Object Oriented Programming
Dr Robert Harle

IA CST, PPS (CS) and NST (CS)
Lent 2011/12

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. As we discussed in the Computer Fundamentals course, there are many reasons we started with this, chief amongst them being that everything is a well-formed *function*, by which we mean that the 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 it's very similar to ML in many respects and translation is very easy. This is really because

functional languages have a very carefully defined set of features and rules.

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. Unlike their functional equivalents, imperative languages can look quite different to each other, although as time goes on there does seem to be a uniformity arising. Imperative programming is much more flexible (some would say it gives you more rope to hang yourself with) and crucially, not all imperative languages support all of the same language concepts, and not always in the same way. So, if you just learn one language (e.g. Java) you'll probably struggle to separate the underlying programming concepts from the Java-specific quirks. So jumping ship to C++ will be a bit tricky...

And that's what we saw when the Java practicals first came into being: students learnt to program in Java, not how to use the Object Oriented Programming (OOP) concepts. And thus the OOP course was born...

Mostly we will be using Java to implement our ideas. Sometimes we'll need some other languages to demonstrate concepts Java doesn't have.
.

You won't be expected to program in anything other than Java in the exam

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
  - Java specification book: http://java.sun.com/docs/books/jls/
  - 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/1112/OOProg/

There is no shortage of books and websites describing the basics of OOP. The concepts themselves are quite abstract, but most texts will use a specific language to demonstrate them. The books I've given favour Java but 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 do *not* assume anything from these lectures (although the deeper knowledge here may help you!)

Chapter 1

From Functional to Imperative Programming

As we saw in the CF course, 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. There are a few important differences that stand out.

1.1 Explicit Start Points

Explicit Start Points

Java: public static void main(String args[])

C/C++: int main(int argc, char **argv)

python:
```
def main():
    # main code here

if __name__ == "__main__":
    main()
```

Mostly in ML you fed in code line-by-line at the ML interpreter prompt and it just interpreted each line as you went. If you wanted to distribute your program, you could just write all of the lines into a file and pass the file to the interpreter.

Difficulties come when you have lots of code distributed across lots of files. In the ML interpreter model, the order files get processed might affect the outcome. Thus many imperative languages have explicit start points: functions they start running when you run the program. Each program has precisely one start point so there's no ambiguity. The most common name for the start point is the `main()` method.

See workbook 1

1.2 Immutable to Mutable Data

Immutable to Mutable Data

ML

```
- val x=5;
> val x = 5 : int
- x=7;
> val it = false : bool
- val x=9;
> val x = 9 : int
```

Java

```
int x=5;
x=7;

int x=9;
```

In a pure functional language[1] the data are immutable. That is to say that whenever we assign a value to a variable, it can never be changed. This is important since it allows all sorts of clever optimisations to take place, not least to delay evaluation, knowing that the result will not change. e.g. `pow(x)` will be the same now as later because the function depends only on its argument and the argument is immutable.

[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 nature! When I talk about ML here I will almost always be ignoring the imperative hacks that have invaded such languages. Although you may wish to reflect on *why* they have invaded...

1.3 Explicit Variable Types

Types and Variables

- We write code like:

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

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

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*) as well as the types of its arguments.

E.g. `int x` declares `x` to be an integer; `float get(int y)` declares a function `get` that takes an integer and returns a floating point value.

E.g. Primitive Types in Java

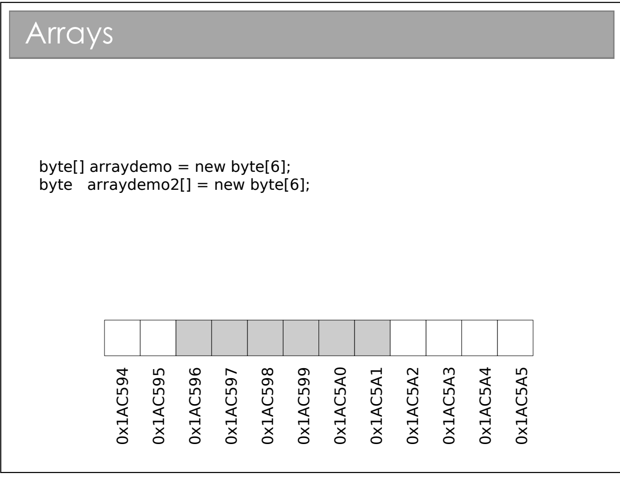
- "Primitive" types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 bits as a signed integer
- int – 32 bits as a signed integer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

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.



See workbook 3

See workbook 3

Whilst ML was littered with tuples and lists, imperative languages feature *arrays* as special built-in types.[2]. An array is a set of values stored sequentially in a single chunk of memory. As such they have some important properties:

- They have $O(1)$ element access. If you want the sixth element of array `a`, you just jump straight to it in memory by saying `a[5]`[3]
- They are more efficient in storage because they store just the value—the next element is implicitly found in the next memory slot.
- They have a set size. Expanding an array involves creating a new (bigger) array in memory, copying over the elements from the old one, and then freeing up the memory associated with the old one. This is costly.

Why don't pure functional languages have arrays? There is an implicit link to the memory: that subsequent elements are allocated in subsequent memory slots. And we don't have low-level memory considerations in FP.

[2]Please note the two ways an array can be declared in Java: either by putting the square brackets on the type (`int[] m`) or on the variable (`int m[]`).

[3]Because we usually count from zero, the first element is `a[0]`, making the sixth `a[5]` and not `a[6]`

1.4 Procedures not Functions

Functions to Procedures

Maths: m(x,y) = xy

ML: fun m(x,y) = x*y;

Java: public int m(int x, int y) = x*y;

```
int y = 7;
public int m(x) {
        y=y+1;
        return x*y;
}
```

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

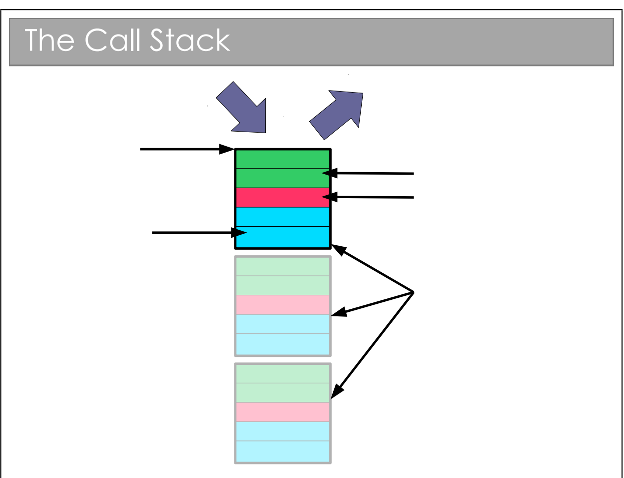
In contrast, the output from a *procedure* can depend on program state that is *not* supplied in the arguments and it can also modify that external state. This is a *side effect* because, given only the procedure name and its arguments, we cannot predict what the state of the system will be after calling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

```
int y=7;

int multiply(int x) {
   y=y+1;
   return x*y;
}
```

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'.

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 functionals. It's just that it introduces a layer of complexity into programming that *might* make the results better but the job harder.

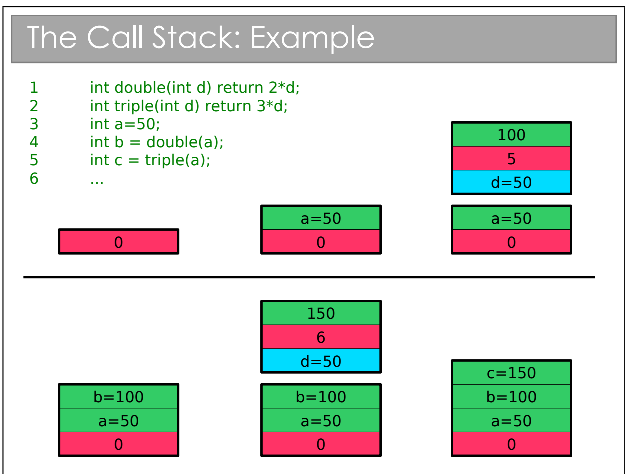
1.4.1 The Call Stack



It's useful to look at how we make use of memory when calling procedures. Remember the CF course: whenever a procedure is called we jump to the machine code for the procedure, execute it, and then jump back to where it was and continue on. This means that, before it jumps to the procedure code, it will have to save where it is somehow.

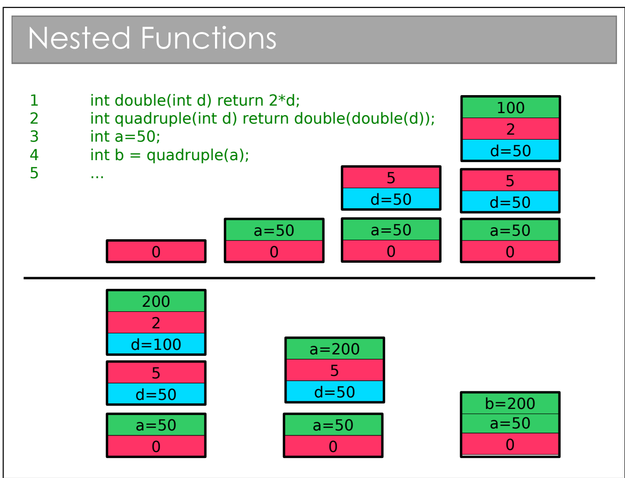
We do this using a *call stack*. A stack is a simple data structure that is the digital analogue of a stack of plates: you add and take from the top of the pile *only*.. By convention, we say that we *push* new entries onto the stack and *pop* entries from its top. Here the 'plates' are called *stack frames* and they contain the function parameters, any local variables the function creates and, crucially, a return address that tells the CPU where to jump to when the function is done. When we finish a procedure, we delete the associated stack frame and continue executing from its return address.

See Algorithms I for a full analysis

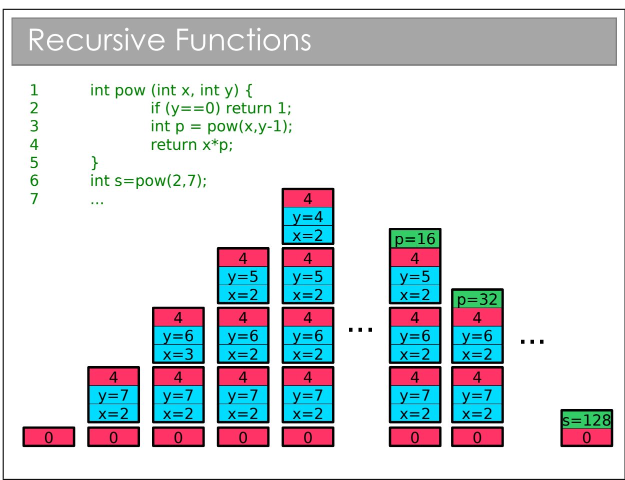


In this example I've avoided going down to assembly code and just assumed that the return address can be the code line number. This causes a small problem with e.g. line 4, which would be a couple of machine instructions (one to get the value of `double{}` and one to store it in `b`). I've just assumed the computer magically "remembers" to store the return value here for brevity.

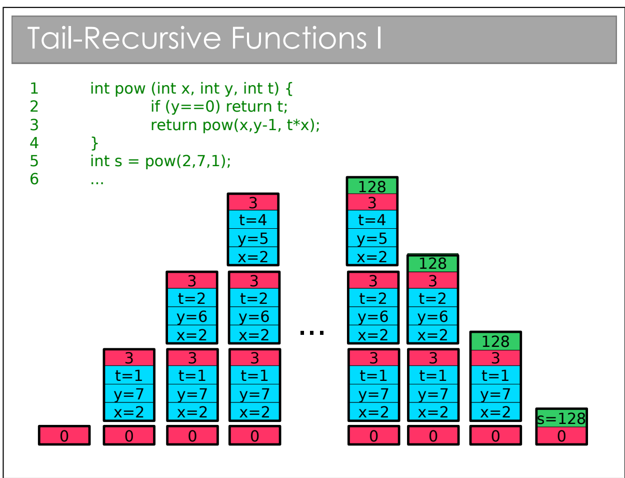
This is pretty simple and the stack never gets very big—things are more interesting if we start nesting functions (i.e. one calls another):



And even more interesting if we start processing recursively:



We immediately see a problem: computers only have finite memory so if our recursion is really deep, we'll be throwing lots of stack frames into memory and, sooner or later, we will run out of space. We call this *stack overflow* and it is an unrecoverable error. You know that tail-recursion does better, but:



There is an easy optimisation to make here, but the compiler has to be developed with that optimisation in mind. The vast majority of imperative compilers *don't* look for tail recursion and so don't optimise for it: Java falls into this camp.

Tail-Recursive Functions II

```
1   int pow (int x, int y, int t) {
2       if (y==0) return t;
3       return pow(x,y-1, t*x);
4   }
5   int s = pow(2,7,1);
6   ...
```

1.5 Control Loops

However, optimised tail-recursion is equivalent to *iteration* and imperative languages support *explicit* iteration through the use of constructs such as for and while.. The following examples iterate exactly eight times.

See Workbook 2

Control Flow: for and while

for(*init; boolean_expression; step* **)**

```
for (int i=0; i<8; i++) ...

int j=0;  for(; j<8; j++) ...

for(int k=7;k>=0; j--) ...
```

while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; ...}

int j=7; while (j>=0) { j--; ...}
```

See workbook 2

You may want to look up the other constructs and keywords for looping. In particular, look at the 'do... while' and 'enhanced for' loops, and the 'break' and 'continue' keywords.

1.6 The Heap

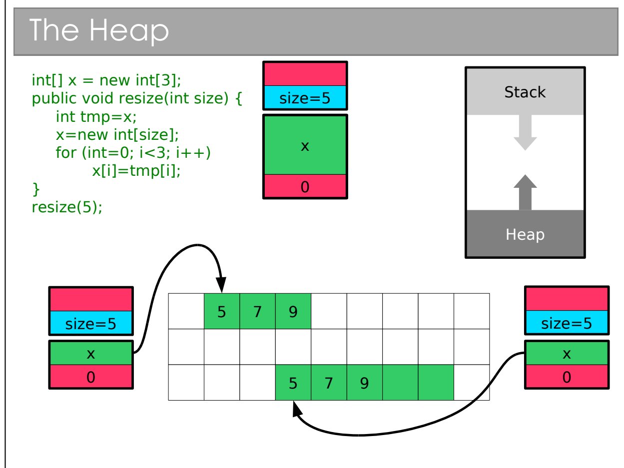
There's a subtlety with the stack that we've passed over until now. What if we want a function to create something that sticks around after it's gone? Or to resize something (say an array)? We talk of memory being *dynamically* allocated rather than *statically* allocated as per the stack.

Why can't we dynamically allocate on the stack? Well, imagine that we do everything on a stack and you have a function that resizes an

array. We'd have to grow the stack, but not from the top, but where the stack was put. This rather invalidates our stack and means that every memory address we have will need to be updated if it comes after the array.

Note: you meet something called a 'heap' in Algorithms I: it is NOT the same thing

We avoid this by using a *heap*. Quite simply we allocate the memory we need from some large pool of free memory, and store a pointer in the stack. Pointers are of known size so won't ever increase. If we can to resize our array, we create a new, bigger array, copy the contents across and update the pointer within the stack.



For those who did the Paper 2 O/S course, you should realise that the heap gets *fragmented*: as we create and delete stuff we leave holes in memory. Occasionally we have to spend time 'compacting' the holes (i.e. shifting all the stuff on the heap so that it's used more efficiently.

1.7 Pointers and References

Looking back at your CF notes, you should recall that pointers (a.k.a. variables holding memory addresses) are immensely useful, but also rather dangerous. We can arbitrarily modify them, either accidentally

or intentionally, and this can lead to all sorts of problems. Although the symptom is usually the same: program crash.

You encountered them (sort-of) in FoCS in the form of references (which I'm sure you all loved). In fact in the FoCS notes it says "...creates references (also called pointers or locations)".

Now, a few decades back, "reference" was synonymous with "pointer" (and to some it always will be). The more modern interpretation distinguishes between them, and that's what we'll do in this course.

1.7.1 References

See workbook 3

References can be seen as a fix for some of the more dangerous aspects of pointers. They are still just variables pointing to chunks of memory, 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 explicit pointers. Whilst slightly limiting, this makes programming much safer (and it's one of the many reasons we teach with Java). Java has two classes of types: *primitive* and *reference*. A primitive type is a built-in type. A reference type is everything else. That includes arrays and (as we will

See Workbook 1

see) objects.

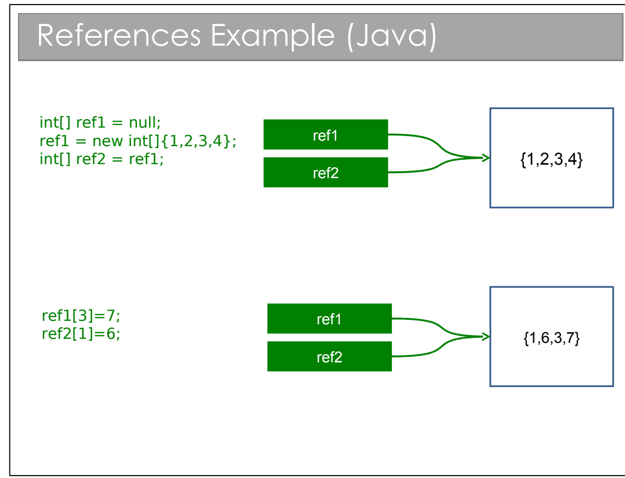
See Workbook 3

References vs Pointers

	Pointers	References
Represents a memory address	Yes	No
Can be randomly assigned	Yes	No
Can be assigned to established object	Yes	Yes
Can be tested for validity	No	Yes

See workbook 3

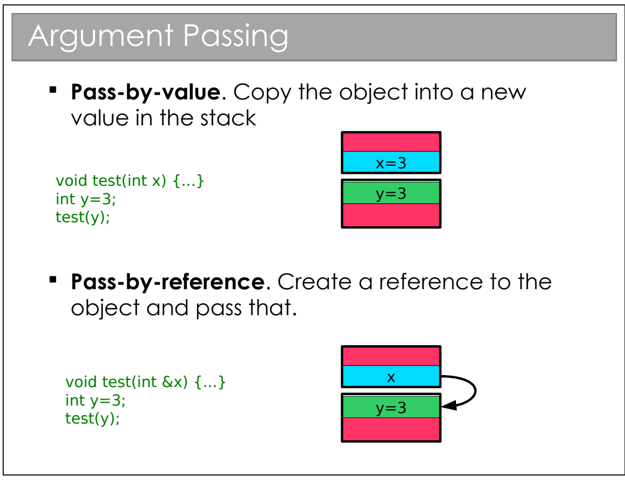
The last point is particularly important. A pointer points to something valid, something invalid, or `null` (a special zero-pointer that indicates it's not initialised). References, however, either point to something valid or to `null`. With a non-null reference, you know it's valid. With a non-null pointer, who knows?



In this example, we create a reference and set it to `null`. Then we create a new array (using the `new` keyword and assign the reference to point to it. Then we create another reference with the same value as `ref1`. i.e. we have two references pointing to the same array in memory.

Thus, when we *dereference* ref1 and make a change, the change will also affect ref2. We will return to this shortly.

1.8 Pass-by-value and Pass-by-reference



Note I had to use C here since Java doesn't have a pass-by-reference operator such as &

When arguments are passed to functions, you may hear it said that that primitive values are "passed by value" and arrays are "passed by reference", but I think this is a rather confusing area for Java. First let's define the terms:

Pass-by-value. The value of the argument is copied into a new argument variable (this is what we assumed in the call stack earlier)

Pass-by-reference. Instead of copying the object (be it primitive of otherwise), we pass a reference to it. Thus the function can access the original and (potentially) change it.

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]);
  }

}
```

This example is taken from your practicals , where you observe the different behaviour of test_i and test_array—the former being a primitive int and the latter being a reference to an array.

See workbook 3

Let's create a model for what happens when we pass a primitive in Java, say an int like test_i. A new stack frame is created and the value of test_i is *copied* into the stack frame. You can do whatever you like to this copy: at the end of the function it is deleted along with the stack frame. The original is untouched.

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, a new stack frame is created. The value of test_array (a memory reference, remember) is copied into a new reference in the stack frame. If we dereference the copy we get to the original so we can change the original. At the end of the function, the frame is deleted, along with the copy. Note we could reassign the copy without effect on the original.

So we can see that java *actually passes all arguments by value*, it's just that arguments are either primitives or references. i.e. Java is strictly pass-by-value.

Don't believe me? See the Java specification, section 8.4.1.

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 (incorrectly) claim. The

examples sheet has a question that explores this further.

Check...

```
public static void myfunction2(int x, int[] a) {
        x=1;
        x=x+1;
        a = new int[]{1};
        a[0]=a[0]+1;
}

public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        myfunction2(num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```

A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

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 ability to specify pass by reference or value from Java!

1.9 Aside: Understanding Java

Since a lot of what you'll do this term will involve Java, now seems like a opportune time to look at the design decisions made in developing it.

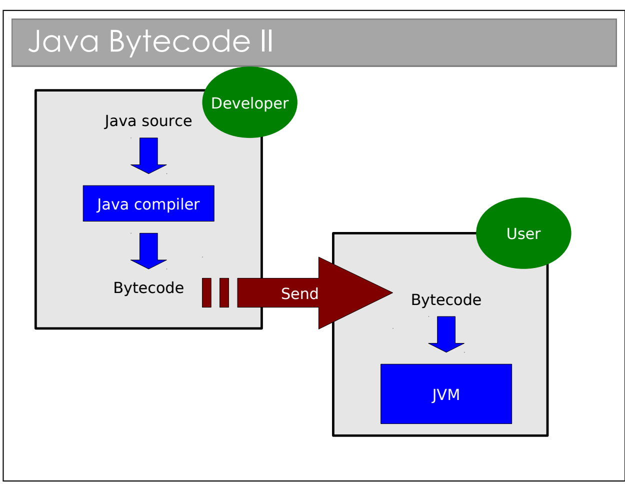
The Java Approach

- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.
- Went for a clever hybrid interpreter/compiler

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.

Java Bytecode I

- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode and spits out the correct machine code for the local computer. i.e. is a bytecode interpreter



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 humans 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".

Java Bytecode III

+ Bytecode is compiled so not easy to reverse engineer

+ The JVM ships with tons of libraries which makes the bytecode you distribute small

+ The toughest part of the compile (from human-readable to computer readable) is done by the compiler, leaving the computer-readable bytecode to be translated by the JVM (→ easier job → faster job)

- Still a performance hit compared to fully compiled ("native") code

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

`http://java.sun.com/docs/books/jvms/`

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 a complete 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

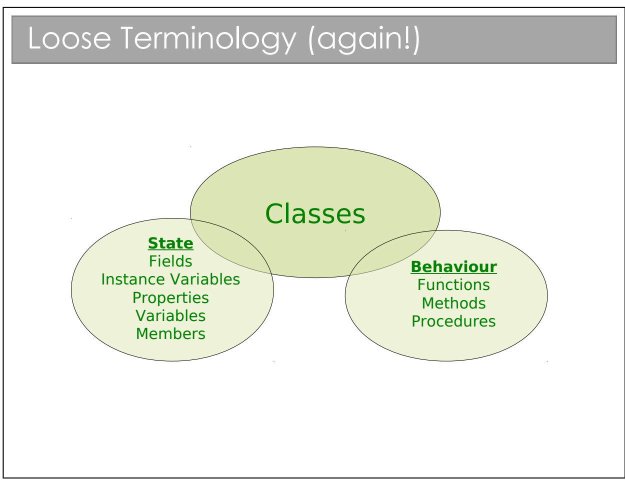
Classes, Instances and Objects II

- Classes are basically templates for various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

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

  makes two instances of class MyCoolClass.
- 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 object of the type should have (a template if you will), whilst each *object* is a specific implementation with set values. So a Person class might specify that a Person has a name and an age. A Person object might represent 40-year old Bob; another might represent 20 year-old Alice.



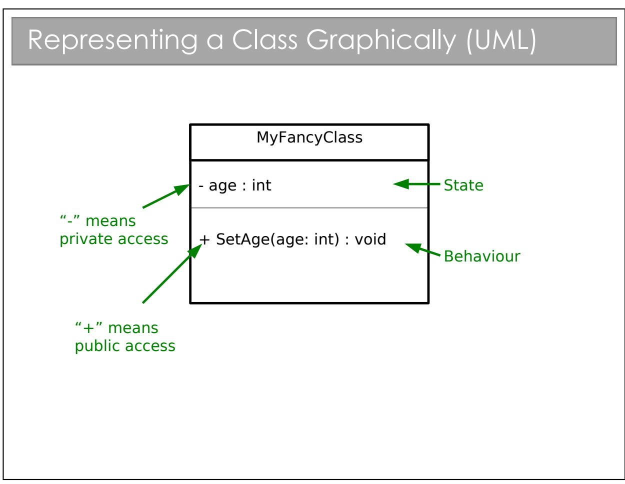
Now, you remember all that fuss we had about 'function' and 'procedure'? Well, it gets worse: when we're talking about a procedure inside a class, it's often called a *method*.

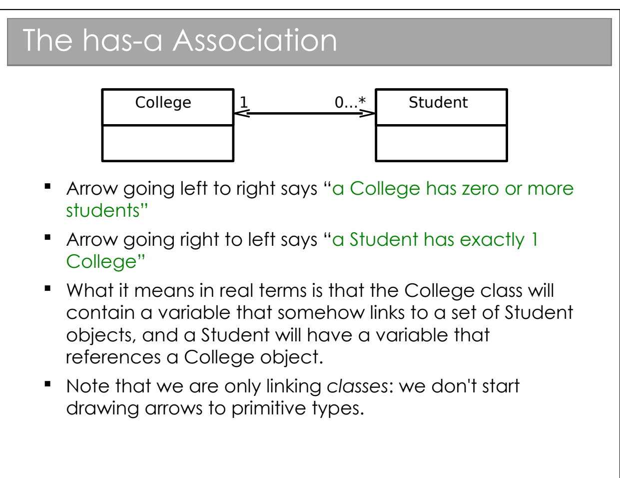
In the real world (which I'm assured does exist), you'll find people use 'function', 'procedure' and 'method' interchangeably. Thankfully you're all smart enough to cope!

Identifying Classes

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

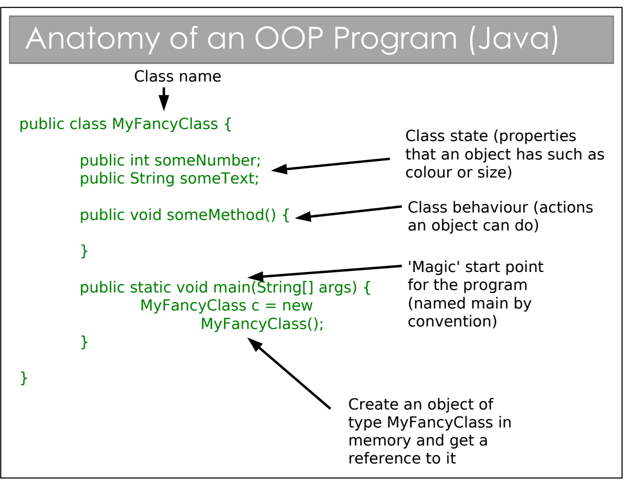
"Write a simulation of the Earth's orbit around the Sun"



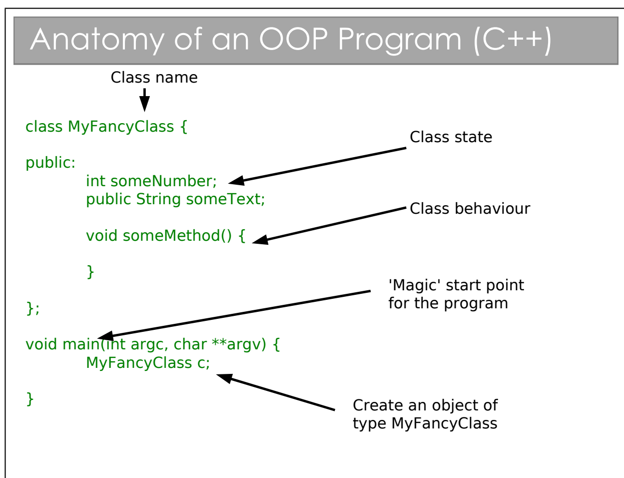


The graphical notation used here is part of UML (Unified Modeling Language). UML is a standardised set of diagrams that can be used to describe software independently of any programming language used to implement it.

UML contains many different diagrams (touched on in the Software Design course for those doing Paper 2). Here we just use the *UML class diagram* such as the one in the slide.



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



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

See workbook 3

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**.

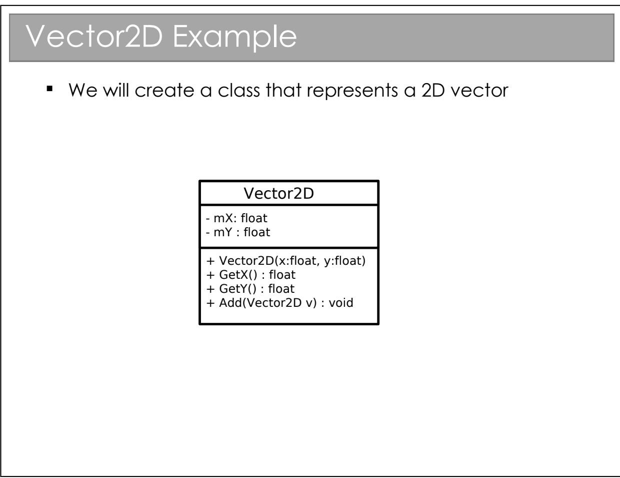
It's also related to ***cohesion***. A highly cohesive class contains only a set of strongly-related functions rather than being a hotch-potch of functionality. We strive for **high cohesion**.

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 a **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.



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;
}
```

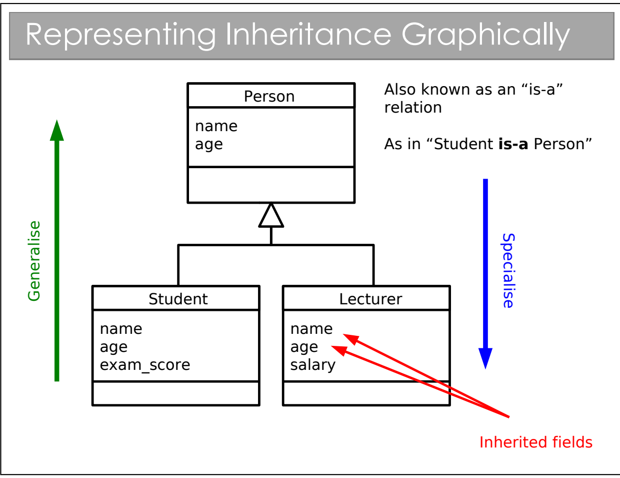
- There is a lot of duplication here
- Conceptually there is a hierarchy that we're not really representing
- Both Lecturers and Students are people (no, really).
- We can view each as a kind of specialisation of a general person
  - They have all the properties of a person
  - But they also have some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



See workbook 5

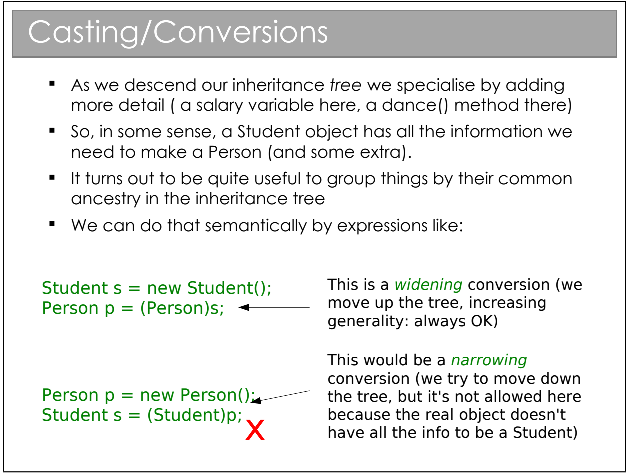
Inheritance is an extremely powerful concept that is used extensively in good OOP. We has discussed the “has-a” relation amongst classes; inheritance adds the “is-a” concept. E.g. A car *is a* vehicle that *has a* steering wheel.

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!

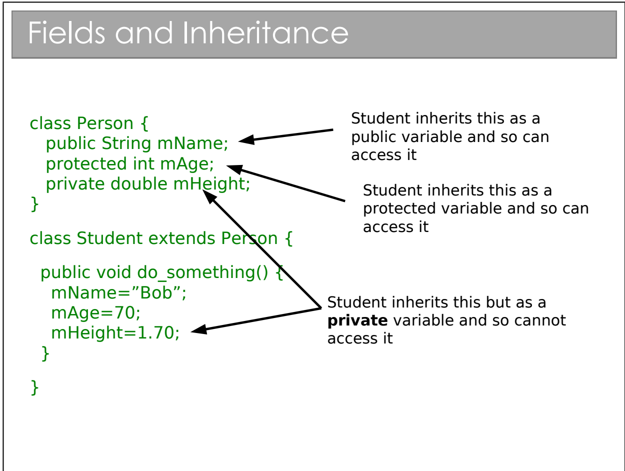


One way to think about this is that when we create a new Student the computer allocates a chunk of memory for that object. In that chunk there is space for all of the fields (name, age, exam score, etc) and definitions of the methods it supports. Because a Student is a Person, that chunk will contain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



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 genuinely being shadowed and nothing is being replaced. This is contrast to the behaviour with methods...

NB: A common novice 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 {
```

[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.

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

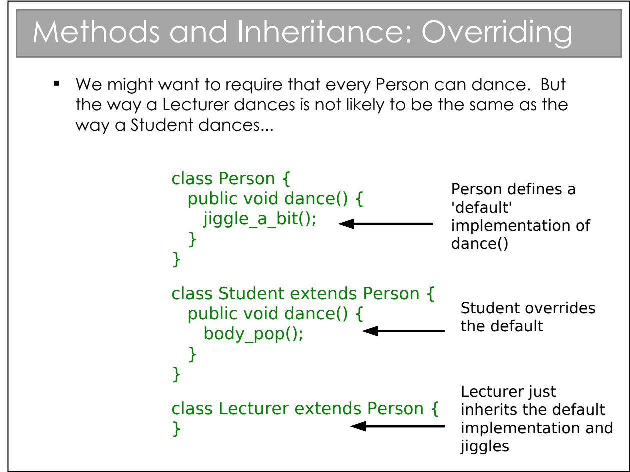
becomes:

```
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?



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.

Polymorphic Methods

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

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

- General problem: when we refer to an object via a parent type and both types implement a particular method: which method should it run?

Polymorphic Concepts I

- **Static** polymorphism
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we must just run the parent method
  - Type errors give compile errors

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

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

In general static polymorphism[3] refers to anything were decisions are made at compile-time. You may realise that all the polymorphism you

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

saw in ML was static polymorphism. The shadowing of fields also fits this description.

Polymorphic Concepts II

- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

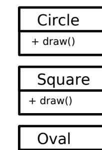
```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in Student

This form of polymorphism is OOP-specific and is sometimes called *subtype* or *ad-hoc* polymorphism. It's crucial to good, clean OOP code. Because it must check types at run-time, there is a performance overhead associated with dynamic polymorphism. however, as we'll see, it gives us much more flexibility and can make our code more legible.

Beware: Most programmers use the word 'polymorphism' to refer to dynamic polymorphism.

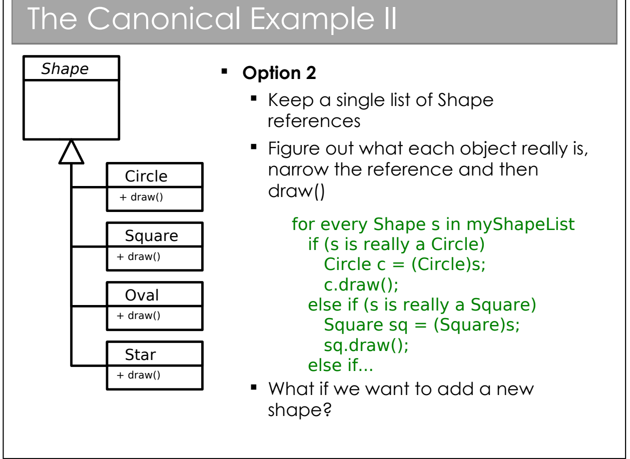
The Canonical Example I

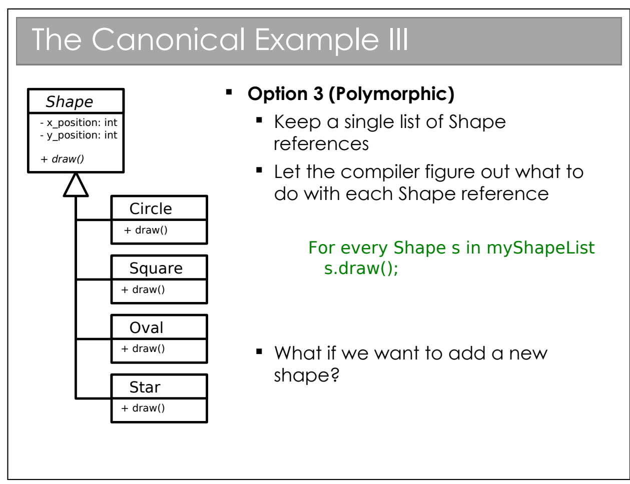


Oval
+ draw()

Star
+ draw()

- 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?





Implementations

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

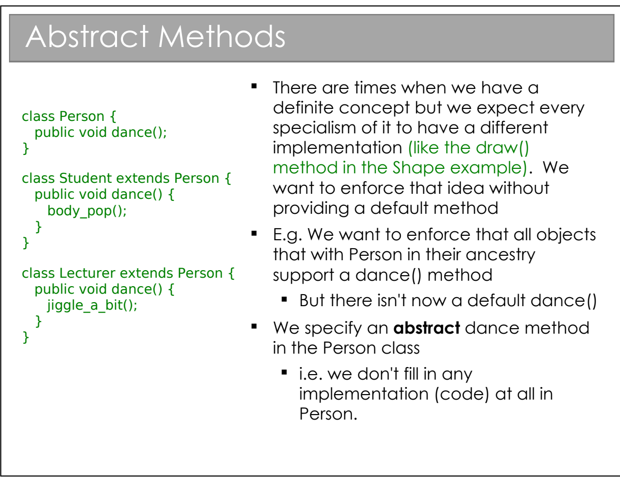
- Polymorphism in OOP is an extremely important concept that you need to make sure you understand...

C++ allows you to choose whether methods are inherited statically (default) or dynamically (explicitly labelled with the keyword 'virtual'). This can be good for performance (only incur the overhead when you need to), but gets complicated, especially if the base method isn't dynamic but a derived method is...

The Java designers avoided the problem by enforcing dynamic polymorphism. You may find reference to static polymorphism being possible

in Java by declaring a method final. Then it is a compile error to try to override it in subclasses. To me, this isn't quite the same: it's not making a choice between multiple implementations but rather enforcing that there can only be one implementation!

2.3 Abstract Classes



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:

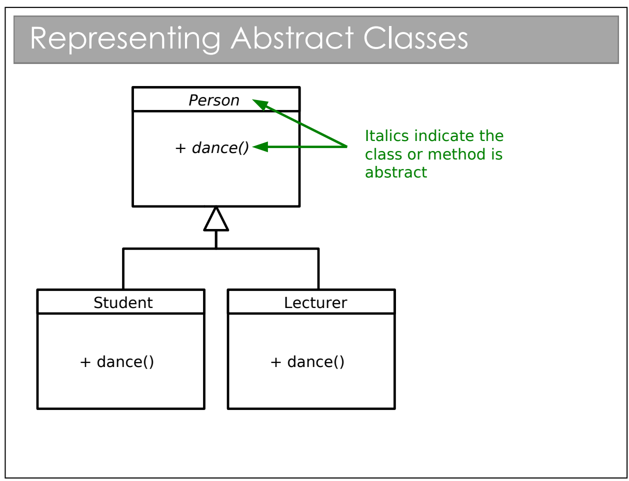
Java:
```
public abstract class Person {
  public abstract void dance();
}
```

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

- 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!!

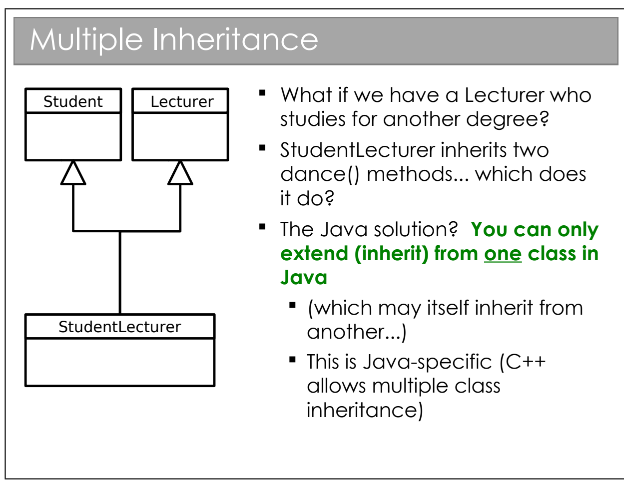
Abstract classes allow us to partially define a type. Because it's not fully defined, you can't make an object from an abstract class (try it). Only once all of the 'blanks' have been filled in can we create an object from it. This is particularly useful when we want to represent high level concepts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



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.

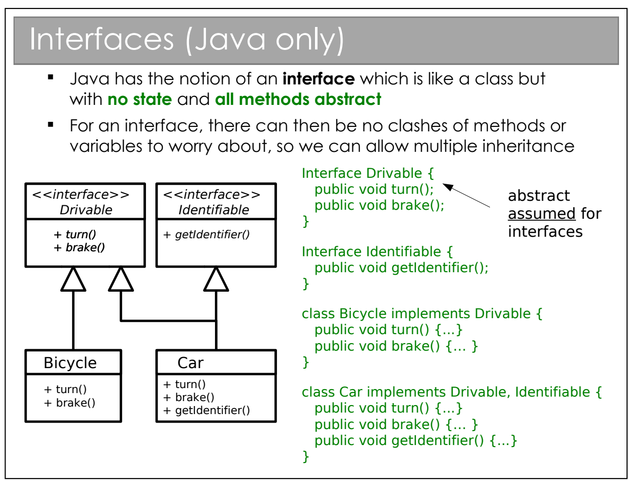
2.4 Interfaces



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).

In this example StudentLecturer would inherit two different dance() methods, so which should it do when instructed to dance()? The answer is that the programming language has to support some way of explicitly choosing the method to run. This can get messy and rather complicated. The Java designers decided that, if you ever find yourself in such a situation, you could redesign all your classes to avoid multiple inheritance. To enforce this, they took away the choice: a Java class can *only* inherit from one parent class.

Generally speaking, this does result in clearer code. There are certainly examples where multiple inheritance is the most concise and attractive solution—in those cases the Java equivalent seems messy and convoluted. However, such cases are rare.



See workbook 5

Interfaces are so important to Java they are considered to be the third reference type (the other two being classes and arrays). Using interfaces encourages high abstraction level in code, which is generally a good thing since it makes the code more flexible/portable. However, i is possible to 'overdo' it, ending up with 20 files where one would do...

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

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

Java

```
public class Person {
  private String mName;

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

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

}
```

C++

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

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

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

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, Java fills in an empty one for you
- Here it creates Person() for us
- The default constructor takes no arguments (since it wouldn't know what to do with them!)

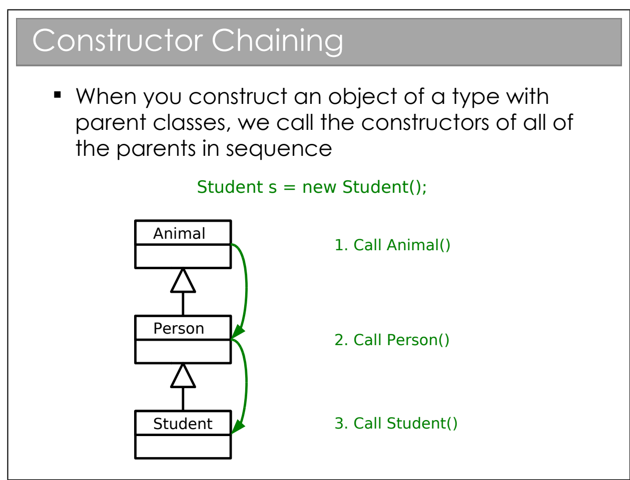
Every class has a constructor. The only question is whether it's been specified manually by the programmer or whether the compiler has filled in a default (empty) constructor.

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)

Beware: As soon as you specify *any* constructor whatsoever (regardless of the arguments), no default constructor will be generated. The default constructor only applies when the compiler notices that there is no way to construct an object of this type, which can't be intentional or what's the point of writing the class?



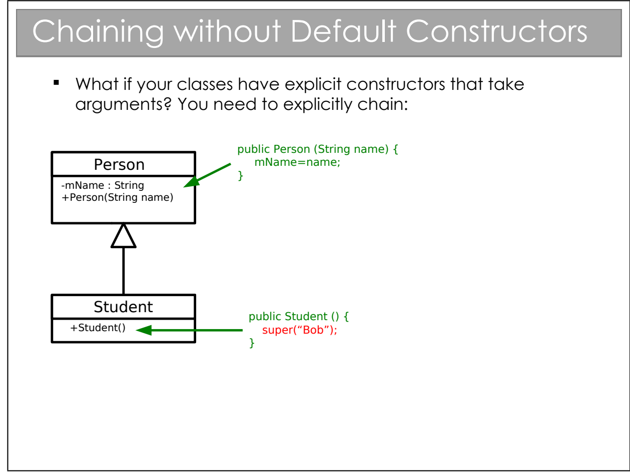
What actually happens is that the first line of a constructor *always* starts with super(), which is a call to the parent constructor (which

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.

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

C++

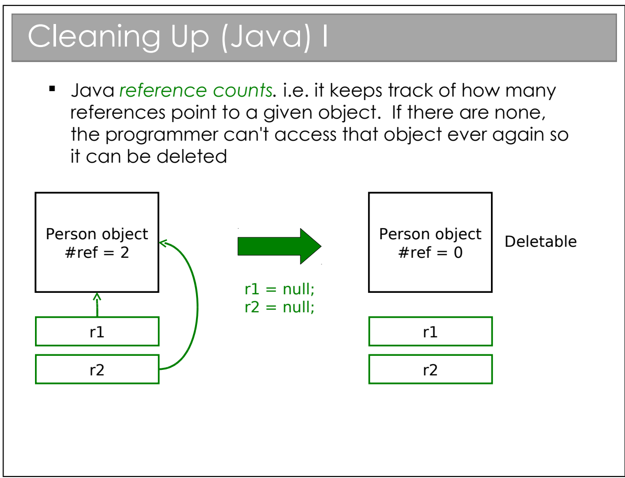
```
class FileReader {
  public:

    // 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;

}
```

It will shortly become apparent why I used C++ and not Java for this example.

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?
    - A "memory leak"
- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object will never be used again and can be deleted??



Note that reference counting has an associated cost - every object needs more memory (to store the reference count) and we have to monitor changes to all references to keep the counts up to date.

Cleaning Up (Java) II

- Actual deletion occurs through a **garbage collector**
  - A separate process that periodically scans the objects in memory for any with a reference count of zero, which it then deletes.
  - Running the garbage collector is obviously not free. If your program creates a lot of short-term objects, you will soon notice the collector running
    - Gives noticeable pauses to your application while it runs.
    - But minimises memory leaks (it does not prevent them...)

Cleaning Up (Java) III

- One problem with GC is we have no idea *when* an object will actually be deleted. The GC may even decide to defer the deletion until a future run.
- This causes issues for destructors – it might be ages before a resource is closed and available again!
- Therefore Java doesn't have destructors
- It does have **finalizers** that gets run when the GC deletes an object
  - BUT there's no guarantee an object will ever 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 = 0.2;

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

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

}
```

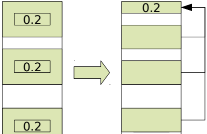
- This is one solution to incorporating VAT into a shop application
- Bad: Every instance will contain a float with the same number
- Bad: If the VAT rate changes, how can we be sure every single object with such a float is properly changed?!

- It can be useful to have *class variables* a.k.a. *static variables*. These are variables that exist per class and not per object
- Create them in Java using 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 a static method 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 methods?

- 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 does 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 to handle 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 return values are meant to signify, etc.
  - The actual result often can't be returned in the same way

In some cases the range of potential results is smaller then the range of return values, in which case we can use the 'spare' values to signify error. E.g. If we know the result will be positive, we can use -1 to signify an error. Whilst this works (and is extremely common in C), it means that we have to write 10 or more lines of error checking code for every line of meaningful program!

Exceptions I

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

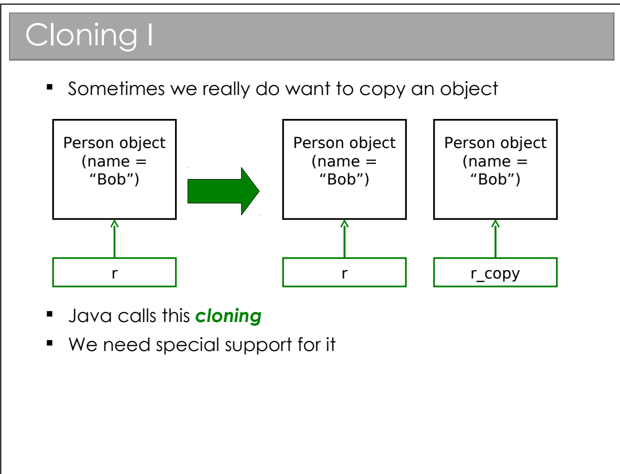
- Advantages:
  - Class name can be descriptive (no need to look up error codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only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

See workbook 4

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

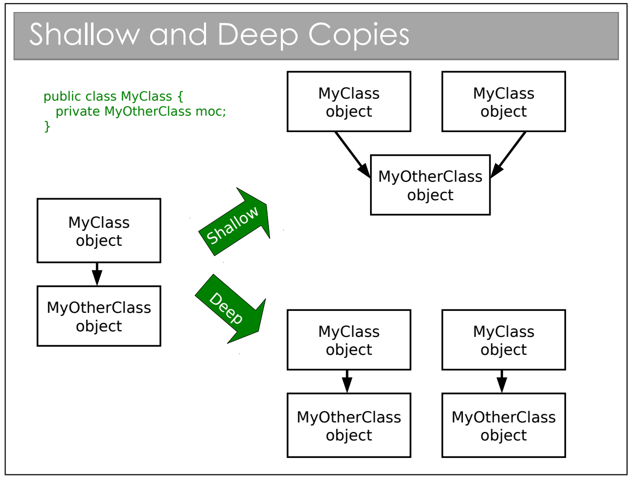


Cloning II

- Every class in Java ultimately inherits from the **Object** class
  - This class contains a clone() method so we just call this to clone an object, right?
  - This can go horribly wrong if our object contains reference types (objects, arrays, etc)

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...



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:

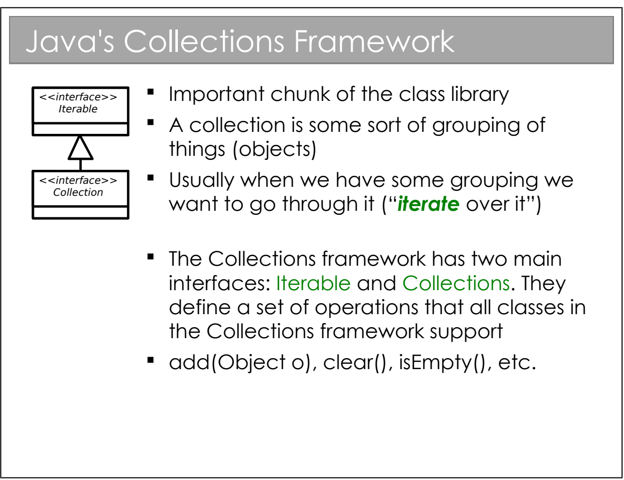
- Complex data structures and algorithms
- I/O (input/output: reading and writing files, etc)

- Networking
- Graphical interfaces

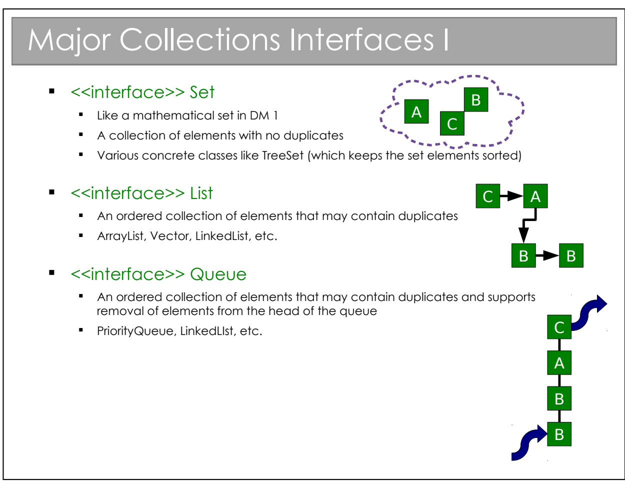
Of course, most programming languages have built-in classes, but Java has a big advantage. Because Java code runs on a virtual machine, the underlying platform is abstracted away. For C++, for example, the compiler ships with a fair few data structures, but things like I/O and graphical interfaces are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

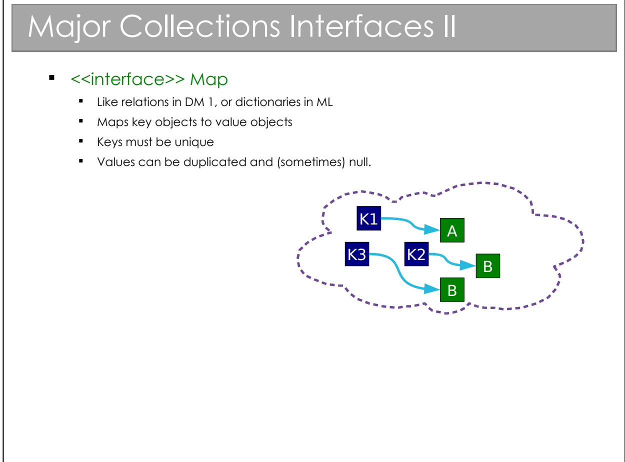
There is, then, good reason to take a look at the Java class library to see how it is structured.

3.0.1 Collections and Generics



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).





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.

Iteration

- for loop

```
LinkedList list = new LinkedList();
...
for (int i=0; i<list.size(); i++) {
   Object next = list.get(i);
}
```

- foreach loop (Java 5.0+)

```
LinkedList list = new LinkedList();
...
for (Object o : list) {

}
```

The foreach notation works for arrays too and it's particularly neat when we have nested iteration. E.g. iteration over all students and their subjects:

```
for (Student stu : studentlist)
  for (Subject sub : subjectlist)
    getMarks(stu, sub);
```

versus:

```
for (int i=0; i<studentlist.size(); i++) {
  Student stu = (Student)studentlist.get(i);
  for (int j=0; i<subjectlist.size(); i++) {
    Subject sub = (Subject)subjectlist.get(j);
    getMarks(stu, sub);
  }
}
```

Iterators

- What if our loop changes the structure?

```
for (int i=0; i<list.size(); i++) {
   If (i==3) list.remove(i);
}
```

- Java introduced the Iterator class

```
Iterator it = list.iterator();

while(it.hasNext()) {Object o = it.next();}

for (; it.hasNext(); ) {Object o = it.next();}
```

- Safe to modify structure

```
while(it.hasNext()) {
   it.remove();
}
```

Note that the foreach structure isn't useful with Iterators. So we sacrifice some code readability for the ability to adjust the Collection's structure as we go.

Collections and Types I

```
// 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;
}
```

- The original Collections framework just dealt with collections of Objects
  - Everything in Java "is-a" Object so that way our collections framework will apply to any class
  - But this leads to:
    - Constant casting of the result (ugly)
    - The need to know what the return type is
    - Accidental mixing of types in the collection

Collections and Types II

```
// 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)

Java Generics

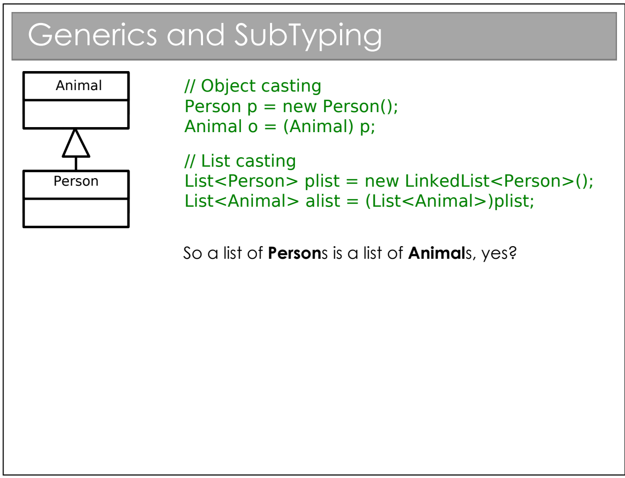
- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

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

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));      <- Won't even compile

// Loop through
iterator<Integer> it = ts.iterator();
while(it.hasNext()) {
    Integer i = it.next();      <- No need to cast :-)
}
```

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 sounds a lot like static polymorphism. And so it is—although it's a special form of it known as *parametric* polymorphism. If you think about it, it maps almost directly to what you called polymorphism in ML...

Generics Declaration and Use

```
public class Coordinate <T> {
   private T mX;
   private T mY;

   public Coordinate(T x, T y) {
      mX=x; mY=y;
   }

   public T getX() { return mX; }
   public T getY() { return mY; }
}

Coordinate<Double> c =
   New Coordinate<Double>(1.0,1.0);

Double d = c.getX();
```



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 (ref1==ref2) do??
  - Test whether they point to the same object?
  - Test whether the objects they point to have the same state?

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 directly*

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

(p1==p2);   <- False (references differ)
(p1!=p2);   <- True (references differ)
(p1==p1);   <- True
```

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

```
public EqualsTest {
  public int x = 8;

  public boolean equals(Object o) {
    EqualsTest e = (EqualsTest)o;
    return (this.x==e.x);
  }

  public static void main(String args[]) {
    EqualsTest t1 = new EqualsTest();
    EqualsTest t2 = new EqualsTest();
    System.out.println(t1==t2);
    System.out.println(t1.equals(t2));
  }
}
```

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, where I trust it to have been done properly.

Option 3: Comparable<T> Interface I

int compareTo(T obj);

- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- 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)

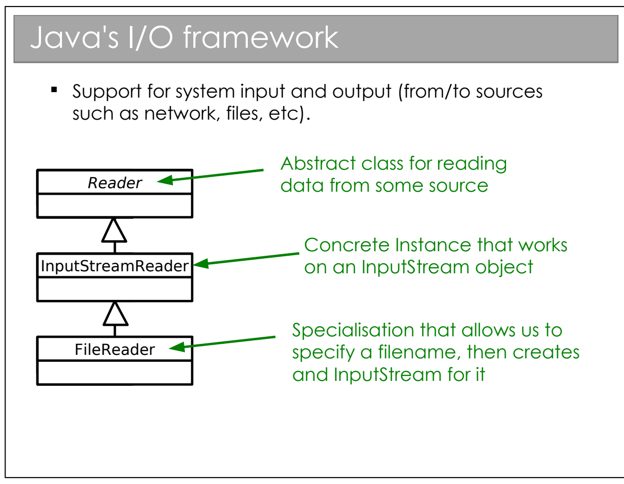
- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections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

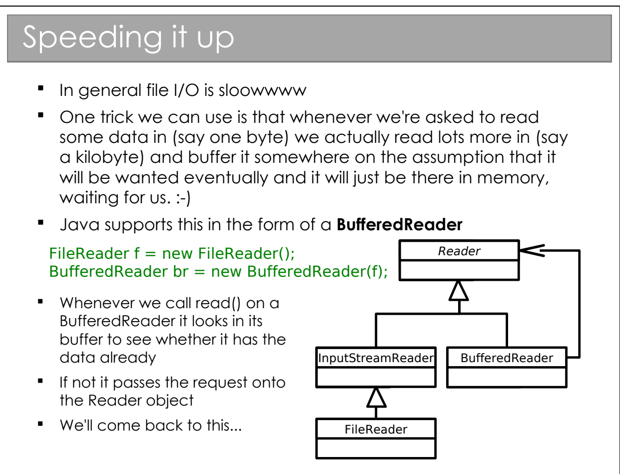
At first glance, it may seem that Comparator doesn't add much over Comparable. However it's very useful to be able to specify Comparators and apply them dynamically to Collections. If you look in the API, you will find that Collections has a *static* method sort(List l, Comparator c).

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





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're about to talk about. Come back to this bit when you've done the whole course!

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

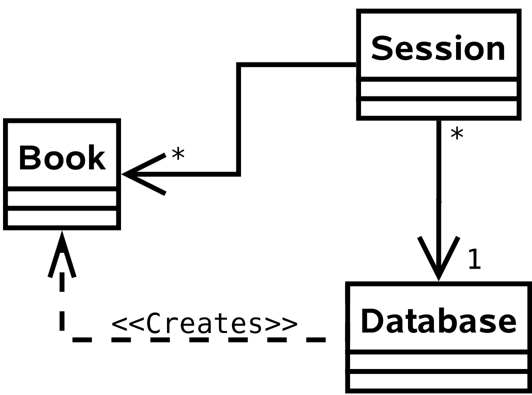
The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments[1] with a single word, the code becomes more readable and maintainable. Furthermore, you can insert the word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

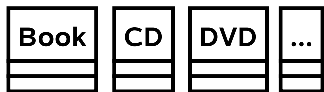
Database. This class wraps around our database, hiding away the query syntax (i.e. the SQL statements or similar).

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

4.3 Supporting Multiple Products

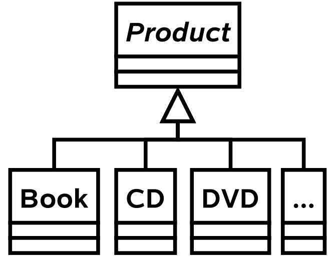
Problem: Selling books is not enough. We need to sell CDs and DVDs too. And maybe electronics. Oh, and sports equipment. And...

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.

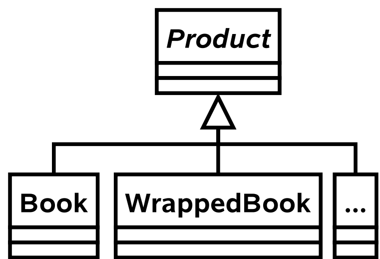
4.4 The Decorator Pattern

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.

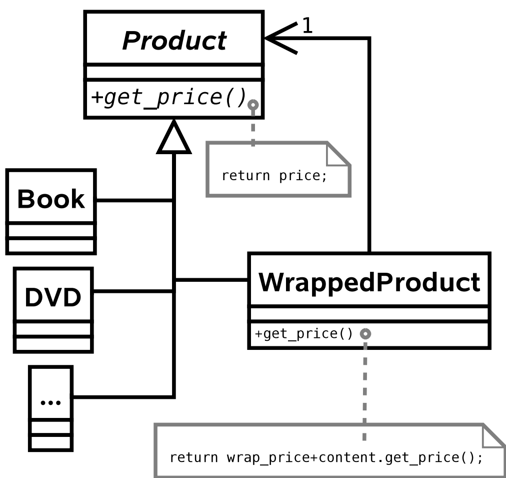
Solution 2: Add WrappedBook (etc.) as subclasses of Product as shown.



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.

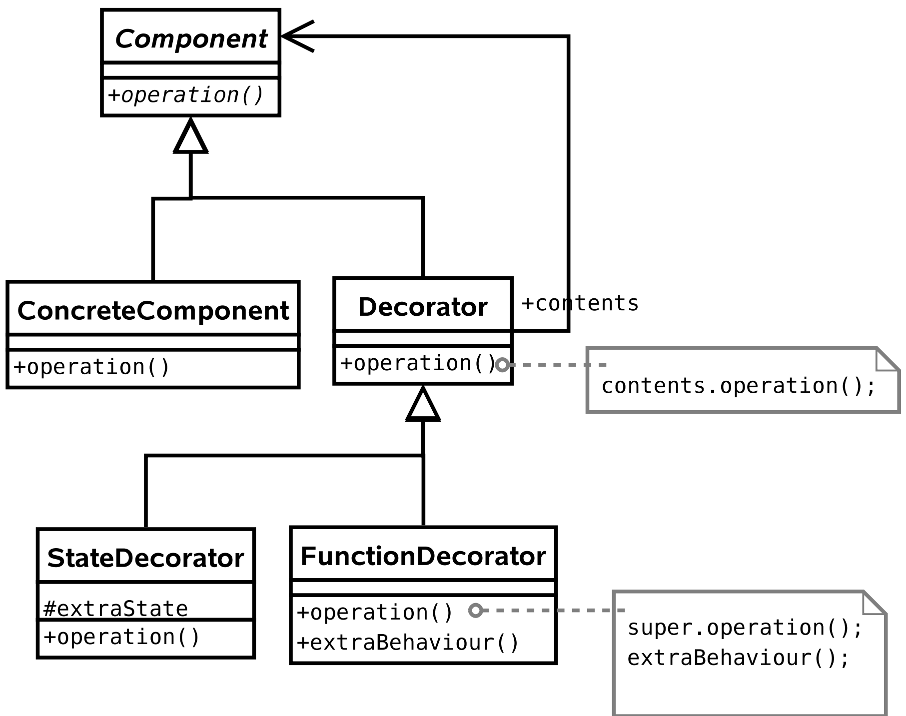
Solution 3: Create a general WrappedProduct class that is both a subclass of Product and references an instance of one of its siblings. Any state or functionality required of a WrappedProduct is 'passed on' to its internal sibling, unless it relates to wrapping.



- ✔ 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

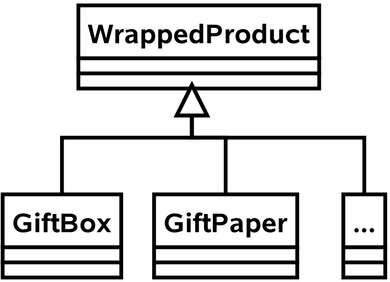
Problem: We need to handle a lot of gift options that the customer may switch between at will (different wrapping papers, bows, gift tags, gift boxes, gift bags, ...).

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 ...
}
```

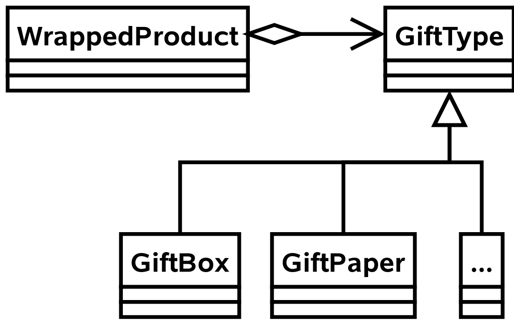
- ✔ It works.
- ✗ The code is far less readable.
- ✗ Adding a new wrapping option is ugly.

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!

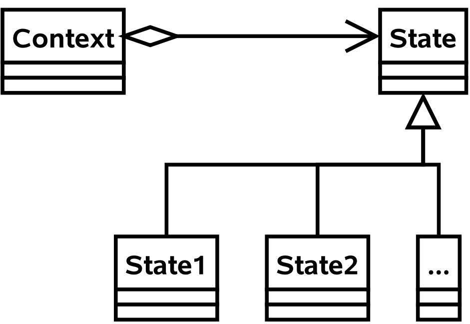
Solution 3: Let's keep our idea of representing states with a class hierarchy, but use a new abstract class as the parent:



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?

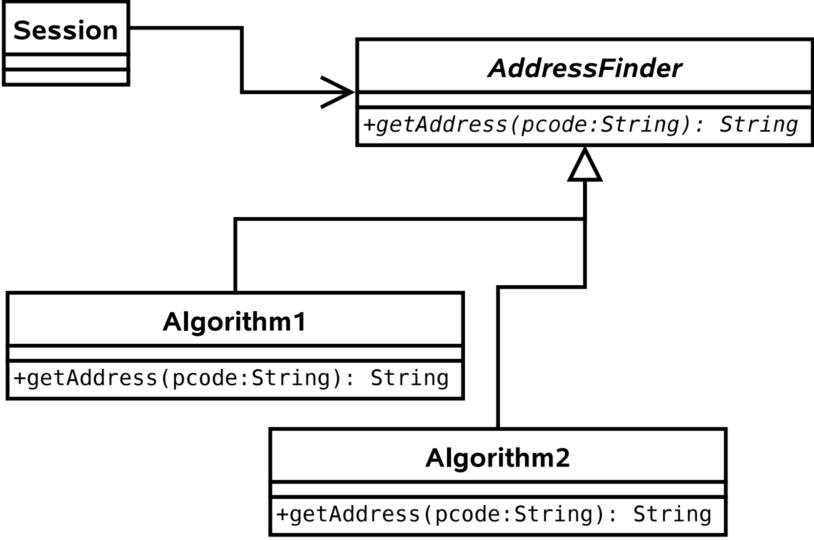
Solution 1: Let there be a class AddressFinder with a method getAddress(String pcode). We could add lots of if/then/else statements to the getAddress() function.

```
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].

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

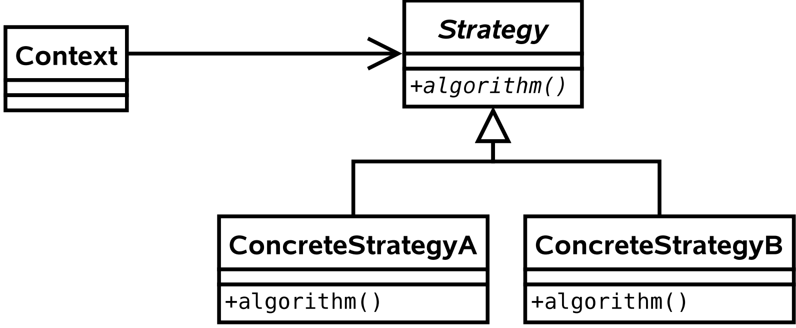
[2]This states that a class should be open to extension but closed to modification. So we allow classes to be easily extended to incorporate new behavior without modifying existing code. This makes our designs resilient to change but flexible enough to take on new functionality to meet changing requirements.



- ✔ 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.

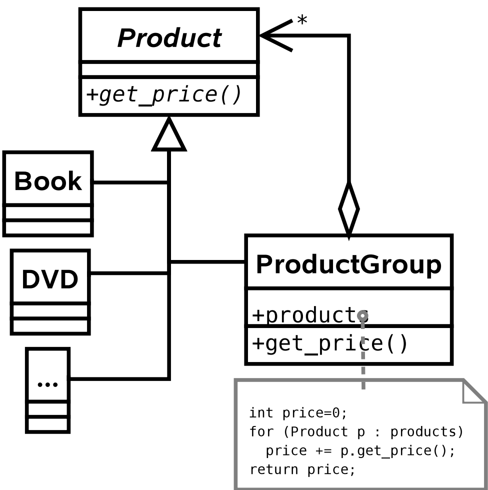
4.7 Composite Pattern

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:



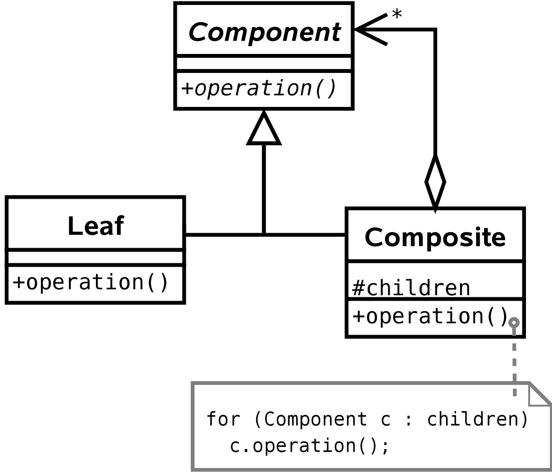
If you're still awake, you may be thinking this is a bit like the **Decorator** pattern, except that the new class supports associations with multiple

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.

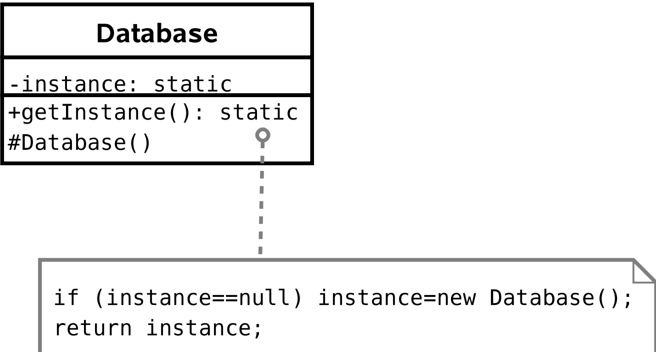
```
public class System {
  public System(Database d) {...}
}

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.

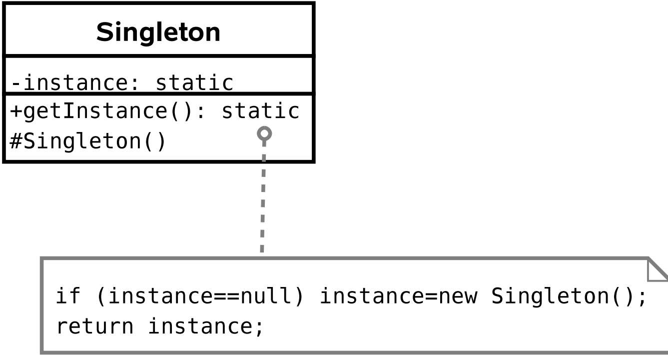
Of course, nothing stops a programmer from ignoring the getInstance() function and just creating a new Database object. So we use a neat trick: we make the constructor *private* or *protected*. This means code like new Database() isn't possible from an arbitrary class.



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

4.8.1 Generalisation

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



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

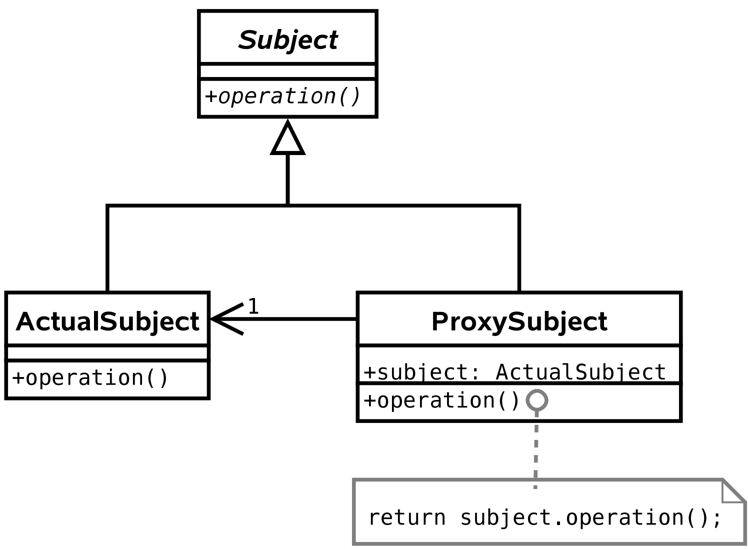
Protected members are accessible to the class, any subclasses, *and all classes in the same package*. Therefore, any class in the same package as your base class will be able to instantiate Singleton objects at will, using the new keyword!

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

4.9 Proxy Pattern(s)

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

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



4.9.1 Virtual Proxy

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

Solution : Here we apply the **Proxy** pattern by only loading part of the full class into the proxy class (e.g. name and price). If someone 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.

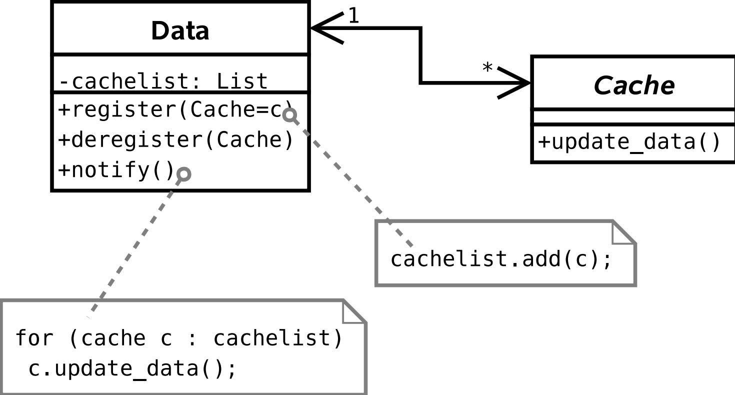
4.10 Observer Pattern

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).

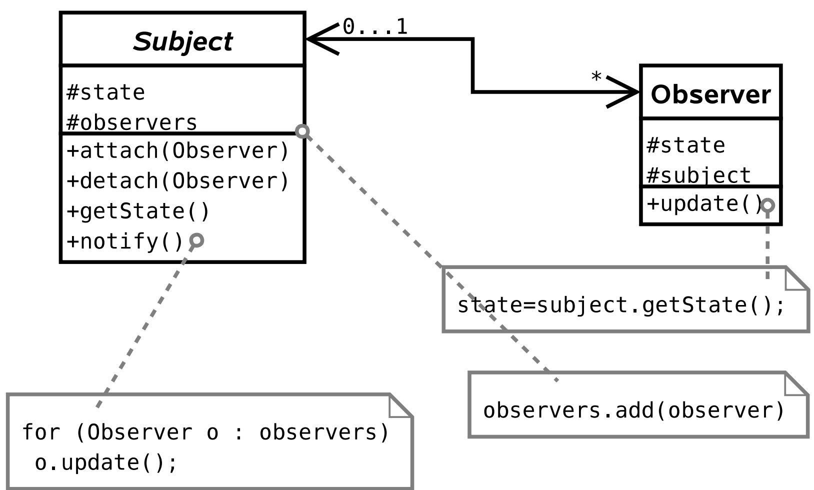
- ✗ How frequently should we poll? Too quickly and we might as well not have cached at all. Too slow and changes will be slow to propagate.

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.



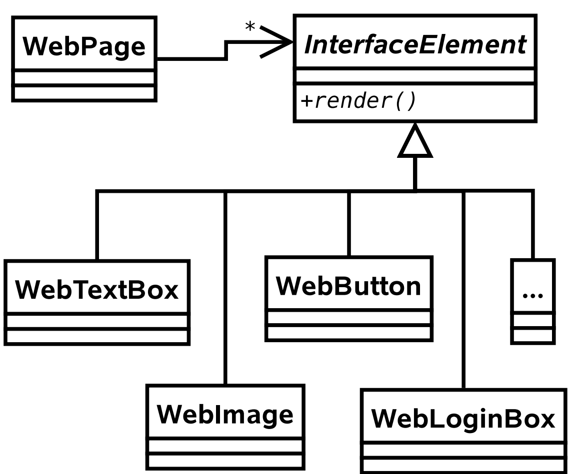
4.10.1 Generalisation

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.



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:



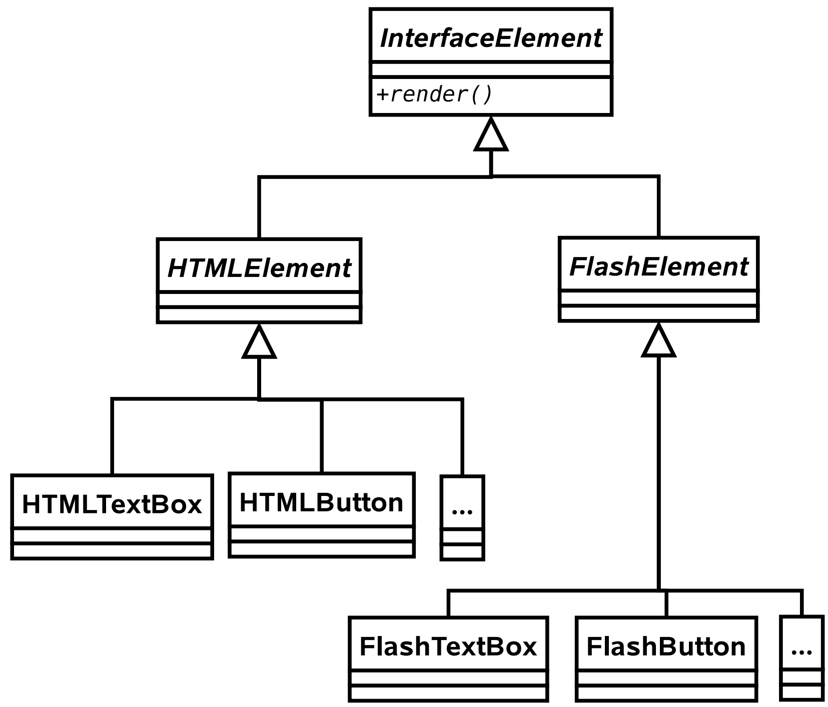
Let's assume that there is a render() method that generates some HTML which can then be sent on to web browsers.

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.

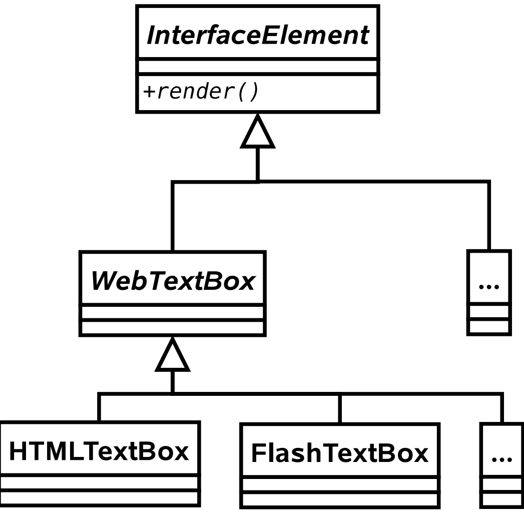
- ✔ Works.
- ✗ The **State** pattern is designed for a single object that regularly changes state. Here we have a family of objects in the same state (Flash, HTML, etc.) that we choose between at compile time.
- ✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

Solution 2: Create specialisations of InterfaceElement:



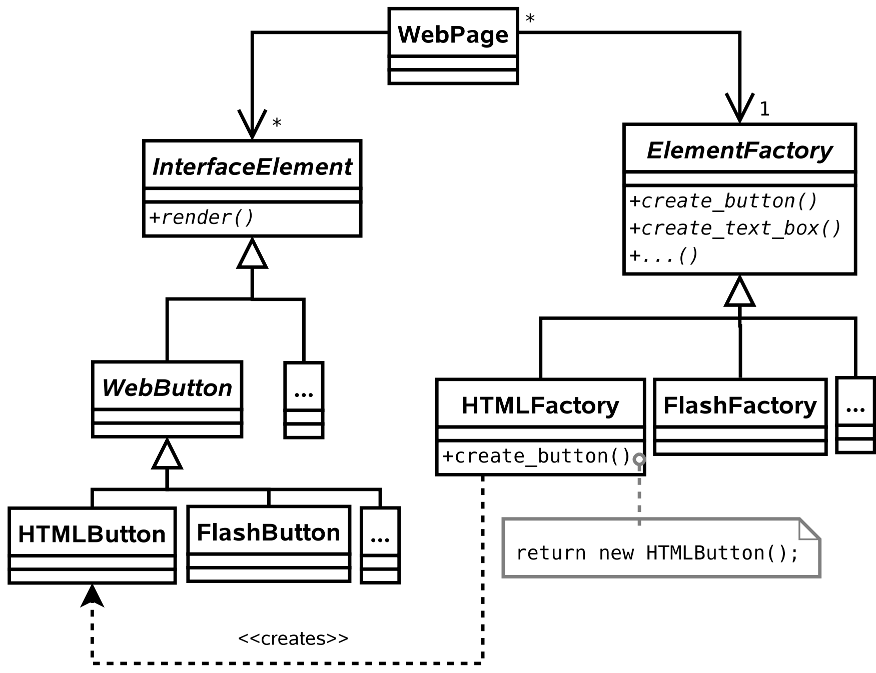
- ✗ Lots of code duplication.
- ✗ Nothing keeps the different TextBoxes in sync as far as the interface goes.
- ✗ A lot of work to add a new interface component type.
- ✗ Doesn't stop us from mixing FlashButton with HTMLButton, etc.

Solution 3: Create specialisations of each InterfaceElement subclass:



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

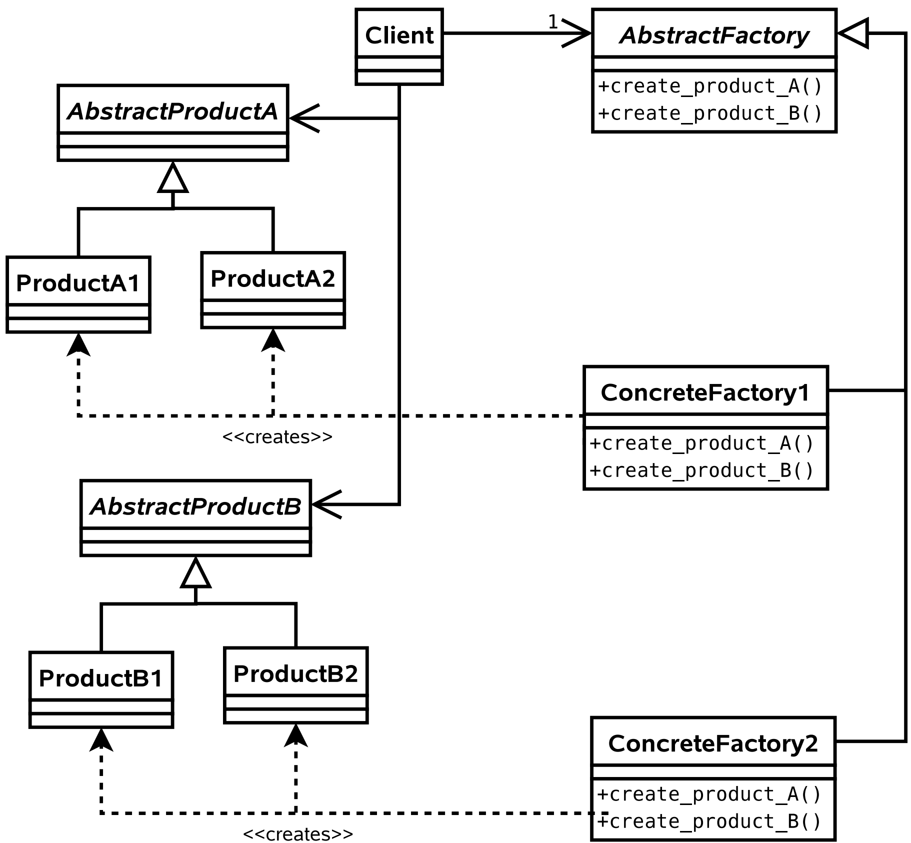
Solution 4: Apply the **Abstract Factory** pattern. Here we associate every WebPage with its own 'factory' — an object that is there just to make other objects. The factory is specialised to one output type. i.e. a FlashFactory outputs a FlashButton when create_button() is called, whilst a HTMLFactory 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.