. We say that variables in parent classes with the same name as those in child classes are *shadowed*.

Note that the variables are being shadowed: i.e. nothing is being replaced. This is contrast to the behaviour with methods...

A common novice OOP error is to assume that we have to redeclare a field in its subclasses for it to be inherited: not so. *Every* field is inherited by a subclass.

There are two new keywords that have appeared here: super and this. The this keyword can be used in any class method[2] and provides us with a reference to the current object. In fact, the this keyword is what you need to access anything within a class, but because we'd end up writing this all over the place, it is taken as implicit. So, for example:

```
public class A {
  private int x;
  public void go() {
    this.x=20;
  }
```

[2]By this I mean it cannot be used outside of a class, such as within a static method: see later for an explanation of these.

```
}
```

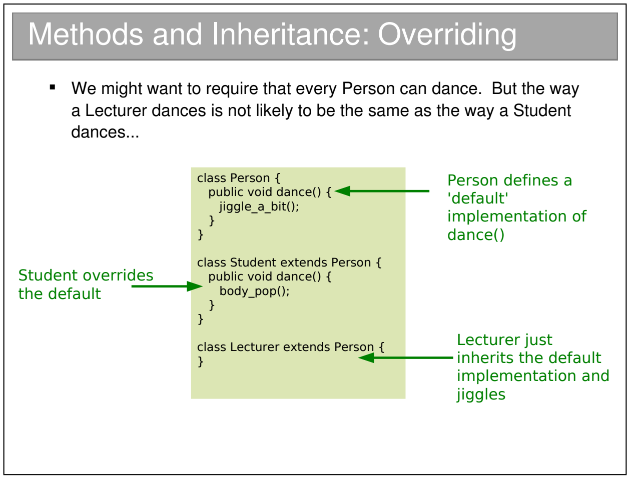
becomes:

```
public class A {
  private int x;
  public void go() {
    x=20;
  }
}
```

The super keyword gives us access to the direct parent (one step up in the tree). You've met both in your Java practicals.

2.2.2 Inheriting Methods and Polymorphism

It's all very well inheriting fields, but what happens to all of the methods?



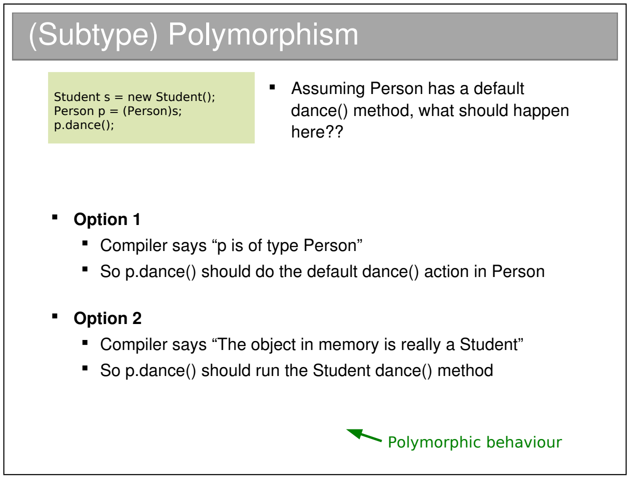
Methods and Inheritance: Overriding

- 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...

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

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

class Lecturer extends Person {
}
```

Person defines a 'default' implementation of dance()
Student overrides the default
Lecturer just inherits the default implementation and jiggles

Every object that has Person for a parent must have a dance() method since it is defined in the Person class and is inherited. The situation so far is directly analogous to what happens with fields.



(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

Option 1 would match our expectations from shadowing: the idea that the object contains two dance() methods, and we can choose between them. Option 2 (called polymorphic) effectively has the parent method replaced by the child method. It turns out that option 2 is immensely useful and the Java designers decided to make it the only behaviour.

Interestingly, not all OO languages choose between the options. C++, for example, allows *you* to choose:

```
#include <iostream>

class A {
public:

  void printNormal() {
    std::cout << "A" << std::endl;
  }
  virtual void printVirtual() {
    std::cout << "A" << std::endl;
  }
};


class B : public A {
public:

  void printNormal() {
    std::cout << "B" << std::endl;
  }

  virtual void printVirtual() {
    std::cout << "B" << std::endl;
  }
};

int main(int argc, char ** argv) {
  B bobj; // Declare an object of type B.

  // Set up pointers in C++
  B *bptr = &bobj;
  A *aptr = (A*)bptr; // Cast

  bptr->printNormal();
  aptr->printNormal();

  bptr->printVirtual();
  aptr->printVirtual();
}
```

Which results in:

```
B
A
B
B
```

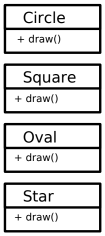
i.e. you need to add a special tag (virtual) to the methods that you want to behave polymorphically in C++.

Aside: You have of course met the word 'polymorphism' in your Foundations course.[3] There it was used to mean that you could avoid specifying the type in your code and the compiler sorted it out when it came to compile it (or interpret it if you prefer). This is more properly called *parametric* or *static* polymorphism, where the compiler figures out the type *before* the program runs, at compile-time.

Here it is slightly different. At compile time the compiler is told a type for the reference (e.g. Person) but the true type of the object may be anything derived from that type (e.g. Student, Lecturer). The ambiguity can only be resolved at run-time, when the computer can look at what's in memory and figure out the real type. This is more properly called *ad-hoc*, *dynamic*, or even *subtype* polymorphism.

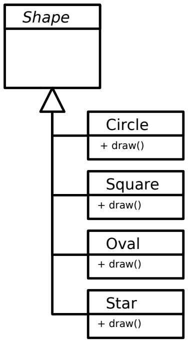
[3]The etymology of the word polymorphism is from the ancient Greek: *poly* (many)–*morph* (form)–ism

The Canonical Example I



- 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?

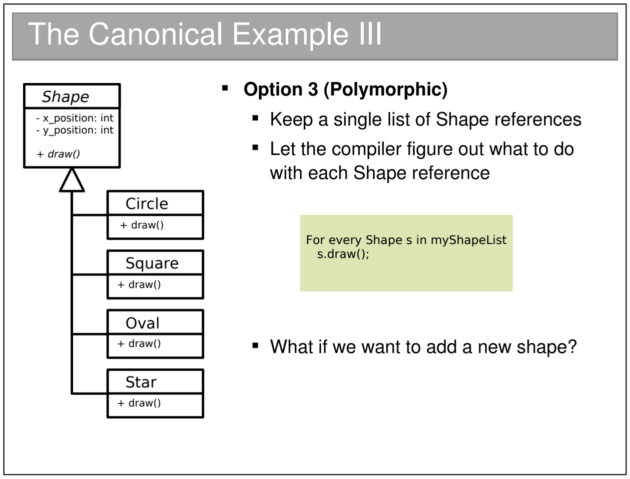
The Canonical Example II



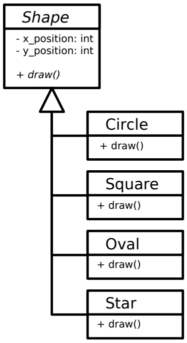
- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
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?



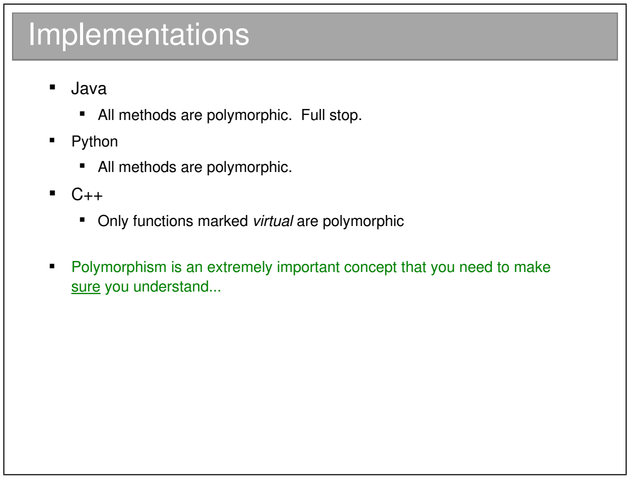
The Canonical Example III



- **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?



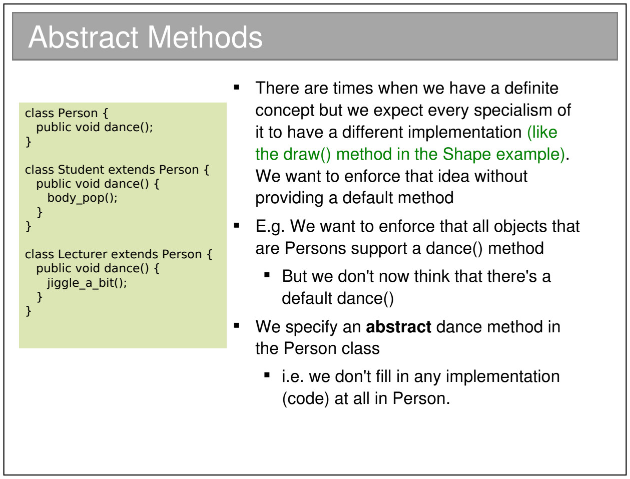
Implementations

- Java
  - All methods are polymorphic. Full stop.
- Python
  - All methods are polymorphic.
- C++
  - Only functions marked *virtual* are polymorphic

- Polymorphism is an extremely important concept that you need to make sure you understand...

What happens when you run a polymorphic function is that the system must dynamically decide which version to run. As you might expect, this isn't instantaneous and so there is an overhead associated with using polymorphic functions. This can be an advantage of languages like C++ (which give you the choice), but more and more languages seem to favour simplicity over power, and this choice is one of the first things to go!

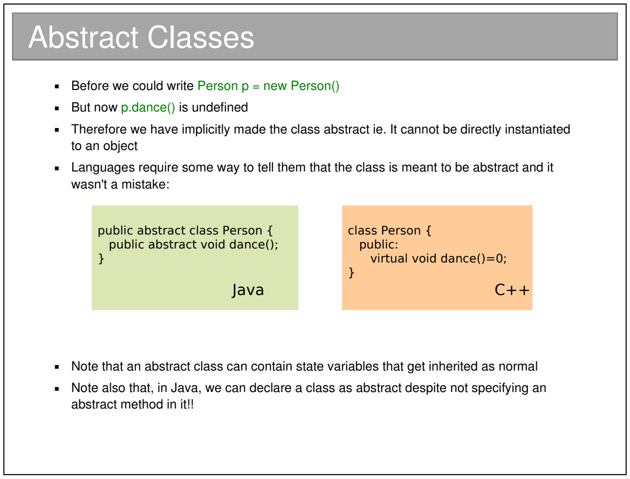
2.3 Abstract Classes


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 (like the draw() method in the Shape example). We want to enforce that idea without providing a default method
- 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.

An abstract method can be thought of as a contractual obligation: any non-abstract class that inherits from this class *will* have that method implemented.



Abstract Classes

- Before we could write Person p = new Person()
- But now p.dance() is undefined
- Therefore we have implicitly made the class abstract ie. 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
```

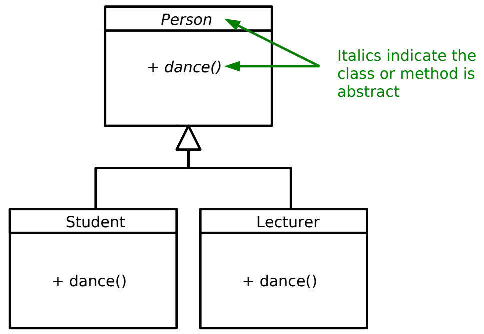
```
class Person {
   public:
      virtual void dance()=0;
}
                         C++
```

- 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!!

Abstract classes allow us to partially define a type. Because it's not fully defined, you can't make an object from an abstract class (try it). Only once all of the 'blanks' have been filled in can we create an object from it. This is particularly useful when we want to represent high level concepts that do not exist in isolation.

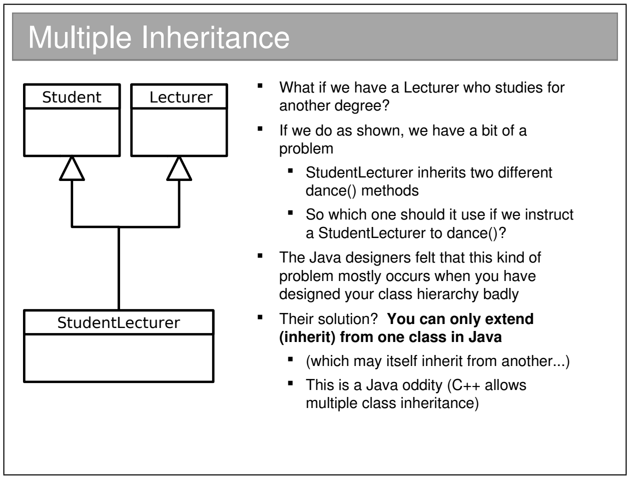
Depending on who you're talking to, you'll find different terminology for the initial declaration of the abstract function (e.g. the `public abstract void dance()` bit). Common terms include *method prototype* and *method stub*.

Representing Abstract Classes



You have to look at UML diagrams carefully since the italics that represent abstract methods or classes aren't always obvious on a quick glance.

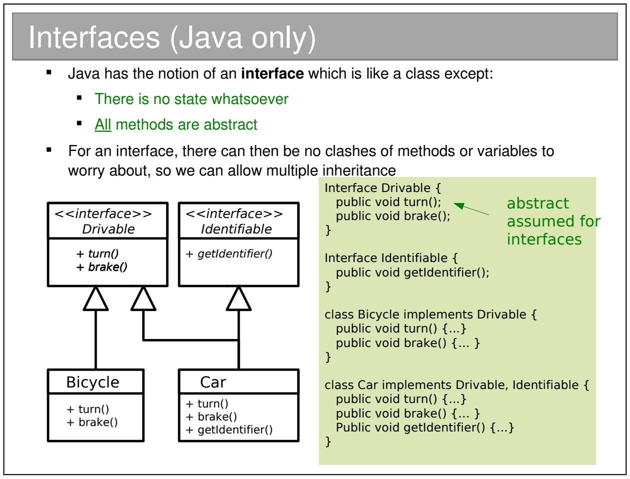
2.4 Interfaces


Student
Lecturer
StudentLecturer

Multiple Inheritance

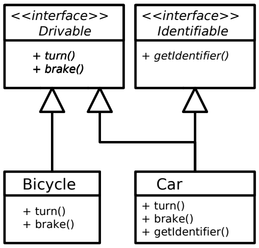
- What if we have a Lecturer who studies for another degree?
- If we do as shown, we have a bit of a problem
  - StudentLecturer inherits two different dance() methods
  - So which one should it use if we instruct a StudentLecturer to dance()?
- The Java designers felt that this kind of problem mostly 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)

Java only allows you to inherit from one class (which may itself inherit from one other, which may itself...). Many programmers coming from C++ find this limiting, but it just means you have to think of another way to represent your classes (arguably a better way).



Interfaces (Java only)

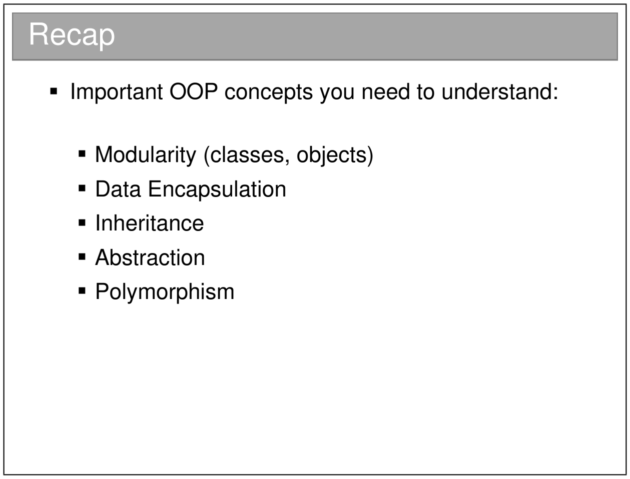
- Java has the notion of an **interface** which is like a class except:
  - There is no state whatsoever
  - All methods are abstract
- For an interface, there can then be no clashes of methods or variables to worry about, so we can allow multiple inheritance



```
Interface Drivable {
   public void turn();
   public void brake();
}

Interface Identifiable {
   public void getIdentifier();
}

class Bicycle implements Drivable {
   public void turn() {...}
   public void brake() {... }
}

class Car implements Drivable, Identifiable {
   public void turn() {...}
   public void brake() {... }
   Public void getIdentifier() {...}
}
```

abstract assumed for interfaces

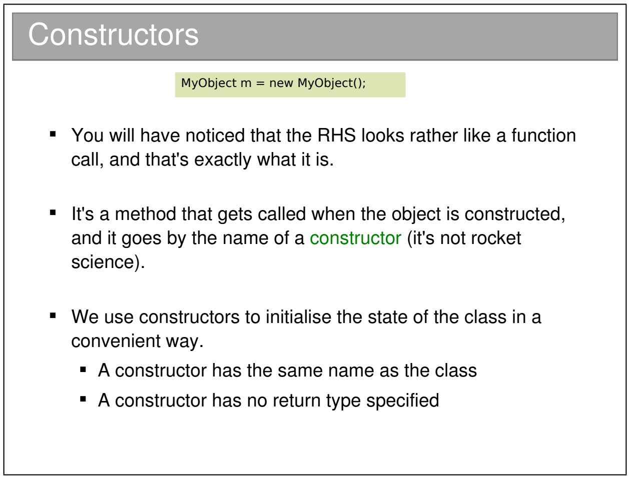
Interfaces are so important to Java they are considered to be the third reference type (the other two being classes and arrays).



Recap

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

2.5 Lifecycle of an Object

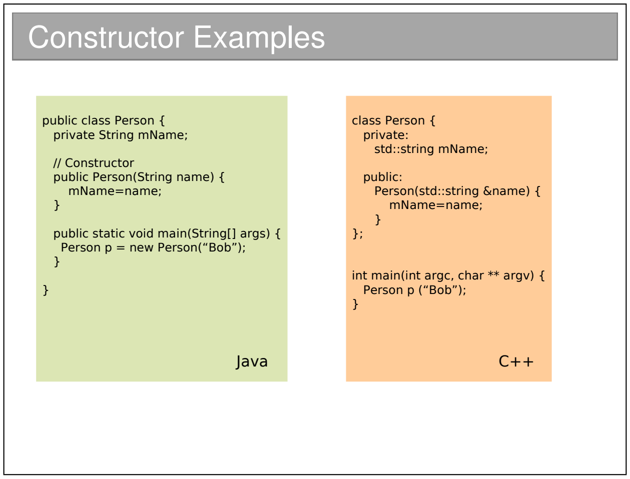


Constructors

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

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.
- It's a method that gets called when the object is constructed, and it goes by the name of a constructor (it's not rocket science).
- We use constructors to initialise the state of the class in a convenient way.
  - A constructor has the same name as the class
  - A constructor has no return type specified

You can't specify a return type for a constructor because it is always called using the special `new` keyword, which must return a reference to the newly constructed object. You can, however, specify arguments for a constructor in the same way as usual for a method.



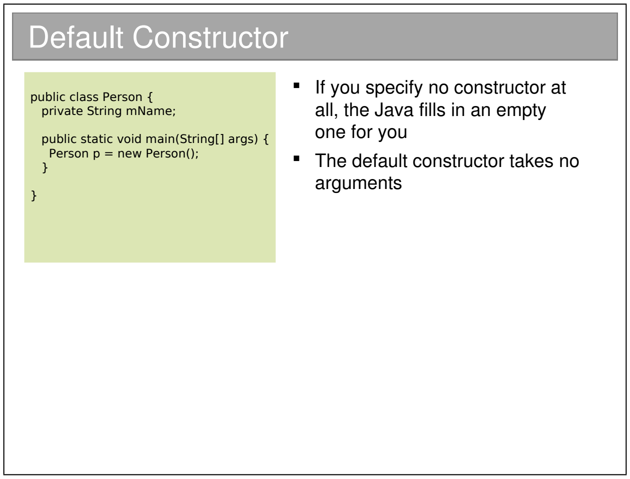
Constructor Examples

```
public class Person {
   private String mName;

   // Constructor
   public Person(String name) {
       mName=name;
   }

   public static void main(String[] args) {
     Person p = new Person(“Bob”);
   }

}
```

Java

```
class Person {
   private:
      std::string mName;

   public:
      Person(std::string &name) {
          mName=name;
      }
};

int main(int argc, char ** argv) {
   Person p (“Bob”);
}
```

C++

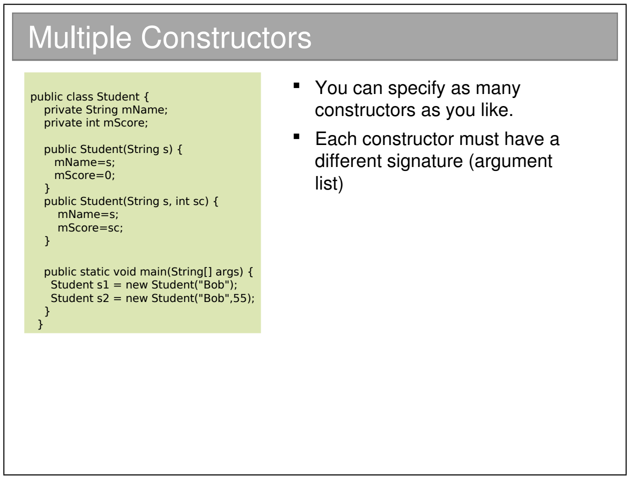


Default Constructor

```
public class Person {
   private String mName;

   public static void main(String[] args) {
     Person p = new Person();
   }

}
```

- If you specify no constructor at all, the Java fills in an empty one for you
- The default constructor takes no arguments

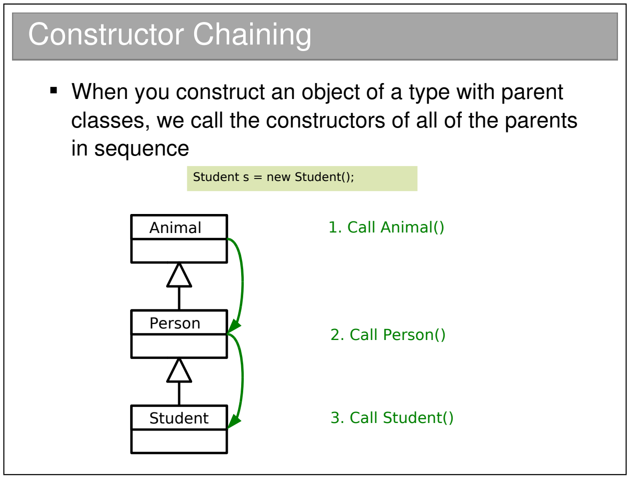
Every class has a constructor. The only question is whether it's been specified manually by the programmer or whether the compiler has filled in a default (empty) constructor. Beware: As soon as you specify any constructor whatsoever, Java won't add in a default constructor...



Multiple Constructors

```
public class Student {
   private String mName;
   private int mScore;

   public Student(String s) {
      mName=s;
      mScore=0;
   }
   public Student(String s, int sc) {
       mName=s;
       mScore=sc;
   }

   public static void main(String[] args) {
     Student s1 = new Student("Bob");
     Student s2 = new Student("Bob",55);
   }
 }
```

- You can specify as many constructors as you like.
- Each constructor must have a different signature (argument list)



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();
```

What actually happens is that the first line of a constructor *always* starts with super(), which is a call to the parent constructor (which itself starts with super(), etc.). If it does not, the compiler adds one for you:

```
public class Person {
  public Person() {

  }
}
```

becomes:

```
public class Person {
  public Person() {
    super();
  }
}
```

This can get messy though: what if the parent does not have a default constructor? In this case, the code won't compile, and we will have to manually add a call to `super` ourselves, using the appropriate arguments. E.g.

```
public class Person {
  protected String mName;
  public Person(String name) {
    mName=name;
  }
}

public class Student extends Person {
  private int mScore;
  public Student (String name, int score) {
    super(name);
    mScore = score;
  }
}
```

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:

    // Constructor
    FileReader() {
      f = fopen("myfile","r");
    }

    // Destructor
    ~FileReader() {
      fclose(f);
    }

  private :
    FILE *file;
}
```

```
int main(int argc, char ** argv) {

  // Construct a FileReader Object
  FileReader *f = new FileReader();

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

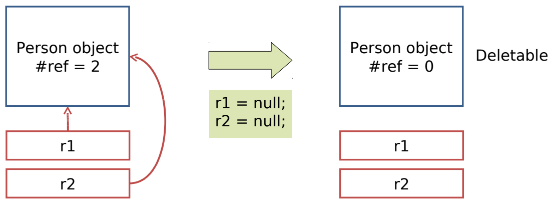
C++

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 if the programmer doesn't explicitly tell you it is?

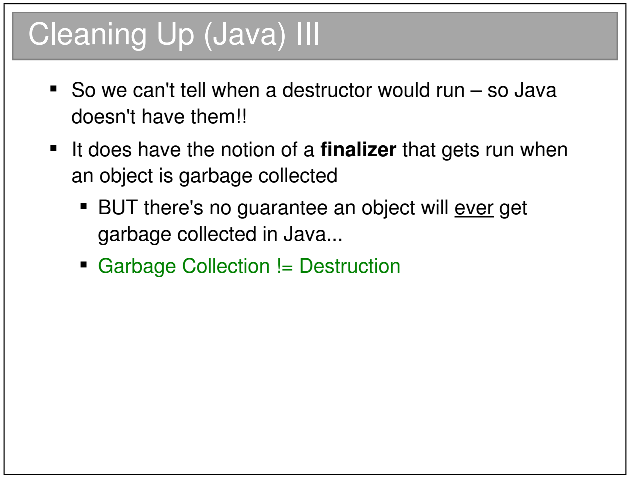
Cleaning Up (Java) I

- 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) II

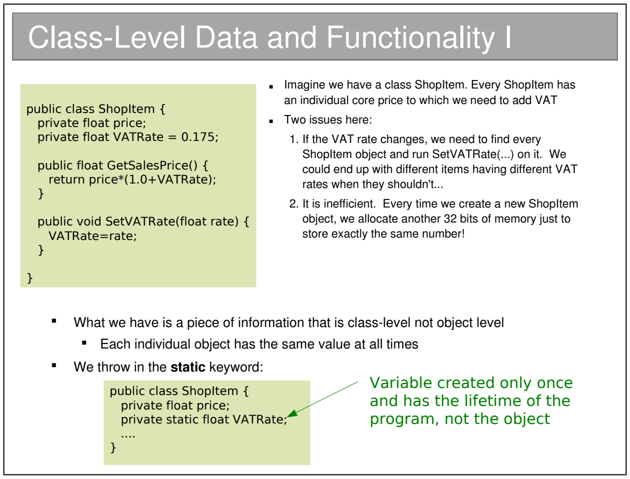
- Good:
  - System cleans up after us
- Bad:
  - It has to keep searching for objects with no references. This requires effort on the part of the CPU so it degrades performance.
  - We can't easily predict when an object will be deleted



Cleaning Up (Java) III

- 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 <u>ever</u> get garbage collected in Java...
  - Garbage Collection != Destruction

2.6 Class-Level Data



Class-Level Data and Functionality I

```
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;
   }

}
```

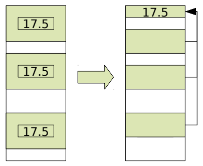
- Imagine we have a class ShopItem. Every ShopItem has an individual core price to which we need to add VAT
- Two issues here:
  1. If the VAT rate changes, we need to find every ShopItem object and run SetVATRate(...) on it. We could end up with different items having different VAT rates when they shouldn't...
  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 II



- 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. E.g

```
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) {...}
}
```

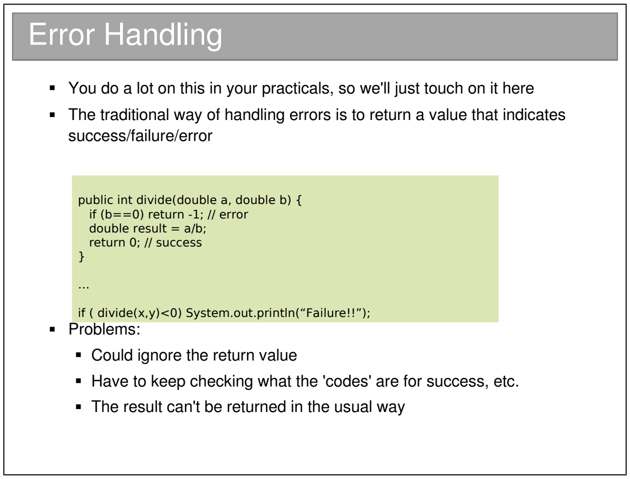
```
...
Math mathobject = new Math();
mathobject.sqrt(9.0);
...
```

vs

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

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

2.7 Exceptions

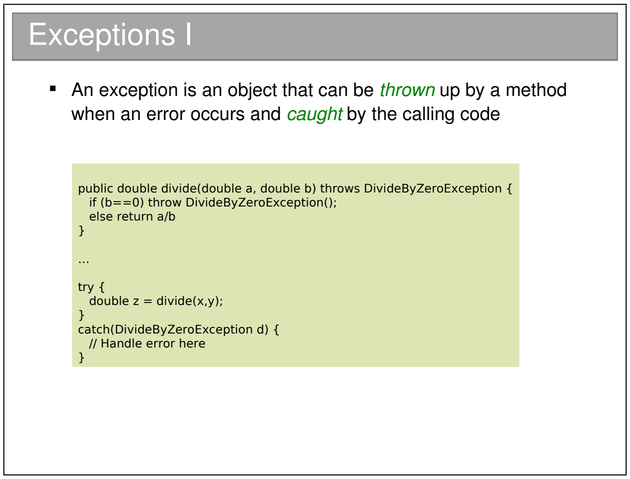


Error Handling

- You do a lot on this in your practicals, so we'll just touch on it here
- The traditional way of handling errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
   if (b==0) return -1; // error
   double result = a/b;
   return 0; // success
}

...

if ( divide(x,y)<0) System.out.println("Failure!!");
```

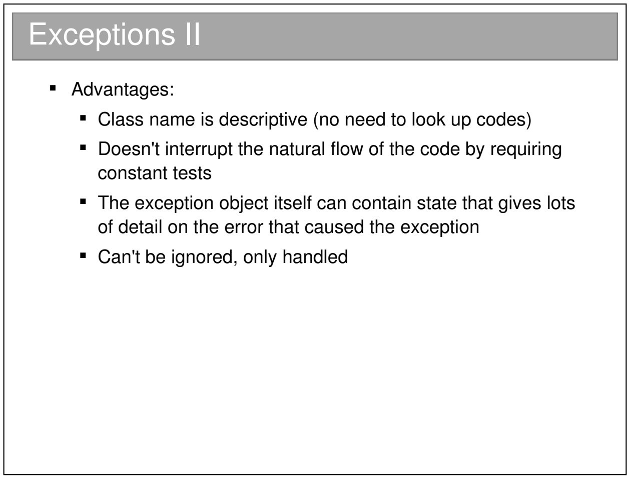
- Problems:
  - Could ignore the return value
  - Have to keep checking what the 'codes' are for success, etc.
  - The result can't be returned in the usual way



Exceptions I

- An exception is an object that can be *thrown* up by a method when an error occurs and *caught* by the calling code

```
public double divide(double a, double b) throws DivideByZeroException {
   if (b==0) throw DivideByZeroException();
   else return a/b
}

...

try {
   double z = divide(x,y);
}
catch(DivideByZeroException d) {
   // Handle error here
}
```

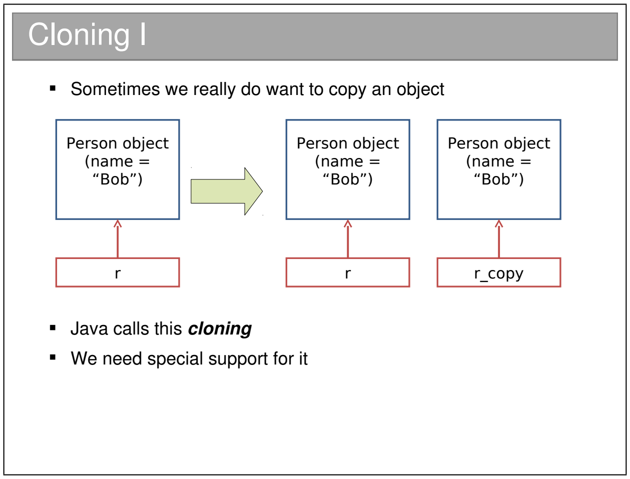

Exceptions II

- Advantages:
  - Class name is descriptive (no need to look up codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only handled

There is a *lot* more we could say about exceptions, but you have the basic tools to understand them and they will be covered in your practical Java course. Just be aware that exceptions are very powerful and very popular in most modern programming languages. If you're struggling to understand them, take a look at:

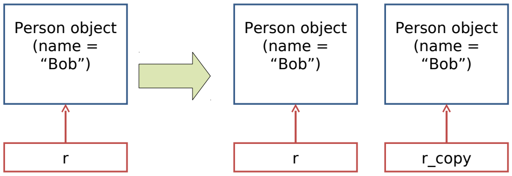
`http://java.sun.com/docs/books/tutorial/essential/exceptions/`

2.8 Copying or Cloning Java Objects

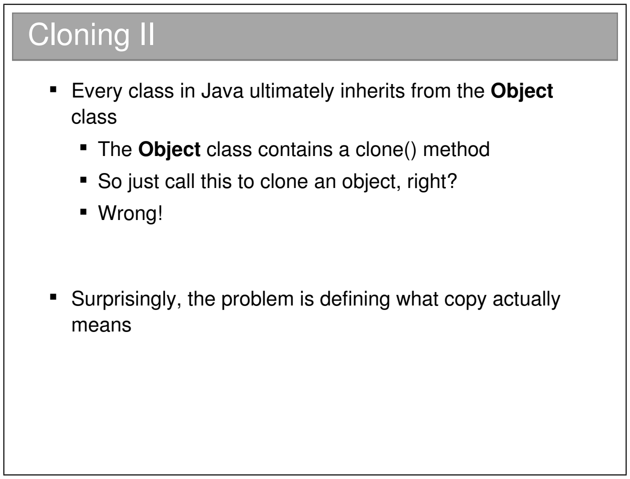


Cloning I

- Sometimes we really do want to copy an object


Person object (name = "Bob")
r
Person object (name = "Bob")
r
Person object (name = "Bob")
r_copy

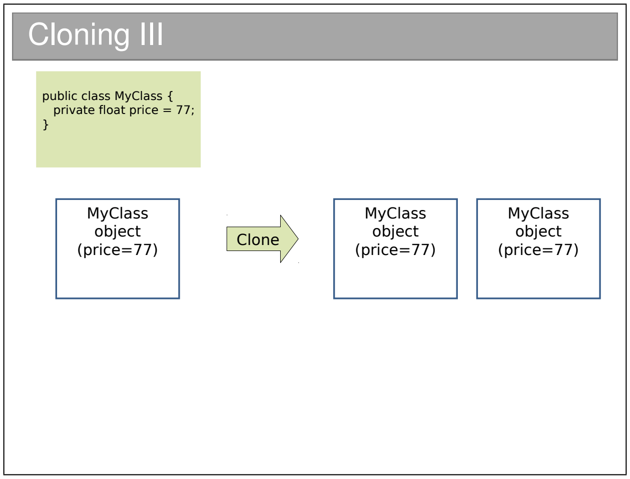
- Java calls this ***cloning***
- We need special support for it



Cloning II

- 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

Java is unusual in that it really, really wants you to use OOP. In your practicals you must have noticed that, even to do simple procedural stuff, you had to encase everything in a class—even the main() method. A further decision they made is that ultimately *all* classes will inherit from a special Object class. i.e. the top of all inheritance trees is Object even though we never explicitly say so in code...



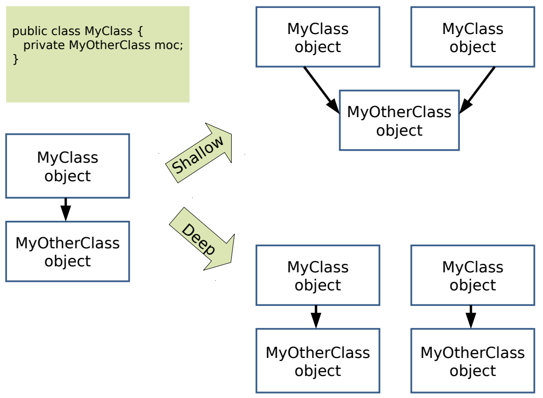
Cloning III

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

MyClass object (price=77)
Clone
MyClass object (price=77)
MyClass object (price=77)

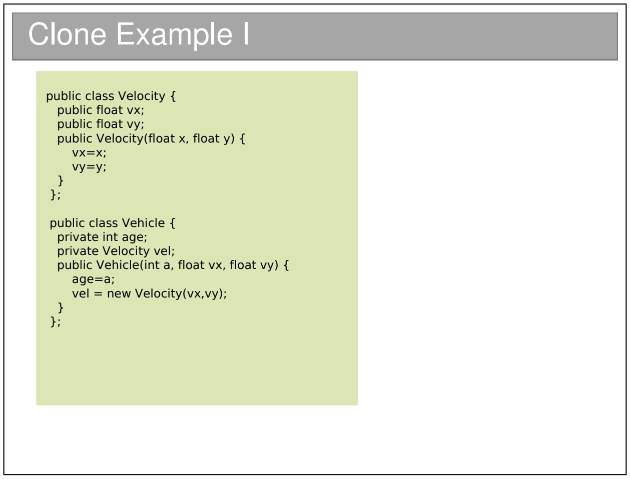
Shallow and Deep Copies

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



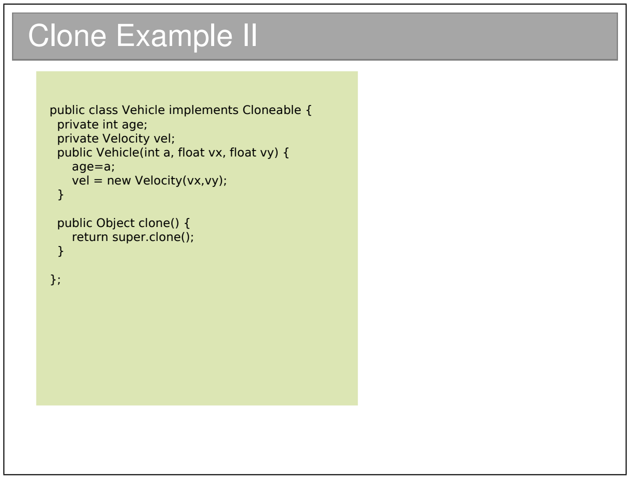
Java Cloning

- 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



Clone Example I

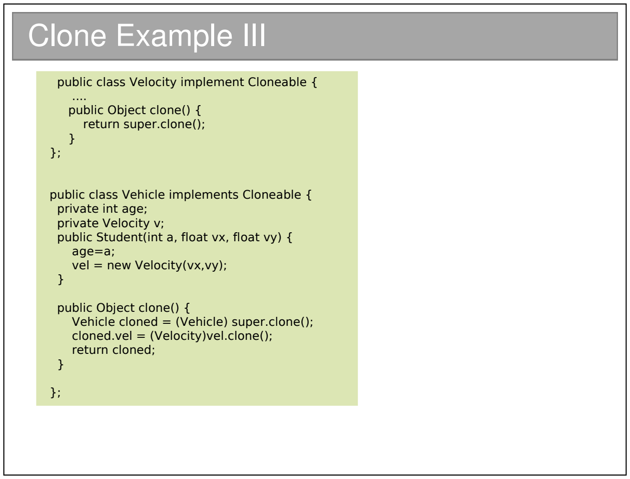
```
public class Velocity {
  public float vx;
  public float vy;
  public Velocity(float x, float y) {
      vx=x;
      vy=y;
  }
};

public class Vehicle {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }
};
```



Clone Example II

```
public class Vehicle implements Cloneable {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }

  public Object clone() {
      return super.clone();
  }

};
```

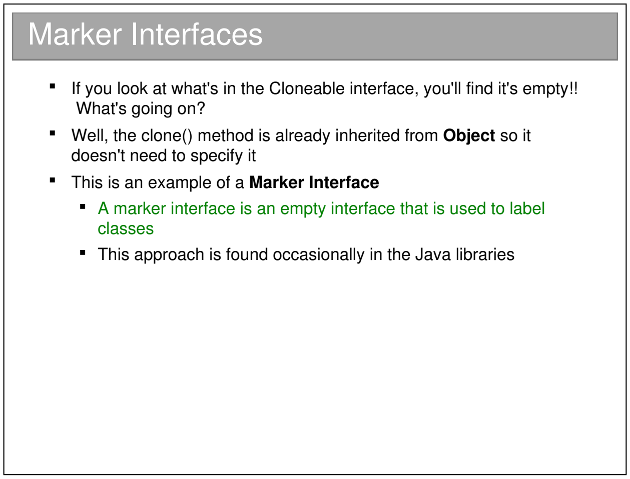
Here we fill in the `clone()` method using `super.clone()`. You can think of this as doing a byte-for-byte copy of an object in memory. Any primitive types (such as age) will therefore be copied. And references will also be copied, but not the objects they point to. Hence this much gets us a shallow copy.



Clone Example III

```
  public class Velocity implement Cloneable {
     ....
     public Object clone() {
        return super.clone();
     }
};


public class Vehicle implements Cloneable {
  private int age;
  private Velocity v;
  public Student(int a, float vx, float vy) {
     age=a;
     vel = new Velocity(vx,vy);
  }

  public Object clone() {
     Vehicle cloned = (Vehicle) super.clone();
     cloned.vel = (Velocity)vel.clone();
     return cloned;
  }

};
```

A deep clone requires that we clone the objects that are referenced (and they, in turn clone any objects they reference, and so on). Here we make Velocity cloneable and make sure to clone the member variable that Vehicle has.

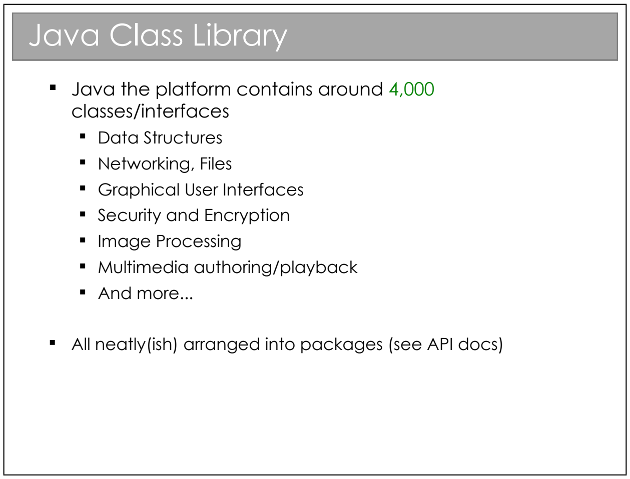

Marker Interfaces

- If you 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).

Chapter 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)

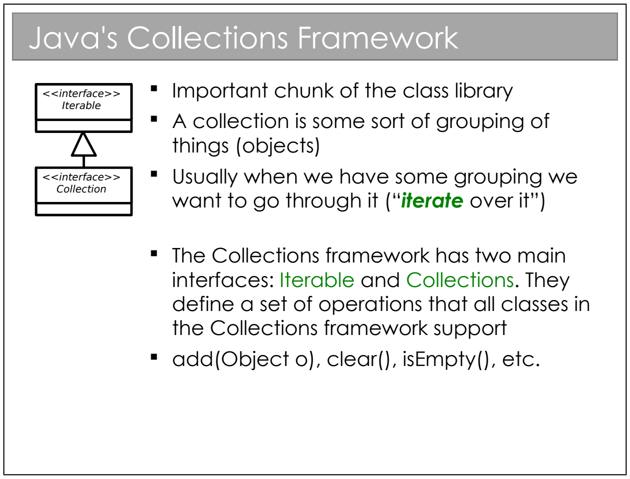
Remember Java is a *platform*, not just a programming language. It ships with a huge *class library*: that is to say that Java itself contains a big set of built-in classes for doing all sorts of useful things like:

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

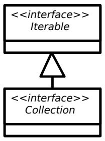
Of course, most programming languages have built-in classes, but Java has a big advantage. Because Java code runs on a virtual machine, the underlying platform is abstracted away. For C++, for example, the compiler ships with a fair few data structures, but things like I/O and graphical interfaces are 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.

3.0.1 Collections and Generics



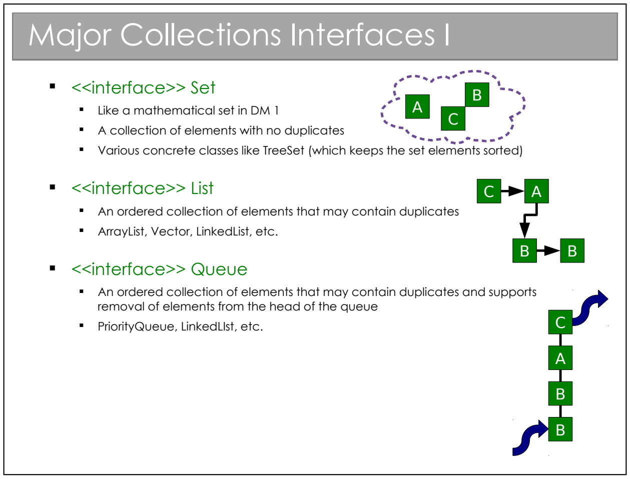
Java's Collections Framework



- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

- The Collections framework has two main interfaces: Iterable and Collections. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

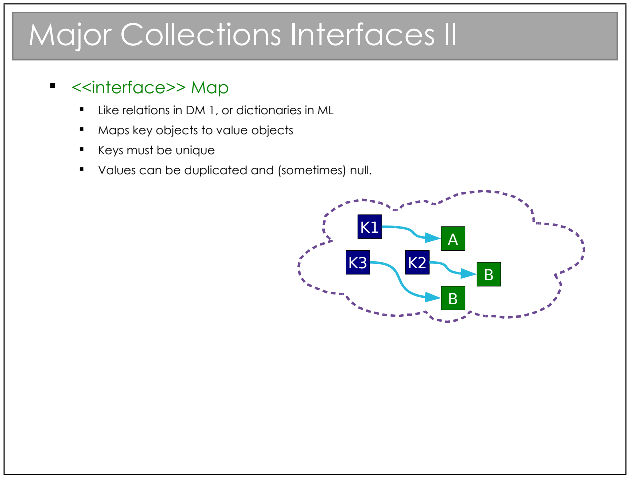
The Java Collections framework is a set of interfaces and classes that handles groupings of objects and allows us to implement various algorithms invisibly to the user (you'll learn about the algorithms themselves next term).



Major Collections Interfaces I

- <<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.

A
B
C
C
A
B
B
C
A
B
B



Major Collections Interfaces II

- <<interface>> Map
  - Like relations in DM 1, or dictionaries in ML
  - Maps key objects to value objects
  - Keys must be unique
  - Values can be duplicated and (sometimes) null.

K1
A
K3
K2
B
B

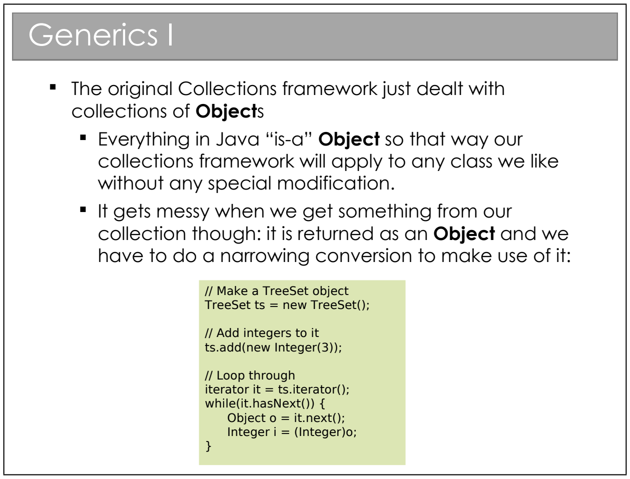
There are other interfaces in the Collections class, and you may want to poke around in the API documentation. In day-to-day programming, however, these are likely to be the interfaces you use.

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.

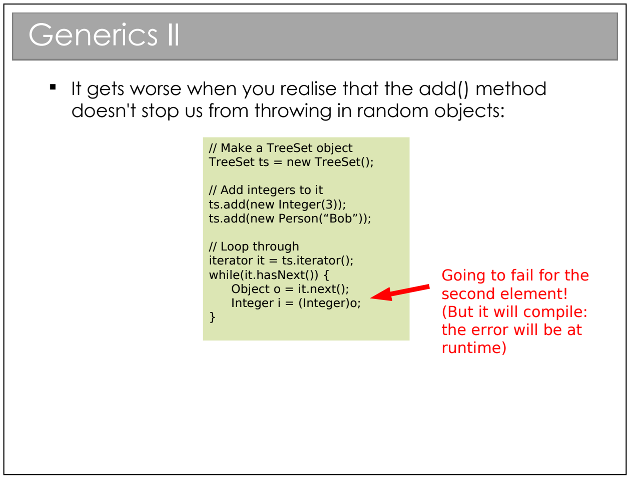
Now, don't worry about what's going on behind the scenes (that comes in the Algorithms course), just recognise that there are a series of implementations in the class library that you can use, and that each has different properties.



Generics I

- The original Collections framework just dealt with collections of **Object**s
  - 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;
}
```

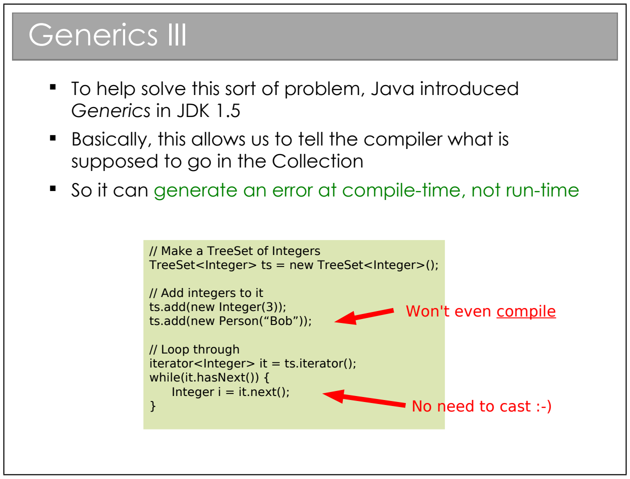


Generics II

- 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)

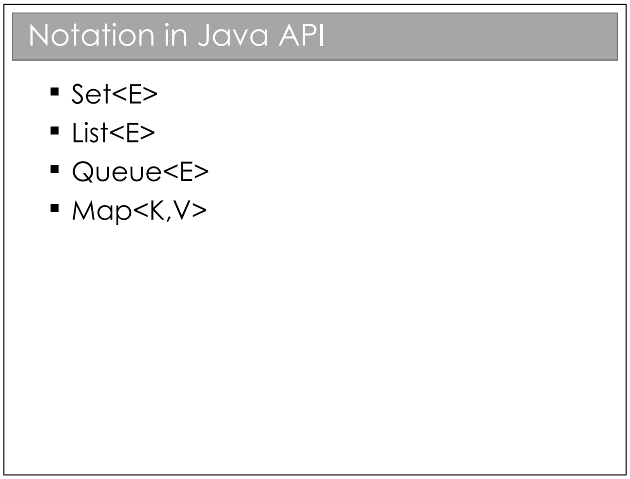


Generics III

- 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();
}
```

Now, assuming you're still awake (long shot, I know), you might have noticed that this is all about determining types at compile-time rather than dynamically at run-time. Which of course is what the (static) polymorphism you met in ML does. So really generics just adds in static polymorphism, calling it "generics" in an attempt to avoid confusion (or perhaps cause it—I'm never sure).

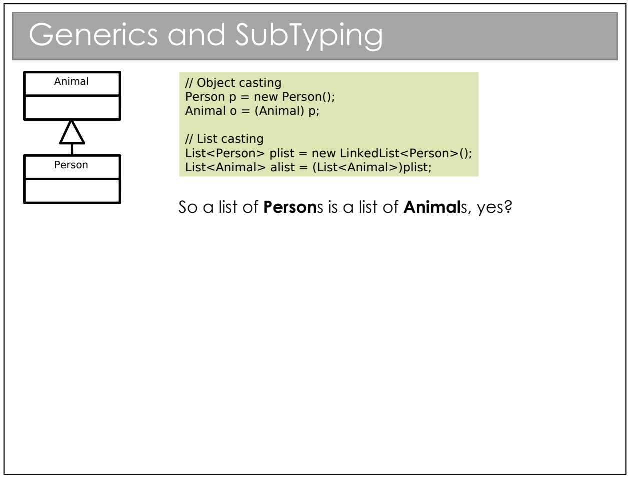


Notation in Java API

- 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>()
```



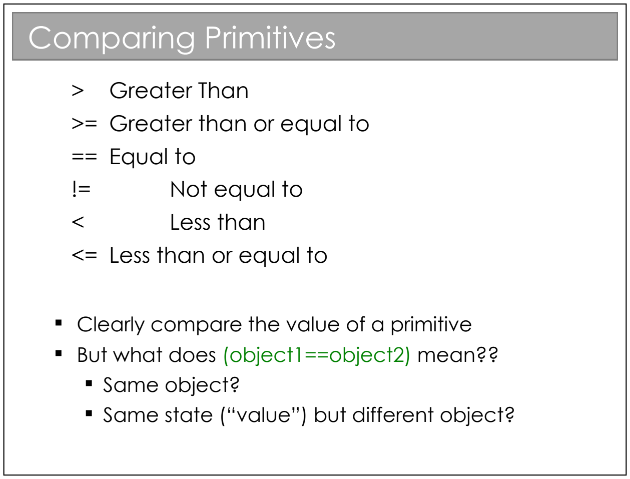
Generics and SubTyping

Animal
Person

```
// 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 **Person**s is a list of **Animal**s, yes?

3.0.2 Comparing Java Objects

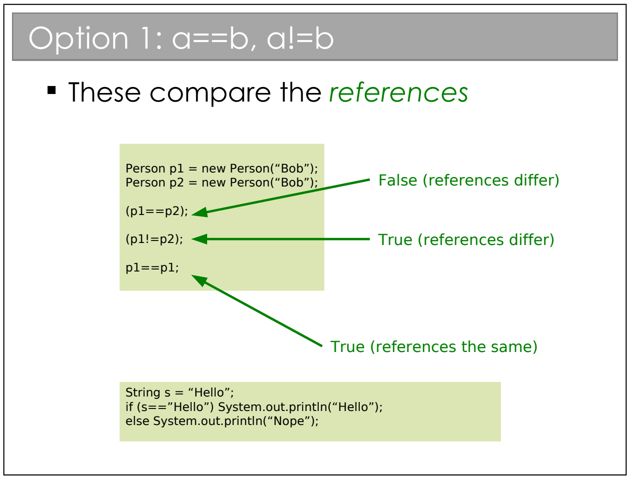


Comparing Primitives

>	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.



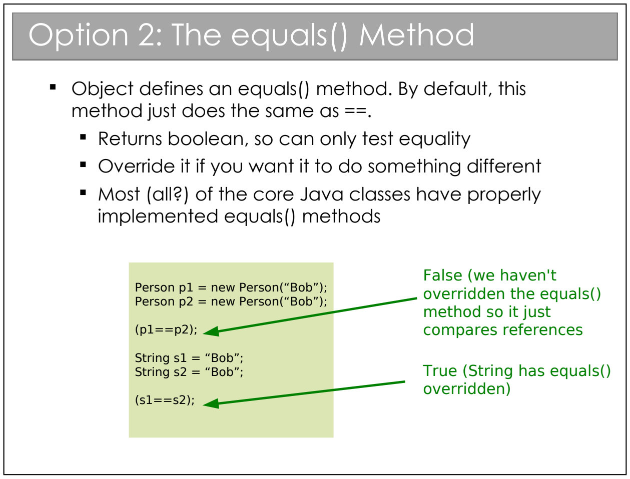
Option 1: a==b, a!=b

- 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");
```



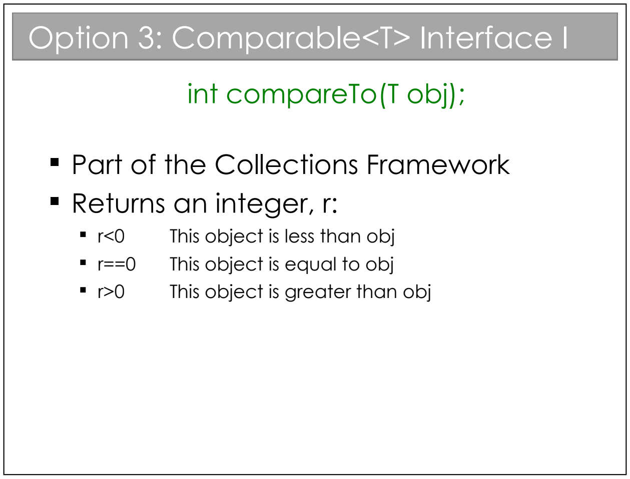
Option 2: The equals() Method

- 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)

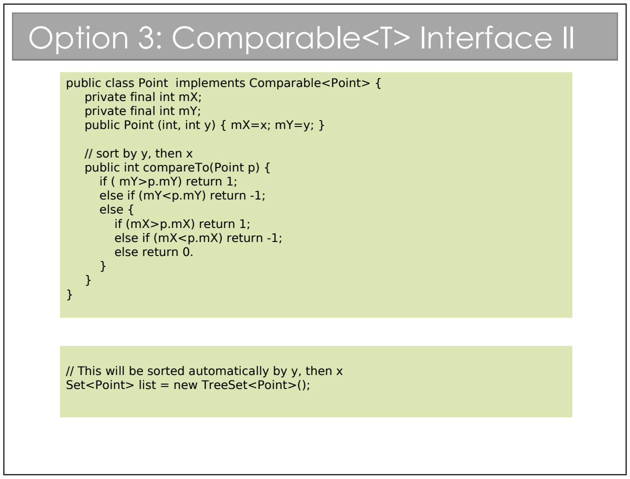
I find this mildly irritating: every class you use will support equals() but you’ll have to check whether or not it has been overridden to do something other than ==. Personally, I only use equals() on objects from core Java classes.



Option 3: Comparable<T> Interface I

```
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

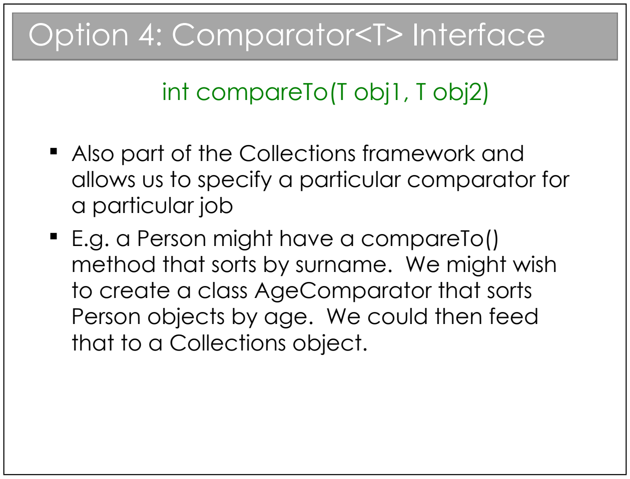


Option 3: Comparable<T> Interface II

```
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*.



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.

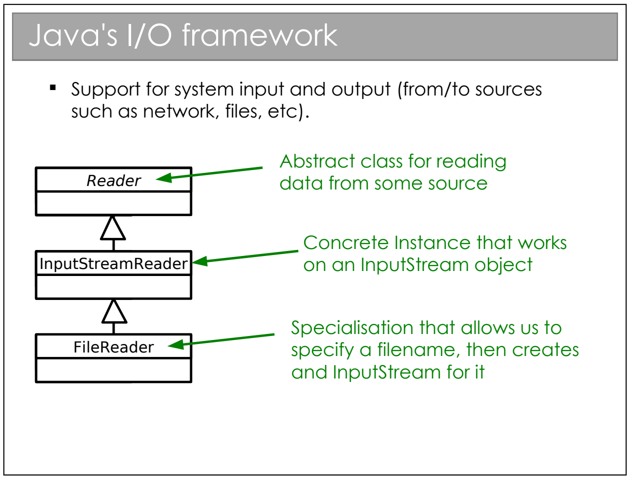
At first glance, it may seem that Comparator doesn't add much over Comparable. However it's very useful to be able to specify Comparators and apply them dynamically to Collections. If you look in the API, you will find that Collections has a *static* method sort(List l, Comparator, c).

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());
```

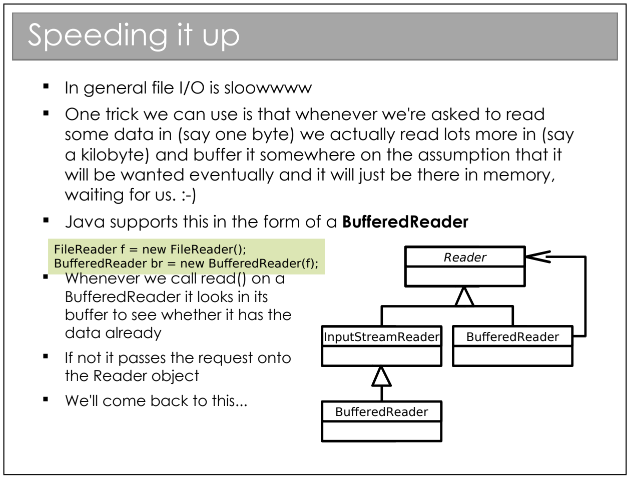
3.0.3 Java's I/O Framework



Java's I/O framework

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

Reader
InputStreamReader
FileReader
Abstract class for reading data from some source
Concrete Instance that works on an InputStream object
Specialisation that allows us to specify a filename, then creates and InputStream for it

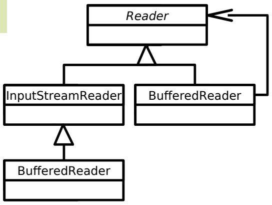


Speeding it up

- In general file I/O is sloowwww
- 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 that 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

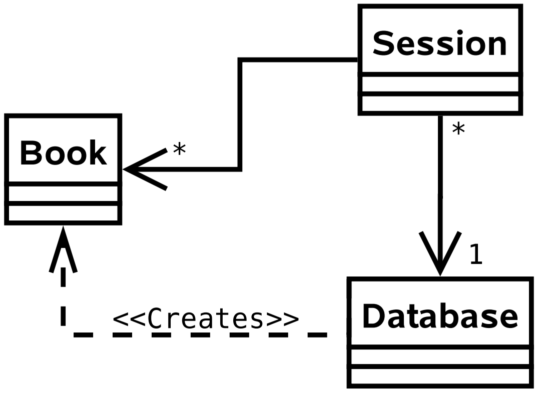
The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments[1] with a single word, the code becomes more readable and maintainable. Furthermore, you can insert the word into the class name itself, making the class self-describing.

[1]You 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 set 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).

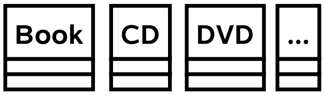
Database. This class wraps around our database, hiding away the query syntax (i.e. the SQL statements or similar).

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

4.3 Supporting Multiple Products

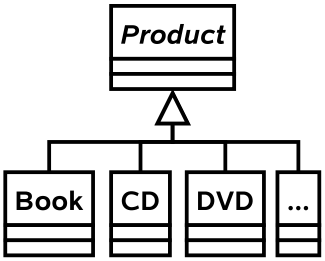
Problem: Selling books is not enough. We need to sell CDs and DVDs too. And maybe electronics. Oh, and sports equipment. And...

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 polymorphism 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.

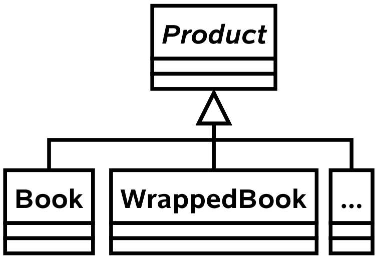
4.4 The Decorator Pattern

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.

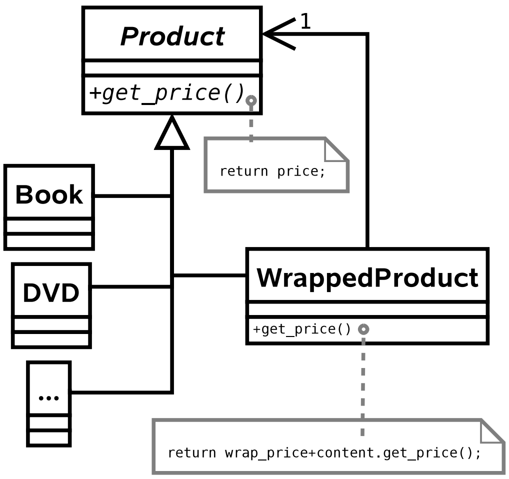


Don't. Do. This. Ever.

- ✔ 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.

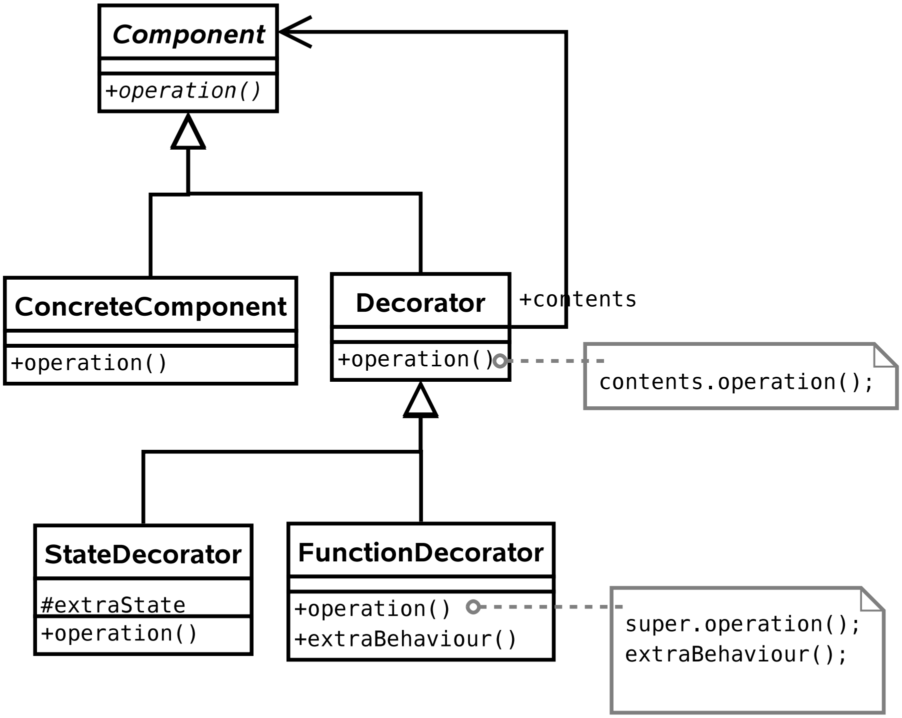


- ✔ We can add new product types and they will be automatically wrappable.
- ✔ We can dynamically convert an established product object into a wrapped product and back again without copying overheads.

- ✗ 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 to an object *dynamically*. By that I mean we can take an object in the system and effectively give it extra state or functionality. I say 'effectively' because the actual object in memory is untouched. Rather, we create a new, small object that 'wraps around' the original. To remove the wrapper we simply discard the wrapping object. Real world example: humans can be 'decorated' with contact lenses to improve their vision.



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

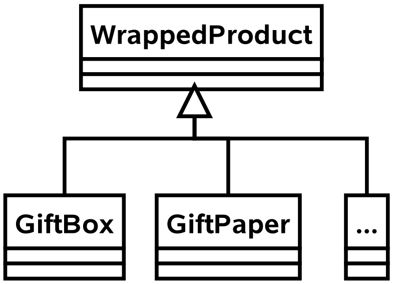
Problem: We need to handle a lot of gift options that the customer may switch between at will (different wrapping papers, bows, gift tags, gift boxes, gift bags, ...).

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 ...
}
```

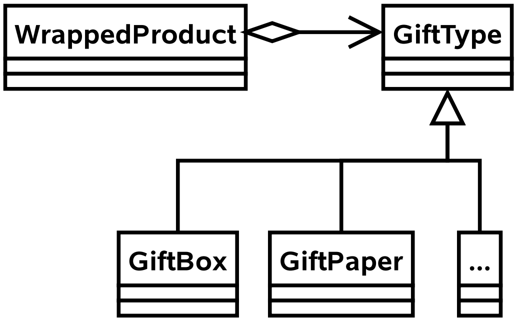
✔ It works.
✗ The code is far less readable.
✗ Adding a new wrapping option is ugly.

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!

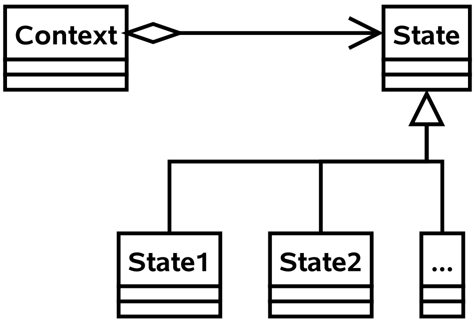
Solution 3: Let's keep our idea of representing states with a class hierarchy, but use a new abstract class as the parent:



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.



4.6 Strategy Pattern

Problem: Part of the ordering process requires the customer to enter a postcode that 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?

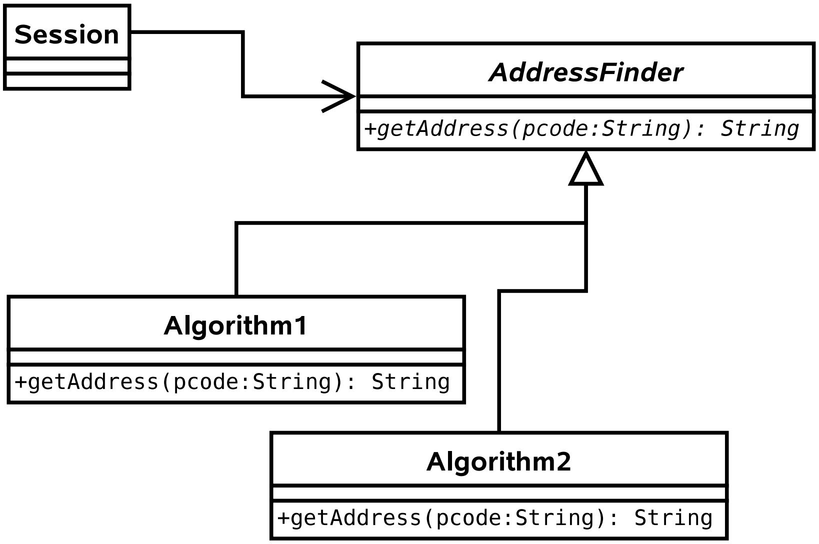
Solution 1: Let there be a class AddressFinder with a method getAddress(String pcode). We could add lots of if/then/else statements to the getAddress() function.

```
String getAddress(String pcode) {
   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 principle[2].

[2]This 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.

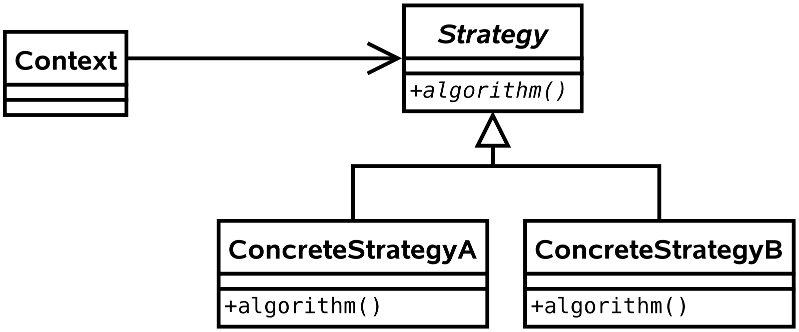
Solution 2: Make AddressFinder abstract with a single abstract function getAddress(String pcode). Derive a new class for each of our algorithms.


Session
AddressFinder
+getAddress(pcode:String): String
Algorithm1
+getAddress(pcode:String): String
Algorithm2
+getAddress(pcode:String): String

✔ 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 run-time.
- 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 Strategys 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.

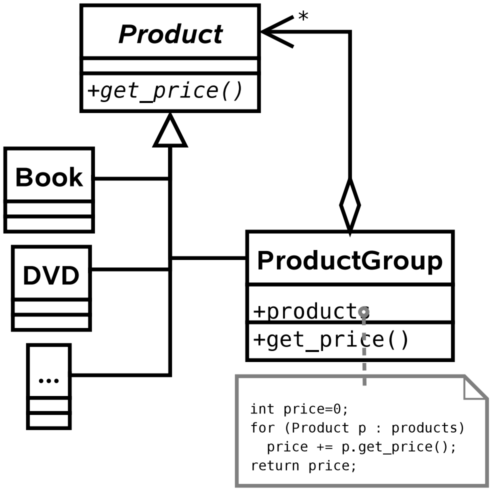
4.7 Composite Pattern

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:

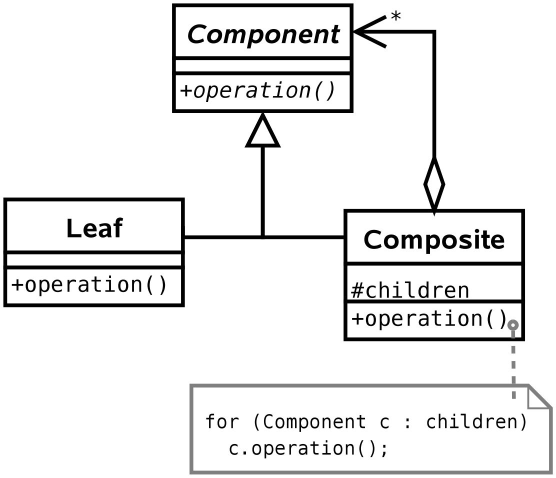


If you're still awake, you may be thinking this is a bit like the **Decorator** pattern, except that the new class supports associations with multiple Products (note the * by the arrowhead). Plus the intent is different – we are not adding new functionality but rather supporting the same functionality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

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 that 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!

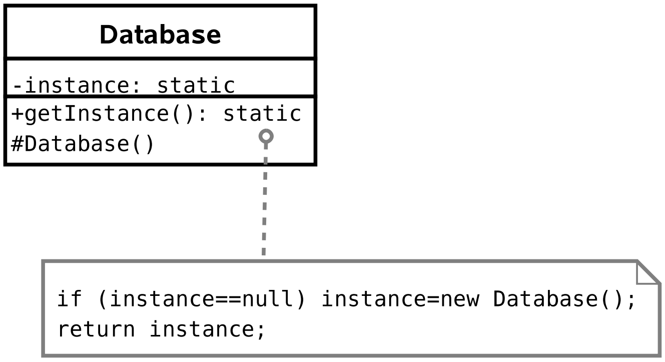
Solution 3: Create an instance of Database at startup, and pass it as a constructor parameter to every Session we create, storing a reference in a member variable for later use.

```
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.



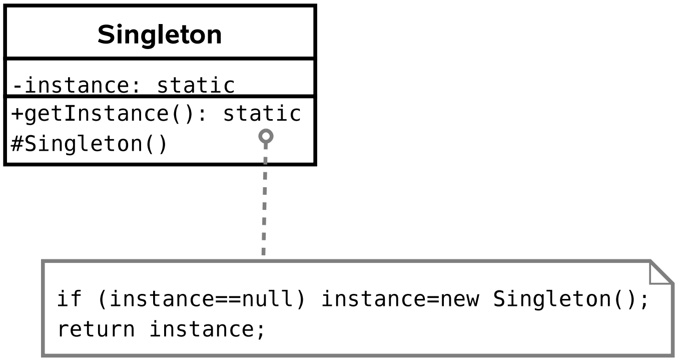
```
if (instance==null) instance=new Database();
return instance;
```

- ✔ *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.



```
if (instance==null) instance=new Singleton();
return instance;
```

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.

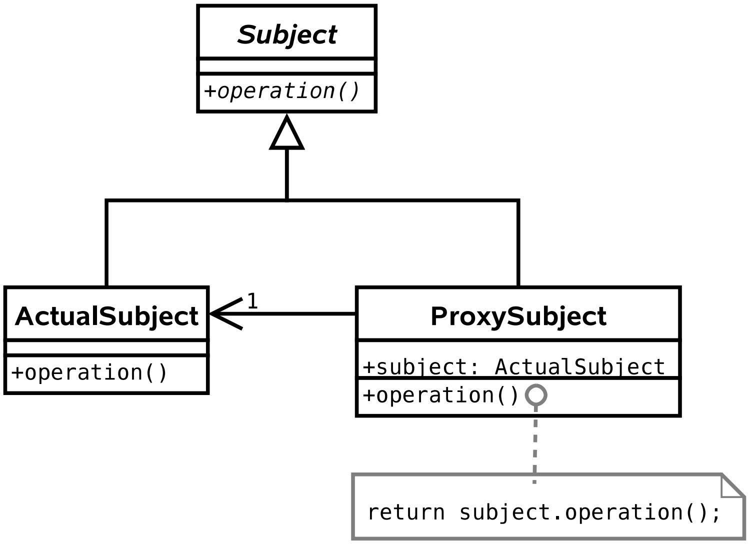
Protected members are accessible to the class, any subclasses, *and all classes in the same package.* Therefore, any class in the same package as your base class will be able to instantiate Singleton objects at will, using the new keyword!

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.

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 place-holder 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 does want to access more information, the associated `get()` methods in the proxy object automatically 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.

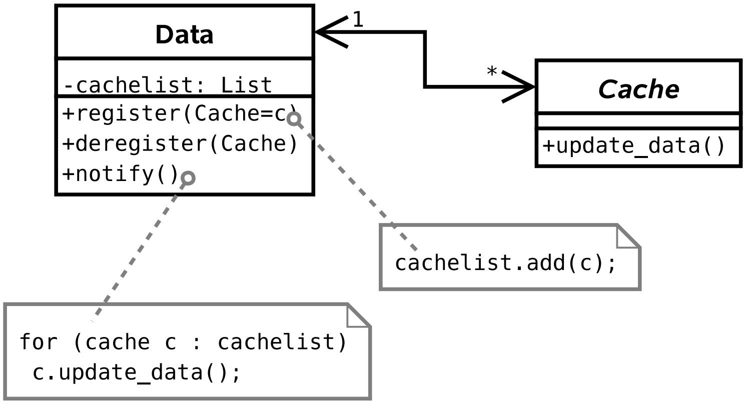
4.10 Observer Pattern

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).

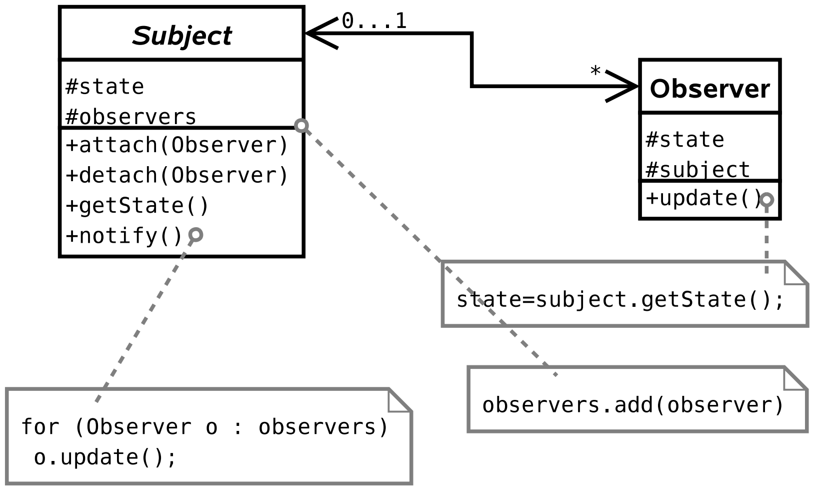
- ✗ How frequently should we poll? Too quickly and we might as well not have cached at all. Too slow and changes will be slow to propagate.

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.



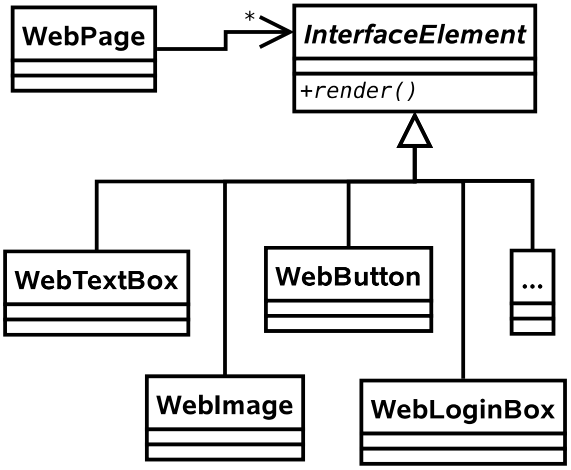
4.10.1 Generalisation

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.


Subject
#state
#observers
+attach(Observer)
+detach(Observer)
+getState()
+notify()
0...1
*
Observer
#state
#subject
+update()
state=subject.getState();
observers.add(observer)
for (Observer o : observers)
o.update();

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?

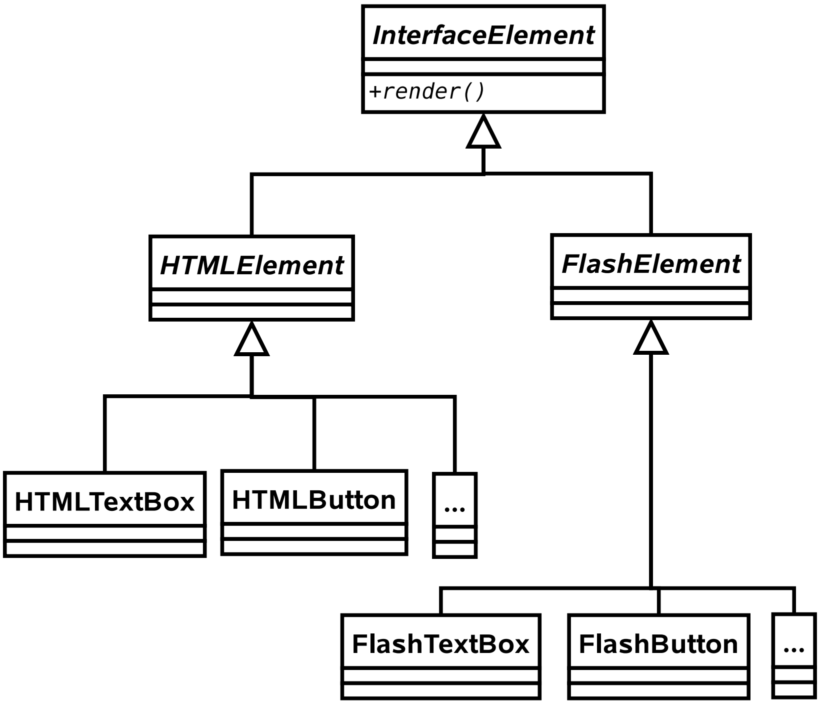
Solution 1: Store a variable ID in the InterfaceElement class, or use the **State** pattern on each of the subclasses.

✔ 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.

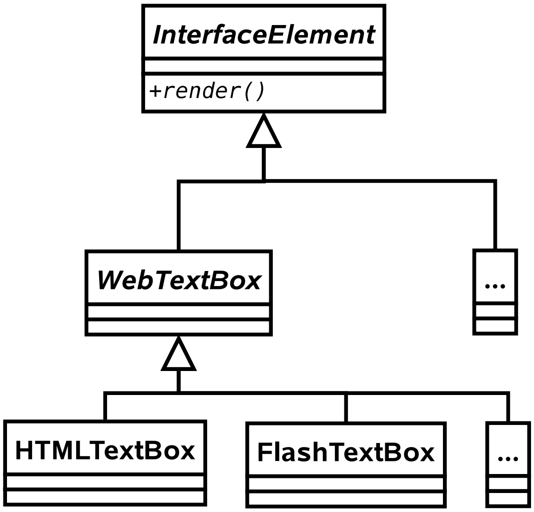
✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

Solution 2: Create specialisations of InterfaceElement:



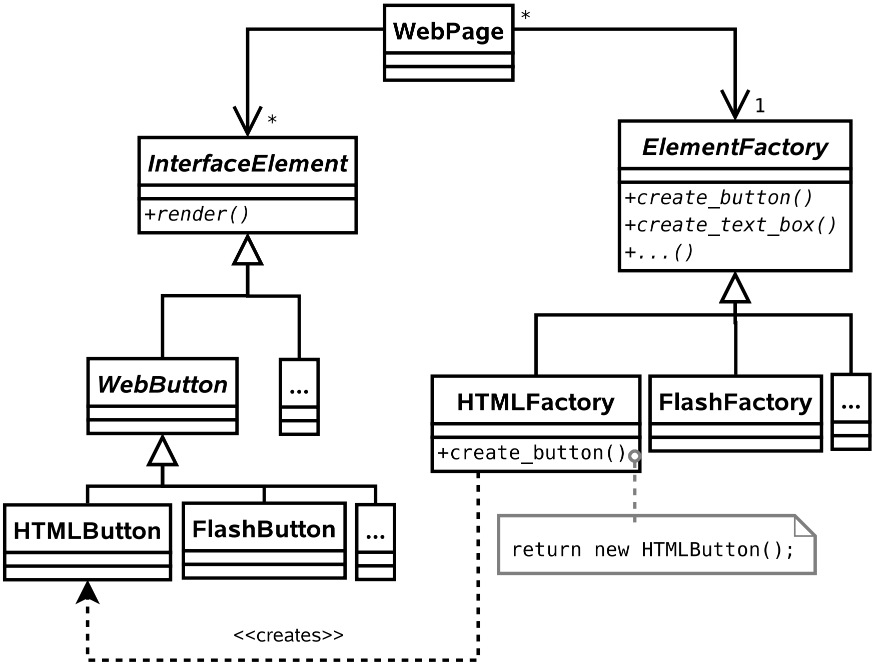
- ✗ Lots of code duplication.
- ✗ Nothing keeps the different TextBoxes in sync as far as the interface goes.
- ✗ A lot of work to add a new interface component type.
- ✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

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.

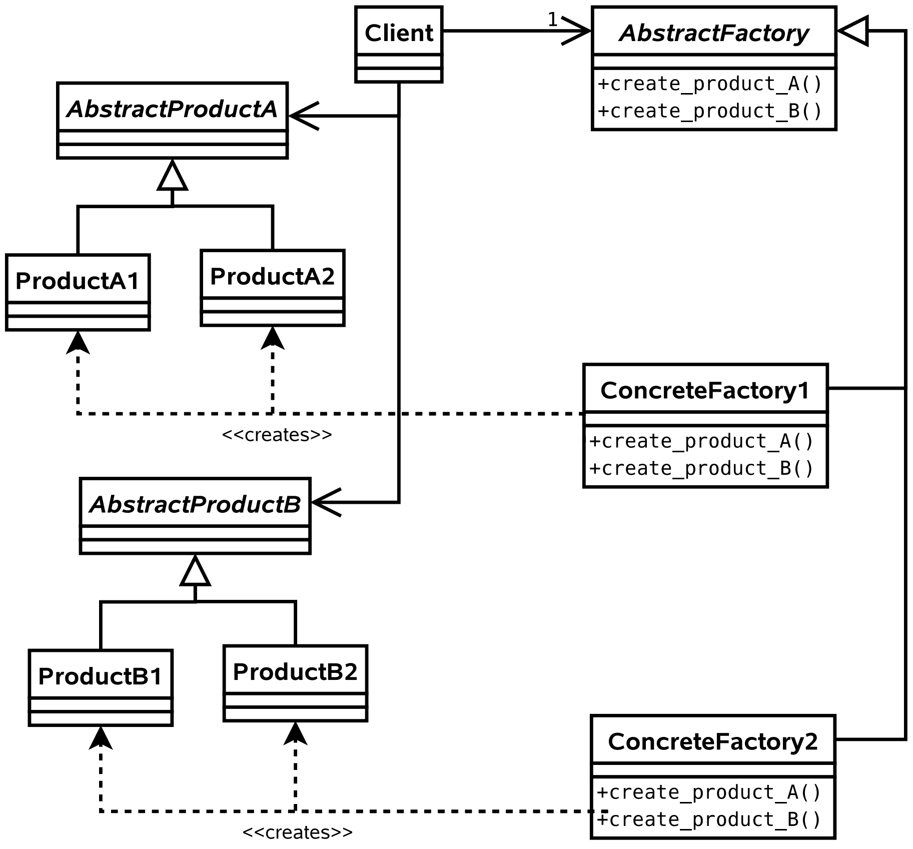


- ✔ Standardised interface to each element type.
- ✔ A given WebPage can only generate elements from a single family.
- ✔ Page is completely decoupled from the family so adding a new family of elements is simple.
- ✗ Adding a new element (e.g. SearchBox) is difficult.

✗ 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.



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.

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 and object to alter its behaviour when its internal state changes.

Strategy Define a family of algorithms, encapsulate each on, 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 objecta 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 s 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!].

Once we have compiled our Java source code, we end up with a set of .class files; these contain bytecode. 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.