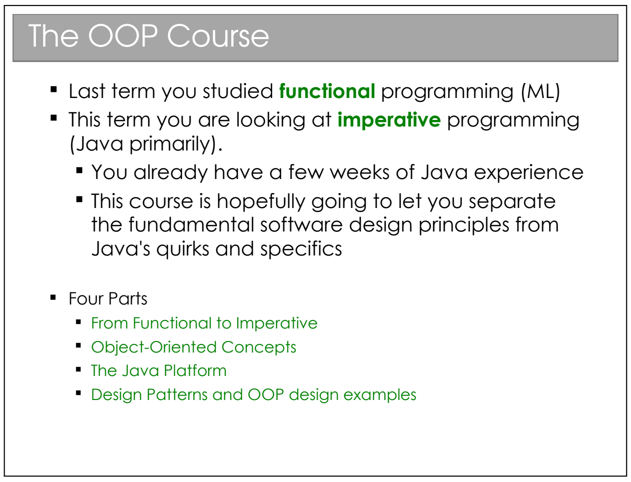
Object Oriented Programming
Dr Robert Harle

IA CST, PPS (CS) and NST (CS)

Lent 2011/12

The Course



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

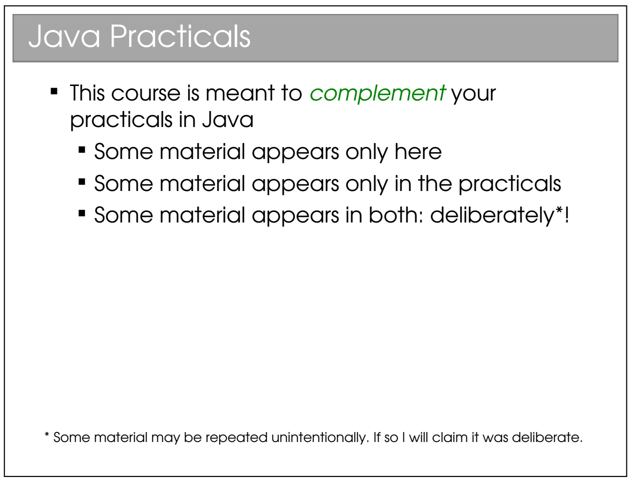
Last term you learnt to program using the functional programming language ML. As we discussed in the Computer Fundamentals course, there are many reasons we started with this, chief among them being that everything is a well-formed *function*, by which we mean that the output is dependent *solely* on the inputs (arguments). This generally makes understanding easier. In fact, if you try any other functional language you'll probably discover that it's very similar to ML in many respects and translation is very easy. This is a consequence of functional languages having very carefully defined 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 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 more uniformity arising. Imperative programming is much more flexible[1] and, crucially, not all imperative languages support all of the same language concepts 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. Consequently 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. This course was introduced to try to address this.

[1] some would say it gives you more rope to hang yourself with!

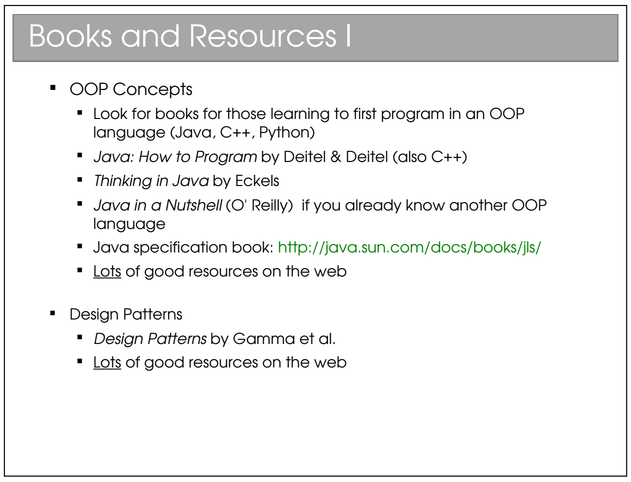
The 'examinable' OOP language for IA Computer Science is Java, and you won't be expected to program in anything else. However, Java doesn't support everything we'll be looking at so other languages will be used to demonstrate certain features. For those of you continuing in the Natural Sciences Tripos next year, you'll probably need to get to grips with C++, so I will make that a nominal second language here (albeit not examinable). We may also find time to try out Python and some other popular languages.



Java Practicals

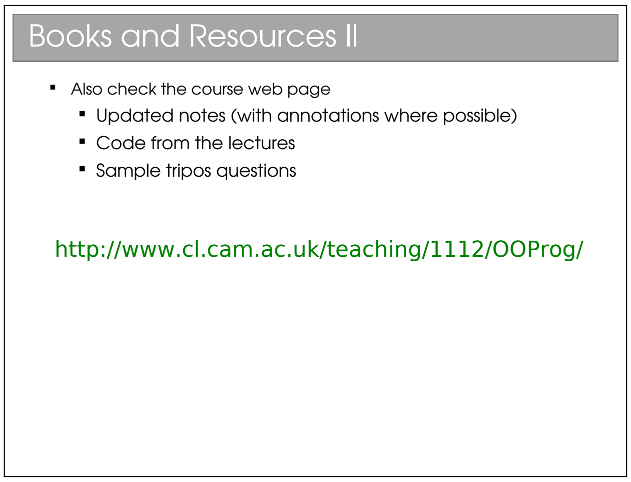
- This course is meant to *complement* your practicals in Java
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately*!

* 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 gained from this course may help you with your ticks!)

Lecture 1

From ML to Java

There are many differences between ML and Java. Here we highlight some key points that might help you with the transition.

1.1 Functional → Imperative

Moving from ML to Java is fundamentally a shift from *functional* programming to *imperative* programming. We met the concepts in Computer Fundamentals, where we saw that functional languages are a subclass of what is called *declarative* languages. As a brief recap we left it at:

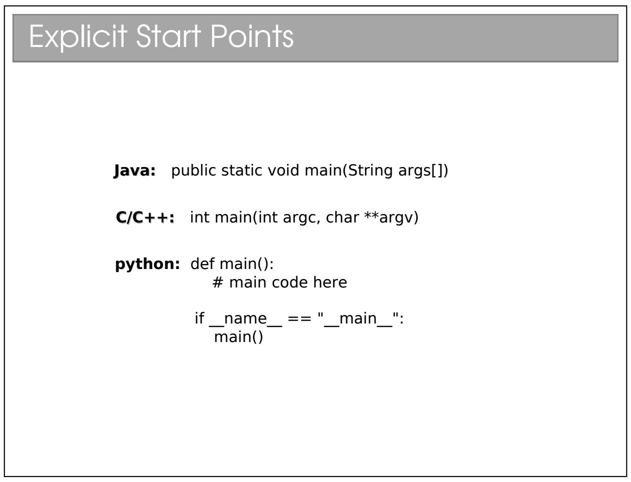
Declarative languages specify *what* should be done but not necessarily *how* it should be done. You know an ML compiler might use your function definition as a guideline to do something different but equivalent;

Imperative languages specify exactly *how* something should be done. You can consider an imperative compiler to act very robotically—it does *exactly* what you tell it to and you can easily map your code to what goes on at a machine code level;

Although it's useful to paint languages with these broad strokes, the truth is today's high-level languages should be viewed more as a collection of features. ML is a good example: it is certainly viewed as a functional language but it also supports all sorts of procedural programming constructs as you saw at the end of last term. Similarly, the compilers for most imperative languages support *optimisations* where they analyse small chunks of code and implement something different at machine-level to increase performance—this is of course a trait of declarative programming[1].

[1] Note that we need a way to switch off optimisations because they don't always work due to the presence of side effects in functions. Tracking down an error in an optimisation is painful: the 'bug' isn't in the code..!

1.2 Run as you go → 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()
```

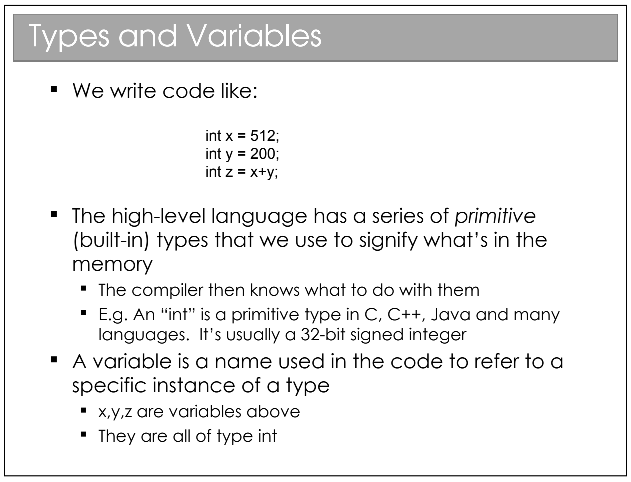
When ML reads in a source file, it interprets it as it goes. A call to execute a function causes the execution to occur at the moment it's read (how could you check that it doesn't compile the whole file first?).

If we are to compile our programs it is more normal to specify a start-of-execution function. This is little more than a function with a special name that the compiler watches out for. It is normal for this function to provide some way to get at the arguments supplied to the program when it is run. Most languages copy C/C++ in calling the function `main(...)`, and this includes Java which uses a signature as follows:[2]

```
public static void main(String[] args)
```

[2] See workbook 1

1.3 Type Inference → Explicit 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

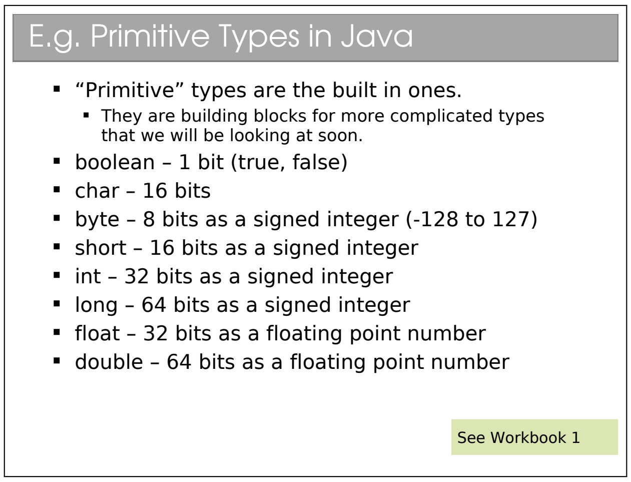
By this stage you've no doubt had a few headaches dealing with types in ML. When you wrote ML functions you tried hard to avoid specifying the types: occasionally you had to but you knew that if you could keep it general then you could use polymorphism to avoid writing separate functions for integers, reals, etc. This is a nice feature, although I acknowledge that ML's error messages could be a little less... cryptic.

There *are* imperative languages where you can still avoid specifying the type and rely on polymorphism (Python or Javascript for example) but they are more the exception than the norm. Java is characterised by:

- *every* value has a type assigned on declaration; and
- *every* function specifies the type of its output (its 'return type') *and* 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.[3]

[3]Later in the course we meet Generics, where the type is left more open. However, there is a type assigned to everything, even if it's just a placeholder.



E.g. Primitive Types in Java

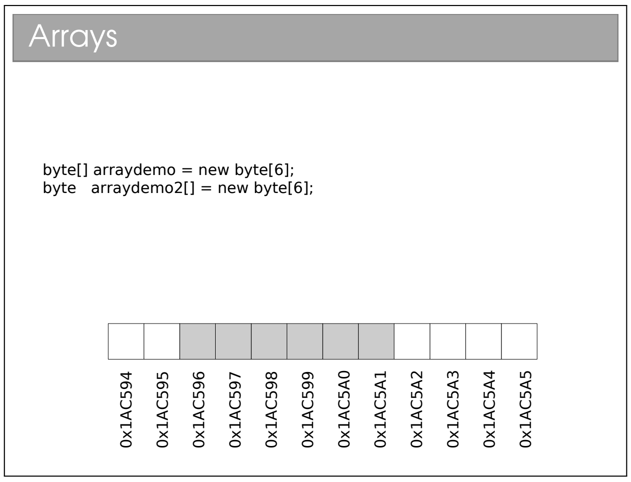
- "Primitive" types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 bits as a signed integer
- int – 32 bits as a signed integer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

These are the primitive types in Java[4]. For any C/C++ programmers out there: yes, Java looks a lot like the C syntax. But watch out for the obvious gotcha — a `char` in C is a byte (an ASCII character), whilst in Java it is two bytes (a Unicode character). If you have an 8-bit number in Java you may want to use a `byte`, but you also need to be aware that a `byte` is *signed*..!

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

1.4 Lists → Arrays



Arrays

```
byte[] arraydemo = new byte[6];
byte   arraydemo2[] = new byte[6];
```

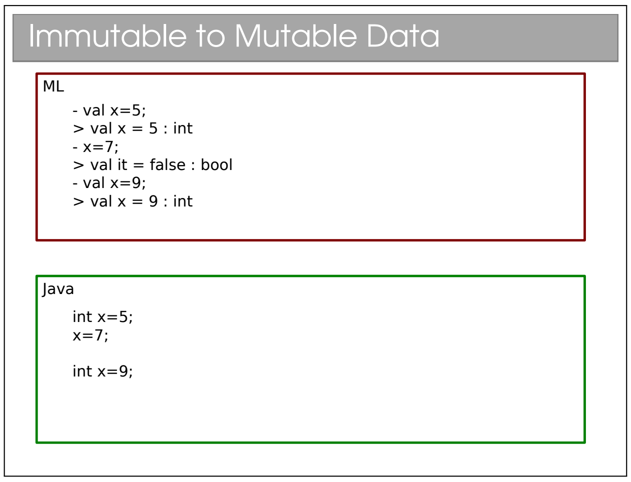
ML features tuples and lists as first class citizens of the language[5]. Most imperative languages feature arrays as a fundamental type. An array is a set of values stored sequentially in a single chunk of memory and maps directly to ML's Array, with the same properties:

[4]See workbook 1
[5]See workbook 3

- $O(1)$ element access;
- efficient storage—the next element is implicitly found in the next memory slot so no space wasted with pointers/references;
- inflexible sizing. 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.

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[]`)[6].

1.5 Immutable Data → 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;
```

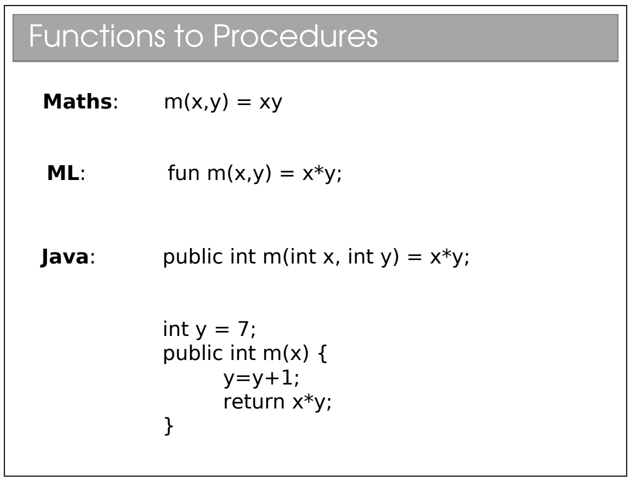
With ML you saw that data were immutable (i.e. unchangeable): an expression such as `val x=6` wrote the value 6 to some place in memory and attached the label `x` to it. You couldn't modify that piece of memory to change the 6 to, say, 5. You *could* reassign the label by writing `val x=5`, but this isn't the same thing (the 6 would still be in memory somewhere, at least for a limited time)

Immutability of data is useful in functional languages because it allows all sorts of clever optimisations to take place, not least to postpone 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, which is stored in some chunk of memory that we know will not change.

Imperative languages are all about manipulating state and you can't very well do so if nothing can be changed! So in Java we can happily assign and reassign values:

```
int x=6;  // Declare a label x to
          // an integer in memory set to 6
x=5;      // Change the value of the memory to 5
```

1.6 Mathematical Functions → Procedures



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;
}
```

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 (see the Computer Fundamentals notes)

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. example in slide).

Health warning: Most common languages today are imperative and many of them use the word 'function' as a synonym for 'procedure'. You will have to use your intelligence when you hear the words. 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 in-

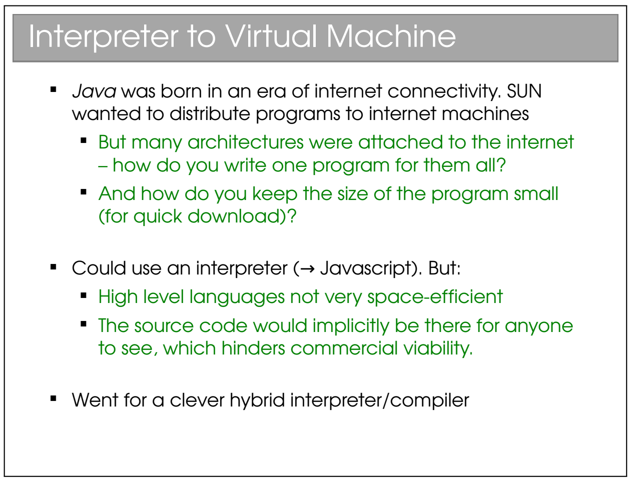
[6]See workbook 3

troduces a layer of complexity into programming that *might* make the results better but the job harder.

If you turn back to the discussion of functional and imperative, you can hopefully see that a function with side effect is *much* harder for a functional compiler to deal with since it is ambiguous *what* the function does (it doesn't have one nicely defined return value for a given argument). Hence functional languages do not allow side effects, sticking with proper functions.

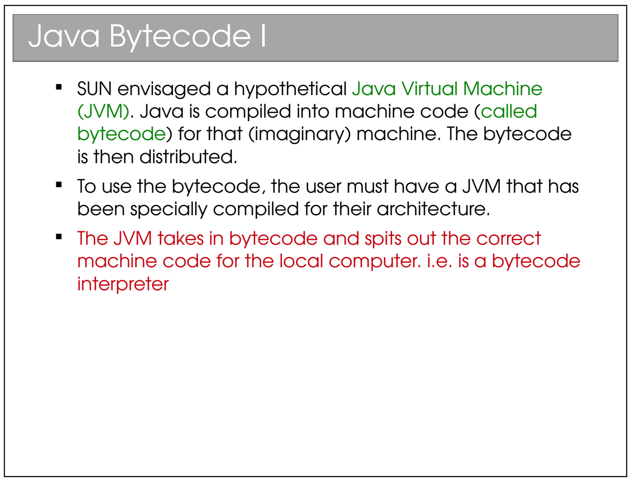
1.7 Interpreter → Virtual Machine

We've already discussed interpreters vs compilers but Java could be seen as a hybrid. Sun Microsystems invented Java as the web started to take off and suddenly many different devices with many different architectures were communicating. 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.



Interpreter to Virtual Machine

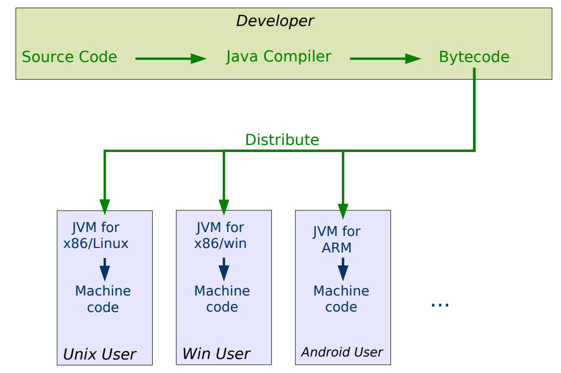
- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?
- Could use an 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



Java Bytecode I

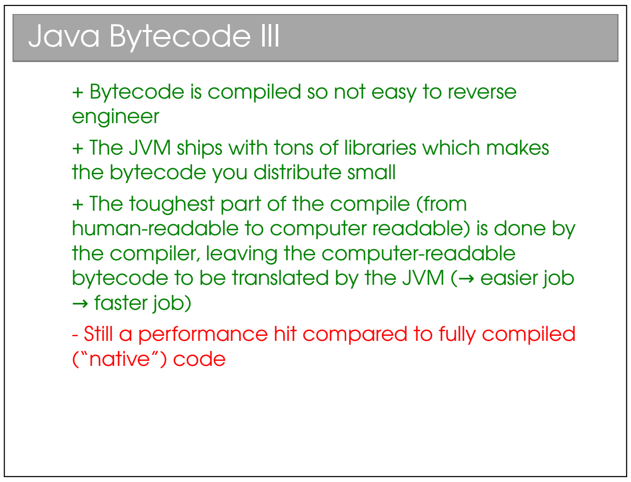
- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode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

Java Bytecode II



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 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 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/`

1.7.1 Viewing Bytecode

Once we have compiled our Java source code using javac, we end up with a set of .class files. These contain the bytecode and are what we distribute to allow people to run or program or use our classes.

There is also a javap program which allows you to poke inside a .class file. For example, you can disassemble a .class file to see an assembly-like view of the bytecode using javap -c classfile. E.g. with this input:

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

we get:

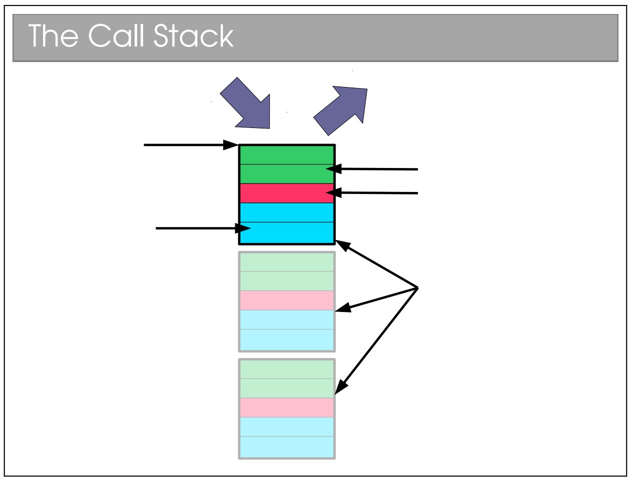
```
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object{
public HelloWorld();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

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.
                                  //(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 the intricacies of bytecode.

Lecture 2

Memory Manipulation

Imperative languages manipulate state held in system memory. They more naturally extend from assembly and it is useful for us to consider how most imperative compilers make use of memory.

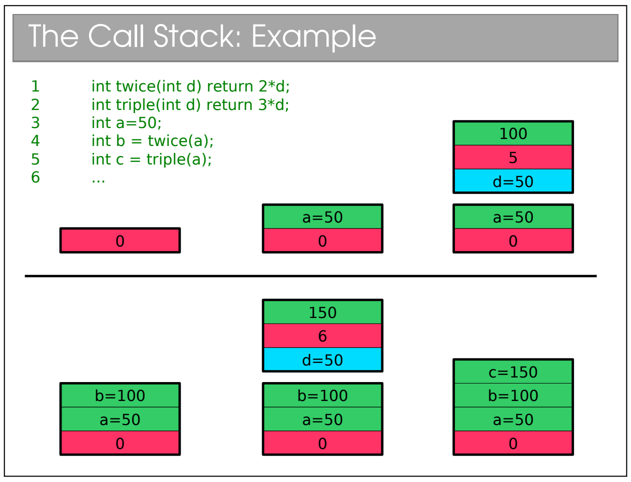


The Call Stack

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 before and continue on. This means that, before it jumps to the procedure code, it must save where it is.

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*[1]. 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 the return address it saved.

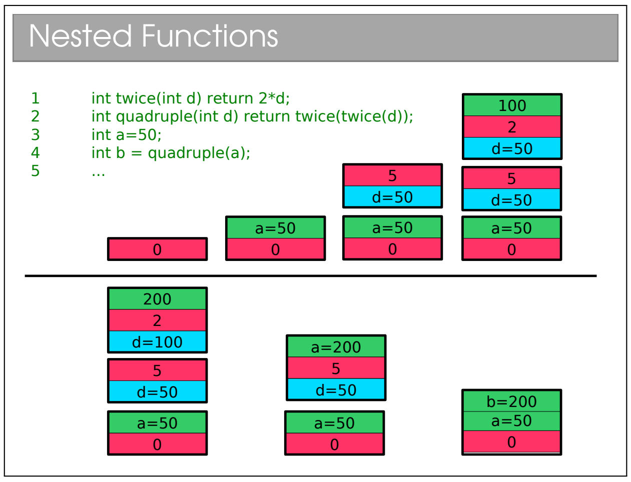
[1]See Algorithms I for a full analysis



The Call Stack: Example

```
1    int twice(int d) return 2*d;
2    int triple(int d) return 3*d;
3    int a=50;
4    int b = twice(a);
5    int c = triple(a);
6    ...
```

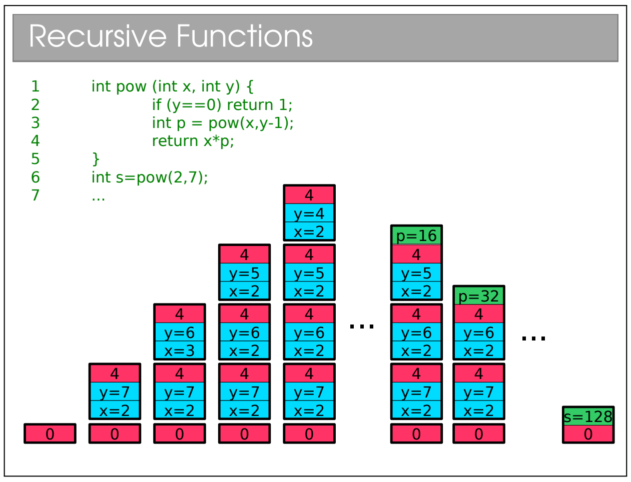
In this example I've avoided going down to assembly code and just assumed that the return address can be the code line number. This causes a small problem with e.g. line 4, which would be a couple of machine instructions (one to get the value of `twice{()` and one to store it in `b`). I've just assumed the computer magically remembers to store the return value for brevity. This is all very simple and the stack never gets very big—things are more interesting if we start nesting functions (i.e. calling functions from within another function):



Nested Functions

```
1    int twice(int d) return 2*d;
2    int quadruple(int d) return twice(twice(d));
3    int a=50;
4    int b = quadruple(a);
5    ...
```

And even more interesting if we start processing recursively:

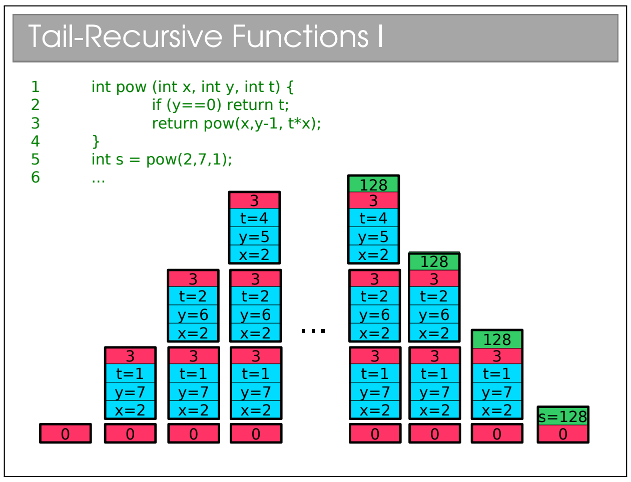


Recursive Functions

```
int pow (int x, int y) {
        if (y==0) return 1;
        int p = pow(x,y-1);
        return x*p;
}
int s=pow(2,7);
...
```

0
4 y=7 x=2 0
4 y=6 x=3 4 y=7 x=2 0
4 y=5 x=2 4 y=6 x=2 4 y=7 x=2 0
4 y=4 x=2 4 y=5 x=2 4 y=6 x=2 4 y=7 x=2 0
...
p=16 4 y=5 x=2 4 y=6 x=2 4 y=7 x=2 0
p=32 4 y=6 x=2 4 y=7 x=2 0
...
s=128 0

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 memory. We call this *stack overflow* and it is an unrecoverable error that you're almost certainly familiar with from ML. You know that tail-recursion does better, but:



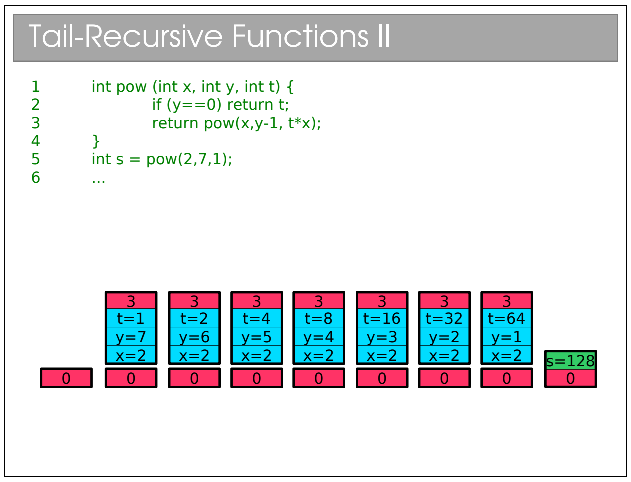
Tail-Recursive Functions I

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

0
3 t=1 y=7 x=2 0
3 t=2 y=6 x=2 3 t=1 y=7 x=2 0
3 t=4 y=5 x=2 3 t=2 y=6 x=2 3 t=1 y=7 x=2 0
...
128 3 t=4 y=5 x=2 3 t=2 y=6 x=2 3 t=1 y=7 x=2 0
128 3 t=2 y=6 x=2 3 t=1 y=7 x=2 0
128 3 t=1 y=7 x=2 0
s=128 0

If you're in the habit of saying tail-recursive functions are better, be careful—they're only better if the compiler/interpreter knows that it can optimise them to use $O(1)$ space. Java compilers don't...[2]

[2]Language designers usually speak of 1tail-call optimisation' since there is actually nothing special about recursion in this case: functions that call other functions may be written to use only tail calls, allowing the same optimisations.

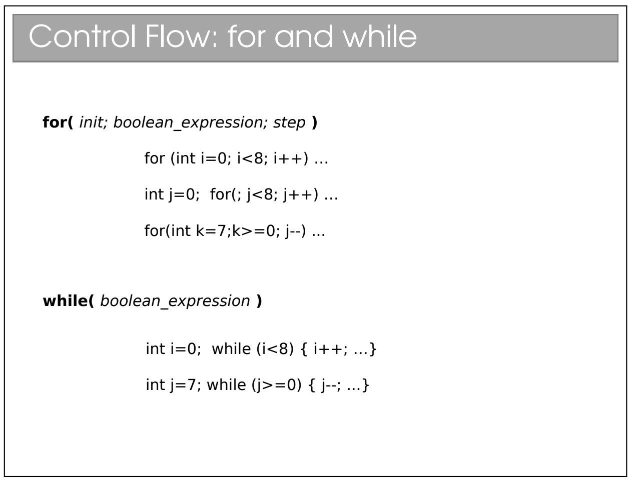


Tail-Recursive Functions II

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

2.1 Control Loops

However, optimised tail-recursion is equivalent to *iteration* and imperative languages support *explicit* iteration through the use of constructs such as while (as per ML) and for[3]. The following examples iterate exactly eight times.



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--; ...}
```

You may like to look up the other constructs and keywords for looping[4]. In particular, look at the 'do... while' and 'enhanced for' loops, and the 'break' and 'continue' keywords.

2.2 The Heap

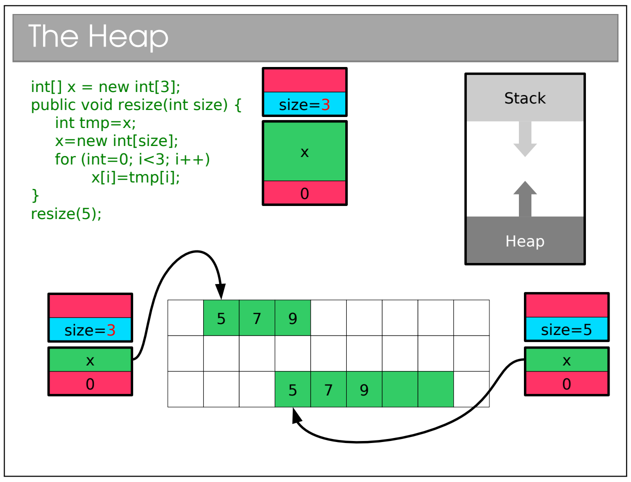
There's a subtlety with the stack that we've passed over until now. What if we want a function to create something that sticks around after it's gone? Or

[3]See Workbook 2
[4]See workbook 2

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.

We avoid this by using a *heap*[5]. 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 want to resize our array, we create a new, bigger array, copy the contents across and update the pointer within the stack.



The Heap

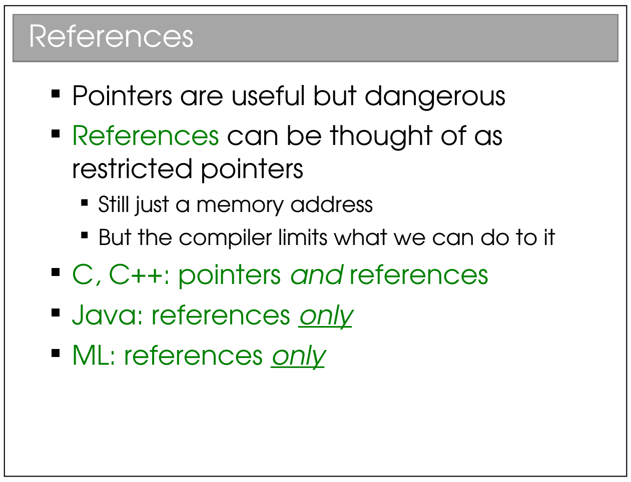
```
int[] x = new int[3];
public void resize(int size) {
    int tmp=x;
    x=new int[size];
    for (int=0; i<3; i++)
        x[i]=tmp[i];
}
resize(5);
```

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.

2.3 Pointers and References

Back in CF, we established pointers as variables holding memory addresses. In FoCS you encountered references, which were (sensibly) equated to pointers. Here, we will be a bit stricter and distinguish between pointers and references.

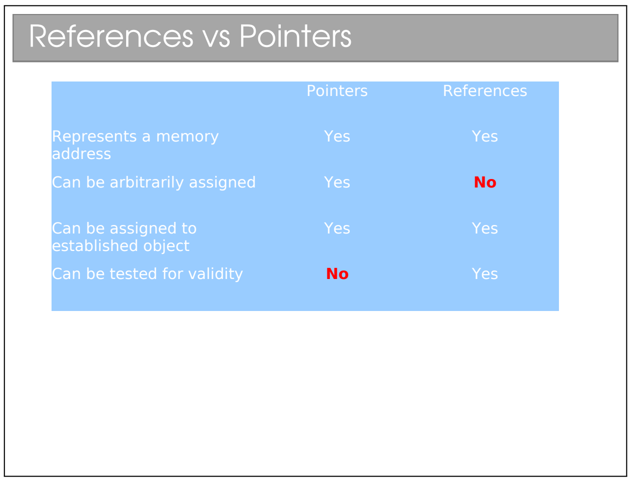
Pointers are simply variables whose value is a memory address. We can arbitrarily modify them either accidentally or intentionally and this can lead to all sorts of problems. Although the symptom is usually the same: program crash.



References

- Pointers are useful but dangerous
- References can be thought of as restricted pointers
  - Still just a memory address
  - But the compiler limits what we can do to it
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

References[6] can be seen as a fix for some of the more dangerous aspects of pointers. They are still just variables holding memory addresses, but the compiler (*not* the computer) will prevent us from doing certain operations on it to make things safer.



References vs Pointers

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

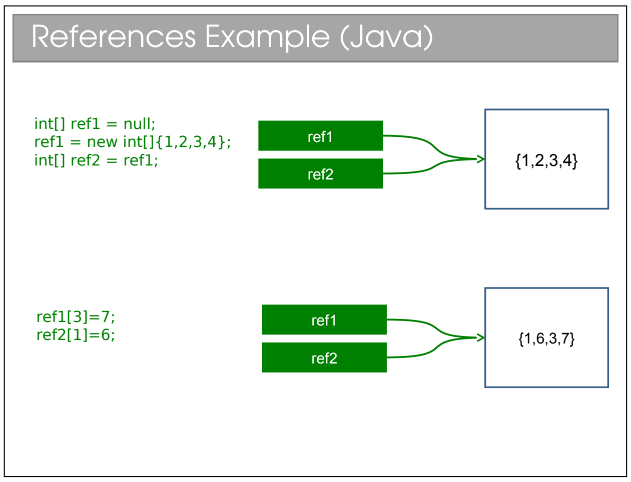
The last point is particularly important. A pointer points to something valid, something invalid, or `null` (a special zero-pointer that indicates it's not initialised). References, however, either point to something valid or to `null`. With a non-null reference, you know it's valid. With a non-null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

For those with experience with pointers, you might have found pointer arithmetic rather useful at times (e.g. incrementing a pointer to move one place forward in an array, etc). You can't do that with a reference since it would be a technique to create an invalid, non-null reference.

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

[6]See workbook 3

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[7]. Everything else is a reference type, including arrays and (as we will see) objects[8].


References Example (Java)

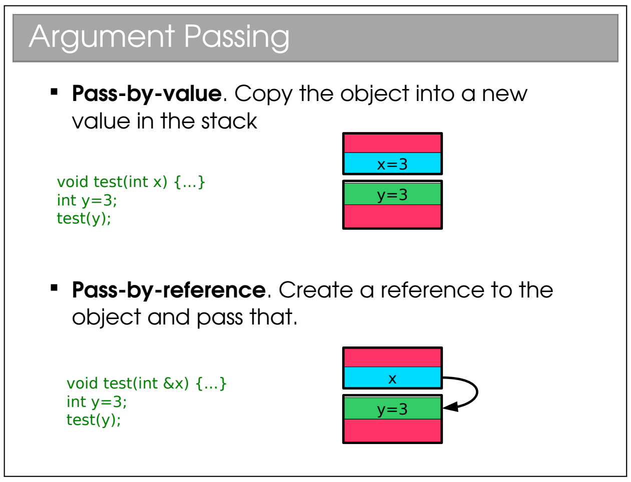
```
int[] ref1 = null;
ref1 = new int[]{1,2,3,4};
int[] ref2 = ref1;
```

```
ref1[3]=7;
ref2[1]=6;
```

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.

2.4 Pass-by-value and Pass-by-reference


Argument Passing

- **Pass-by-value**. Copy the object into a new value in the stack

```
void test(int x) {...}
int y=3;
test(y);
```

- **Pass-by-reference**. Create a reference to the object and pass that.

```
void test(int &x) {...}
int y=3;
test(y);
```

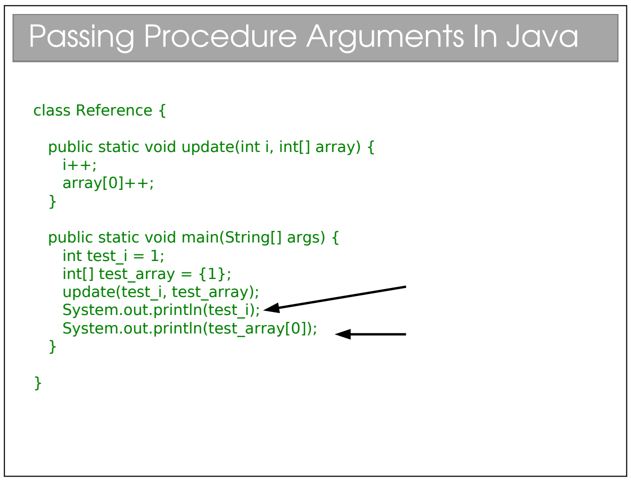
[7]See Workbook 1
[8]See Workbook 3

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

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 or otherwise), we pass a reference to it. Thus the function can access the original and (potentially) change it.

When arguments are passed to java functions, you may hear it said that primitive values are "passed by value" and arrays are "passed by reference". I think this is misleading (and technically wrong).


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[9], where you observed the different behaviour of test_i and test_array—the former being a primitive int and the latter being a reference to an array.

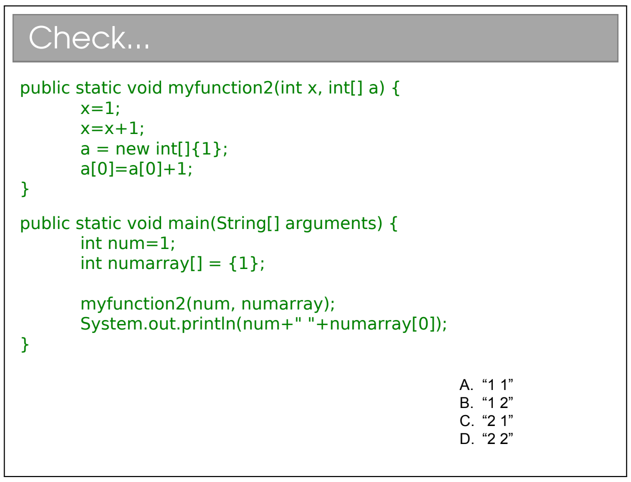
Let's create a model for what happens when we pass a primitive in Java, say an int like test_i. A new stack frame is created and the value of test_i is *copied* into the stack frame. You can do whatever you like to this copy: at the end of the function it is deleted along with the stack frame. The original 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 (which is just a memory address) is copied into a *new* reference in the stack frame. So, we have two references pointing at the same thing. Making modifications through either changes the original array.

[9]See workbook 3

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[10].

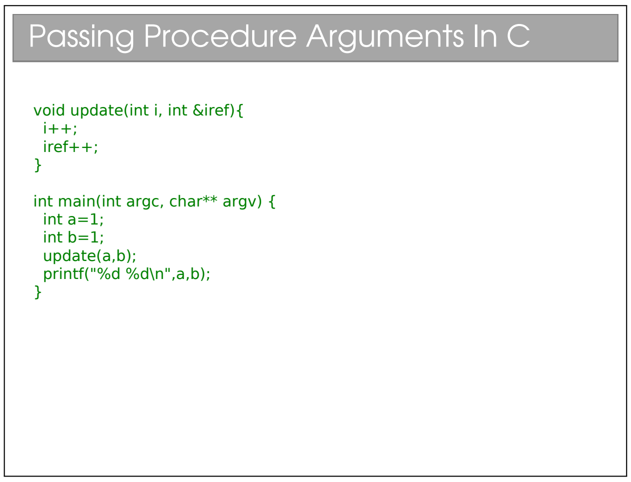
The confusion over this comes from the fact that many people view test_array to *be* the array and not a reference to it. If you think like that, then Java passes it by reference, as many books (incorrectly) claim. The examples sheet has a question that explores this further.



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;
}
```

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

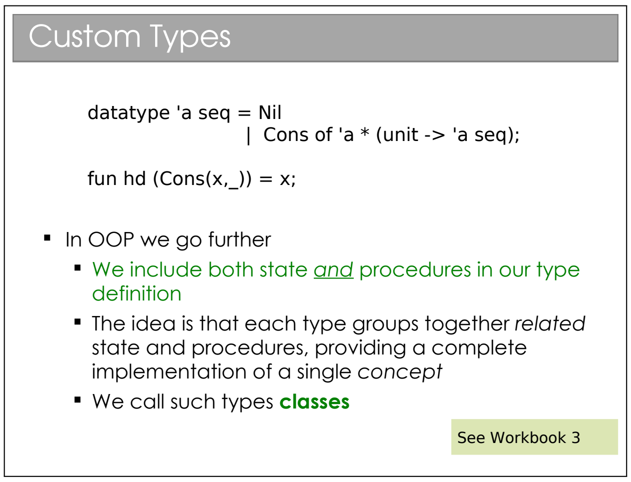
Here I have made a fictional type HugeInt which is meant to represent something that takes a lot of space in memory. Calling either of these functions will give the same answer, but what happens at a low level is quite different. In the first, the variable is copied (lots of memory copying required—bad) 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!

Lecture 3

OOP and Classes



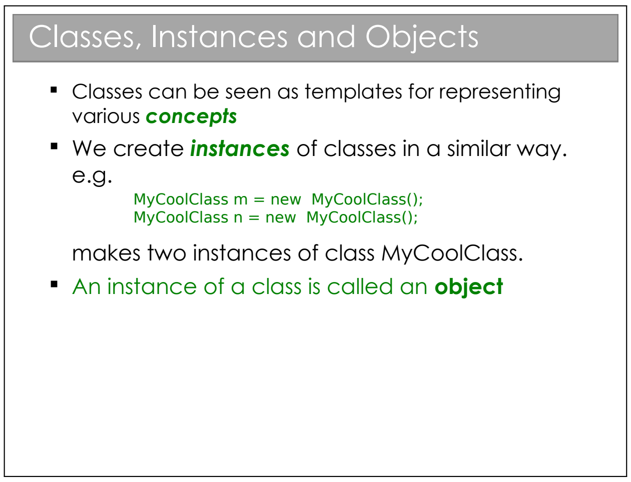
Custom Types

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

fun hd (Cons(x,_)) = x;
```

- In OOP we go further
  - We include both state *and* procedures in our type definition
  - The idea is that each type groups together *related* state and procedures, providing a complete implementation of a single *concept*
  - We call such types **classes**

See Workbook 3

In ML, each time you created a new type (such as sequences), you also had to construct a series of helper functions to manipulate it (e.g. hd(), tail(), merge(), etc.). There was an implicit link between the data type and the helper functions, since one was useless without the other. In OOP, we make this link explicit by defining classes that contain both state and behaviour—i.e. the functions become part of the data type declaration.



Classes, Instances and Objects

- Classes can be seen as templates for representing various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

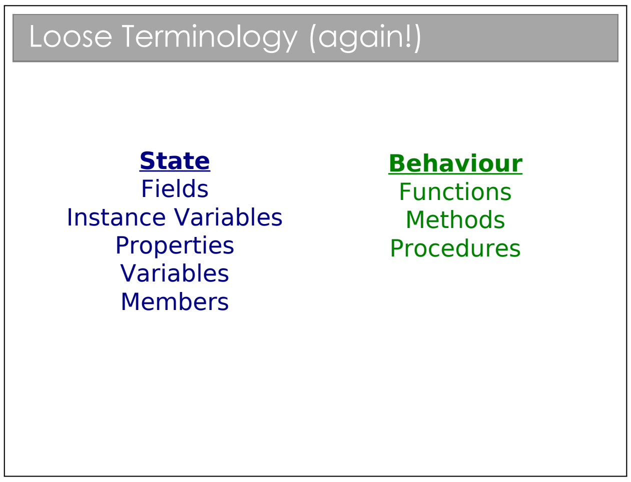
```
MyCoolClass m = new  MyCoolClass();
MyCoolClass n = new  MyCoolClass();
```

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

Each variable we declare is an *instance* of the type we assign. So a declaration such as int a declares an instance of the primitive int type and assigns it the name a.

Whenever we create an instance of a class, we call it an *object*. The difference between a class an an object is thus 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 particular values. So a Person class might specify that a Person has a name and an age. Our program may instantiate two Person objects—one might represent 40-year old Bob; another might represent 20 year-old Alice.

Note that we have just added a keyword to our repertoire: new is used to instantiate objects. We follow it with what looks like a function—this is actually the constructor for the type as we will see shortly.

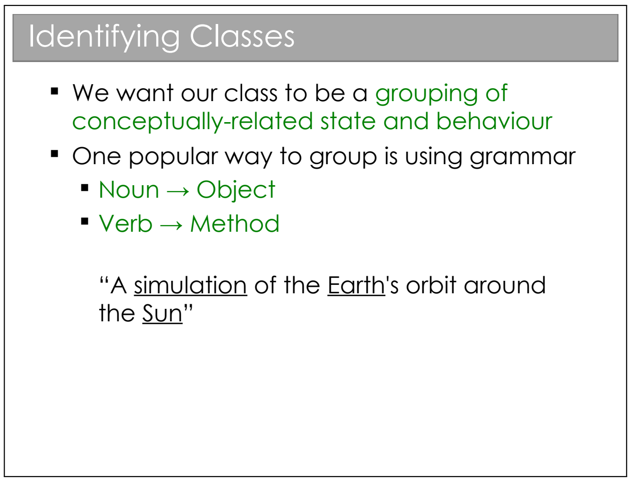


Loose Terminology (again!)

State	**Behaviour**
Fields	Functions
Instance Variables	Methods
Properties	Procedures
Variables	
Members	

Having made all that fuss about 'function' and 'procedure', it only gets worse here: when we're talking about a procedure inside a class, it's often called a *method*.

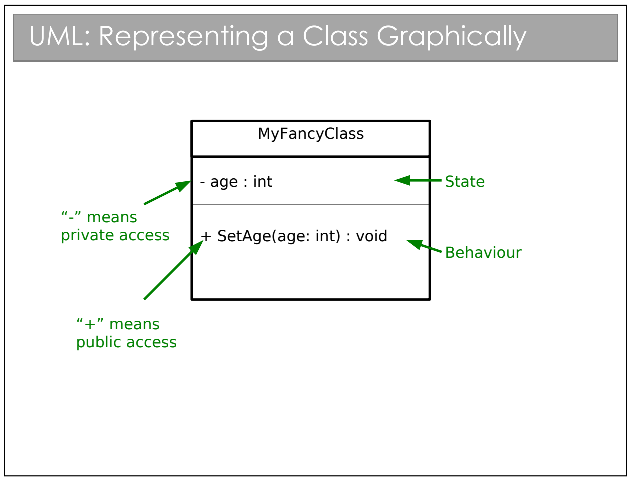
In the wild, you'll find people use 'function', 'procedure' and 'method' interchangeably. Thankfully you're all smart enough to cope!



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

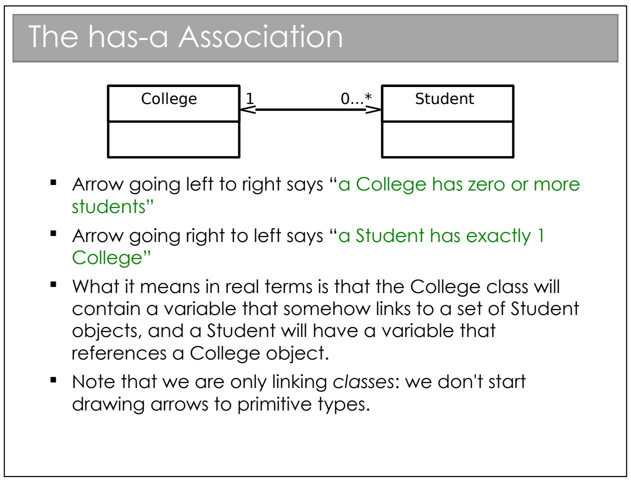
"A simulation of the Earth's orbit around the Sun"



UML: Representing a Class Graphically

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.

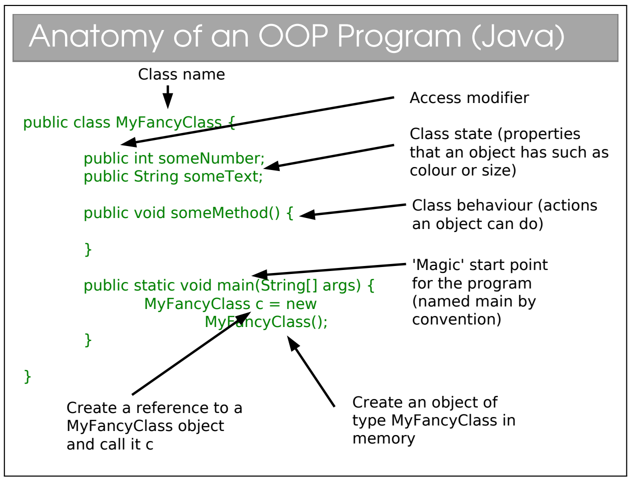
UML contains many different diagrams (touched on in the Software Design course for those doing Paper 2). In this course we will only use the *UML class diagram* such as the one in the slide.



The has-a Association

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

Note that the arrowhead must be 'open'. It is normal to annotate the head with the multiplicity, but some programmers are lax on this (for examination purposes, you *are* expected to annotate the heads). I've shown a dual-headed arrow; if the multiplicity value is zero, you can leave off the arrowhead and annotation entirely.

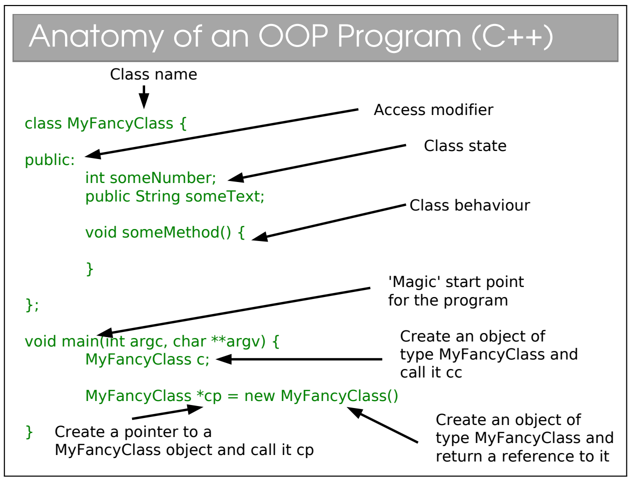


Anatomy of an OOP Program (Java)

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

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

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

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.



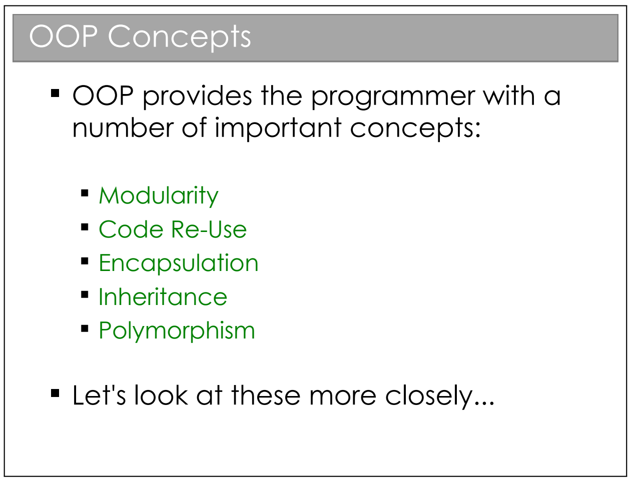
Anatomy of an OOP Program (C++)

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

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

        MyFancyClass *cp = new MyFancyClass()

}
```

Class name
Access modifier
Class state
Class behaviour
'Magic' start point for the program
Create an object of type MyFancyClass and call it cc
Create a pointer to a MyFancyClass object and call it cp
Create an object of type MyFancyClass and return a reference to it

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).

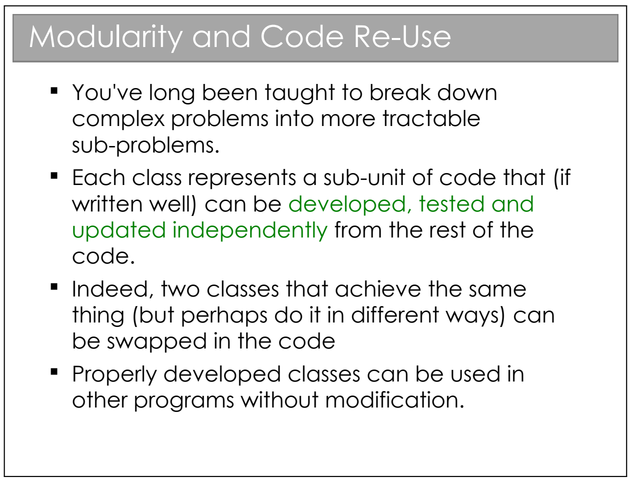
3.1 OOP and Classes


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 good for software design.

3.1.1 Modularity and Code Re-Use

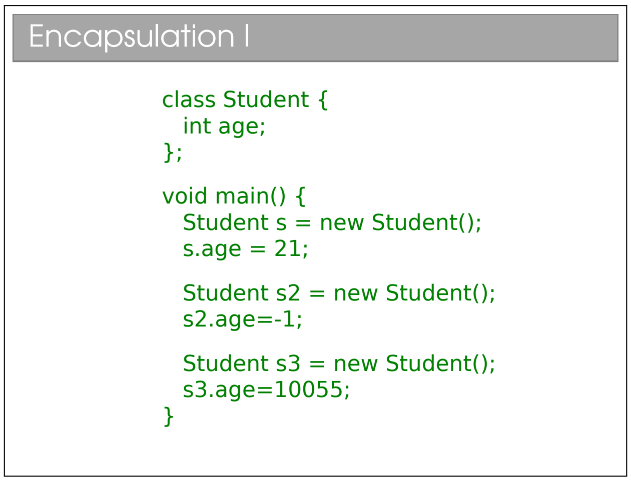

Modularity and Code Re-Use

- 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).

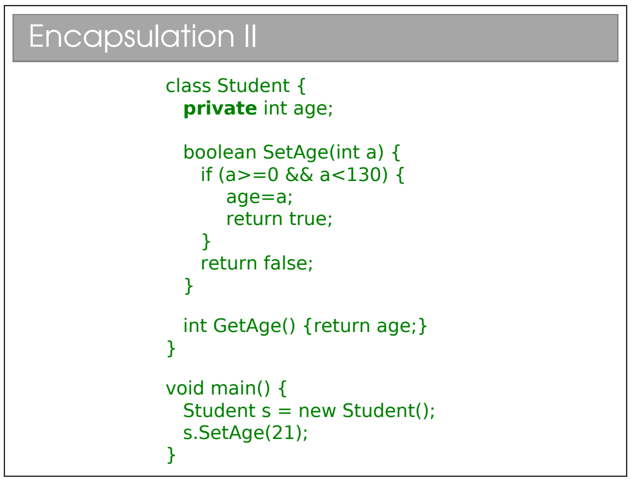
There is a further advantage to breaking a program down into self-contained objects: those objects can be ripped from the code and put into other programs. So, once you've developed and tested a class that represents a Student, say, you can use it in lots of other programs with minimal effort. Even better,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

3.1.2 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;
}
```

This code defines a basic Student class, with only one piece of state per Student. In the main() method we create three instances of Students. We observe that nothing stops us from assigning nonsensical values to 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);
}
```

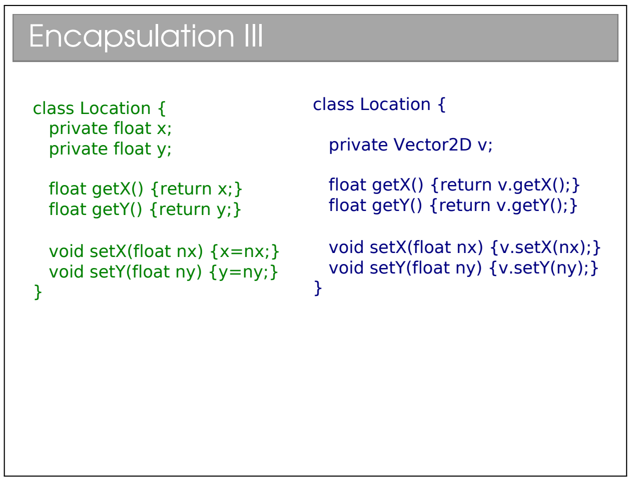
Here we have assigned an *access modifier* called private to the age variable. This means nothing external to the class (i.e. no piece of code defined outside of the class definition) can read or write the age variable[1].

Another name for encapsulation is *information hiding* or even *implementation hiding* in some texts. The basic idea is that a class should expose a clean interface

[1]See workbook 3

that allows full interaction with it, but nothing about its internal state. The general rule is that all state is private unless there is a very good reason for it not to be.

To get access to the age variable we define a getAge() and a setAge() method to allow read and write, respectively. On the face of it, this is just more code to achieve the same thing. However, we have new options: by omitting setAge() altogether we can prevent anyone modifying the age (thereby adding immutability!); or we can provide sanity checks in the setAge() code to ensure we store sensible values.



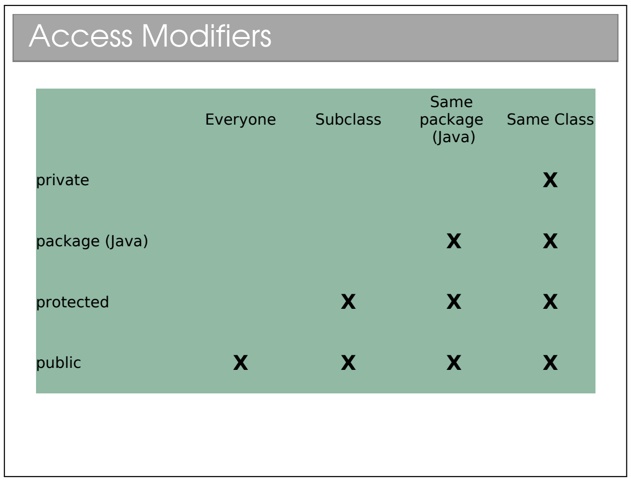
Encapsulation III

```
class Location {
  private float x;
  private float y;

  float getX() {return x;}
  float getY() {return y;}

  void setX(float nx) {x=nx;}
  void setY(float ny) {y=ny;}
}
```

```
class Location {

  private Vector2D v;

  float getX() {return v.getX();}
  float getY() {return v.getY();}

  void setX(float nx) {v.setX(nx);}
  void setY(float ny) {v.setY(ny);}
}
```

Here we have a simple example where we wish to change the underlying representation of a co-ordinate (x,y) from raw primitives to a custom Vector2D object. We can do this without changing the public interface to the class and hence without having to update any piece of code that uses the Location class.

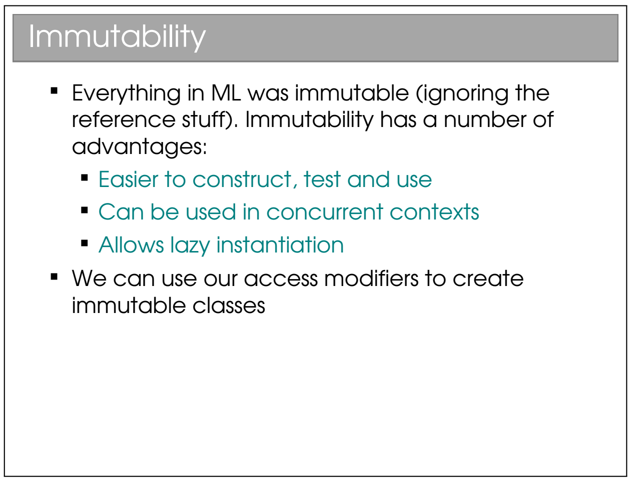
You may hear people talking about *coupling* and *cohesion*. Coupling refers to how much one class depends on another. High coupling is bad since it means changing one class will require you to fix up lots of others. Cohesion is a qualitative measure of how strongly related everything in the class is—we strive for high cohesion. Encapsulation helps to minimise coupling and maximise cohesion.



Access Modifiers

	Everyone	Subclass	Same package (Java)	Same Class
private				X
package (Java)			X	X
protected		X	X	X
public	X	X	X	X

OOP languages feature some set of access modifiers that allow us to do various levels of data hiding. C++ has the set {public, protected, private}, to which Java has added package.[2] Don't worry if you don't yet know what a "Subclass" is—that's in the next lecture.



Immutability

- Everything in ML was immutable (ignoring the reference stuff). Immutability has a number of advantages:
  - Easier to construct, test and use
  - Can be used in concurrent contexts
  - Allows lazy instantiation
- We can use our access modifiers to create immutable classes

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.

To quote *Effective Java* by Joshua Bloch:

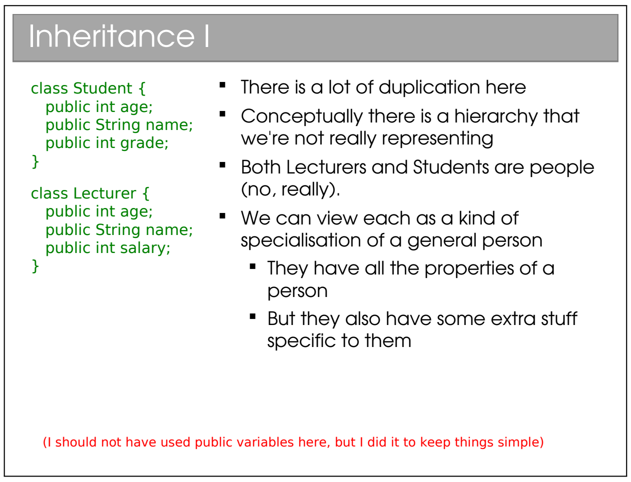
> "Classes should be immutable unless there's a very good reason to make them mutable...

[2]You've met Java packages in your practicals as a way to group classes together. It's useful there because you all write classes with the same names, and having unique packages (based on your CRSID) allows us to distinguish them when testing.

If a class cannot be made immutable, limit
its mutability as much as possible.”

Lecture 4

Inheritance and Polymorphism



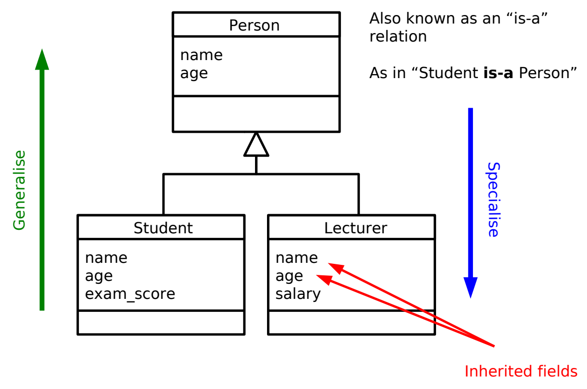
Inheritance I

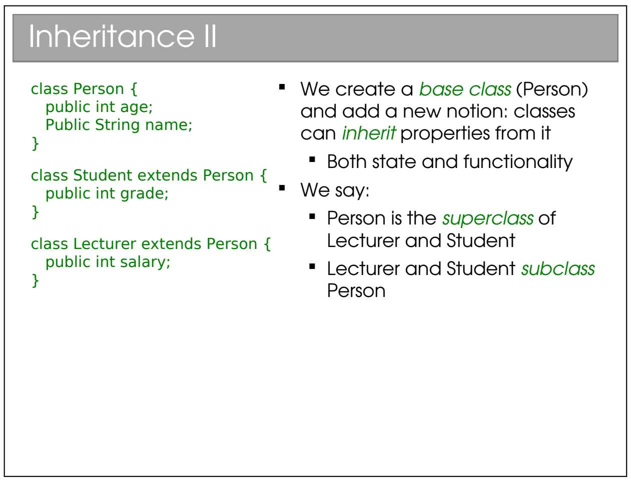
```
class Student {
   public int age;
   public String name;
   public int grade;
}
class Lecturer {
   public int age;
   public String name;
   public int salary;
}
```

- 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)

Representing Inheritance Graphically





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

Java uses the keyword `extends` to indicate inheritance of classes. In C++ it's a more opaque colon:

```
class Parent {...};
class Student : public Parent {...};
class Lecturer : public Parent {...};
```

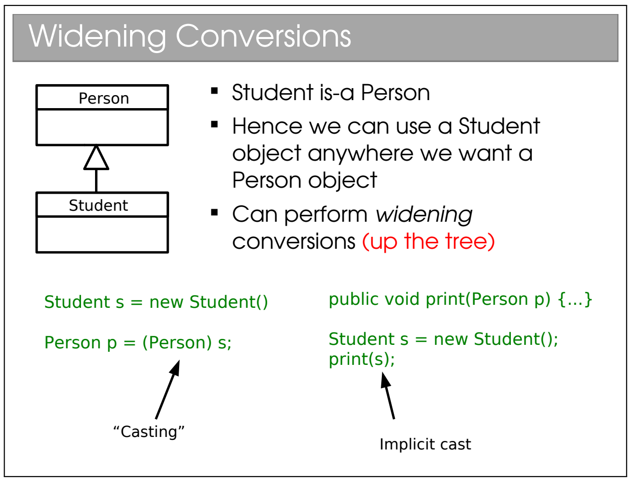
Inheritance[1] is an extremely powerful concept that is used extensively in good OOP. We discussed the "has-a" relation amongst classes; inheritance adds an "is-a" concept. E.g. A car *is a* vehicle that *has a* steering wheel.

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 the child of A
- B derives from A
- B extends A
- B inherits from A
- B subclasses A

Many students confuse "is-a" and "has-a" arrows in their UML class diagrams: please make sure you don't! Inheritance has an empty triangle for the arrowhead, whilst association has two 'wings'.

[1]See workbook 5



Widening Conversions

Person
Student

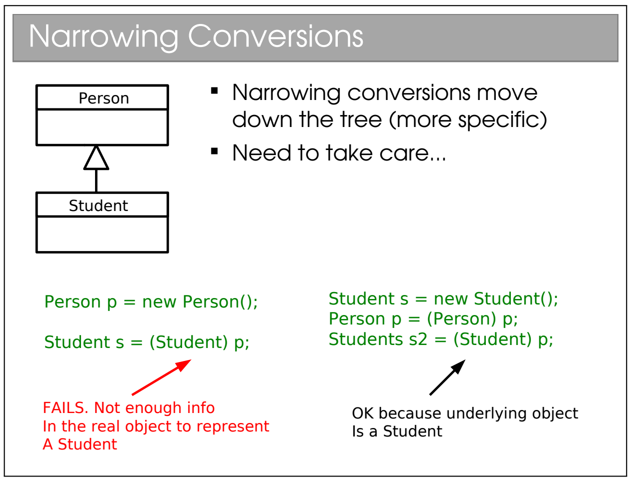
- Student is-a Person
- Hence we can use a Student object anywhere we want a Person object
- Can perform *widening* conversions (up the tree)

```
Student s = new Student()

Person p = (Person) s;
```

"Casting"

```
public void print(Person p) {...}

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

Implicit cast



Narrowing Conversions

Person
Student

- Narrowing conversions move down the tree (more specific)
- Need to take care...

```
Person p = new Person();

Student s = (Student) p;
```

FAILS. Not enough info
In the real object to represent
A Student

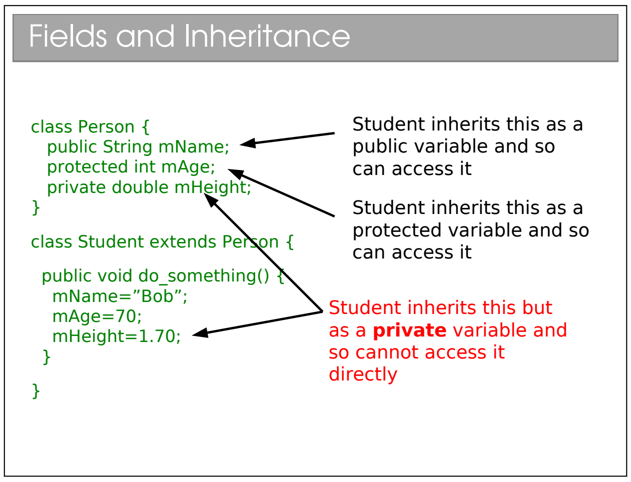
```
Student s = new Student();
Person p = (Person) p;
Students s2 = (Student) p;
```

OK because underlying object
Is a Student

When we create an object, a specific chunk of memory is allocated with all the necessary info and a reference to it returned (in Java). Casting just creates a new reference with a different type and points it to the same memory chunk. Everything we need will be in the chunk if we cast to a parent class (plus some extra stuff).

If we try to cast to a child class, there won't be all the necessary info in the memory so it will fail. *But* beware—you don't get a compiler error in the failed example above! The compiler is fine with the cast and instead the program chokes when we try to *run* that piece of code—a *runtime* error.

4.0.3 Inheritance and State



Fields and Inheritance

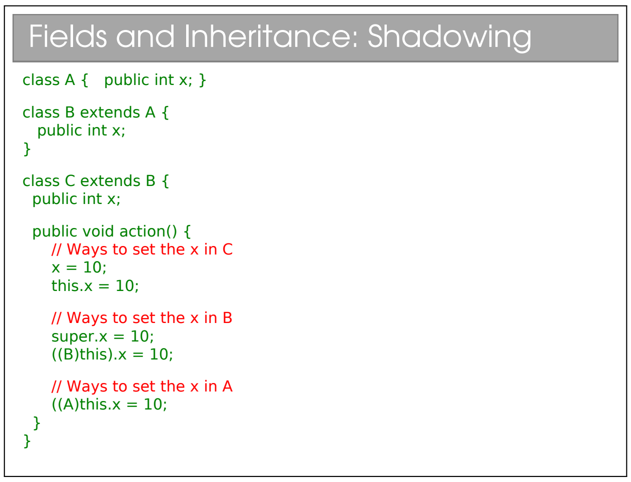
```
class Person {
  public String mName;
  protected int mAge;
  private double mHeight;
}
class Student extends Person {
  public void do_something() {
    mName="Bob";
    mAge=70;
    mHeight=1.70;
  }
}
```

Student inherits this as a public variable and so can access it

Student inherits this as a protected variable and so can access it

Student inherits this but as a **private** variable and so cannot access it directly

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[2].



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 in 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.

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

There are two new keywords that have appeared here: super and this. The this keyword can be used in any class method[3] and provides us with a reference to the current object. In fact, the this keyword is what you need to access anything within a class, but because we'd end up writing this all over the place, it is taken as implicit. So, for example:

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

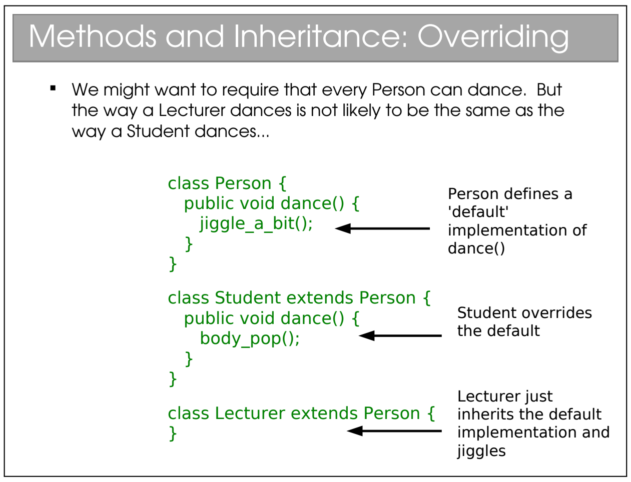
becomes:

```
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 keywords in your Java practicals.

4.0.4 Inheriting Methods and Polymorphism

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



Methods and Inheritance: Overriding

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

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

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

class Lecturer extends Person {
}
```

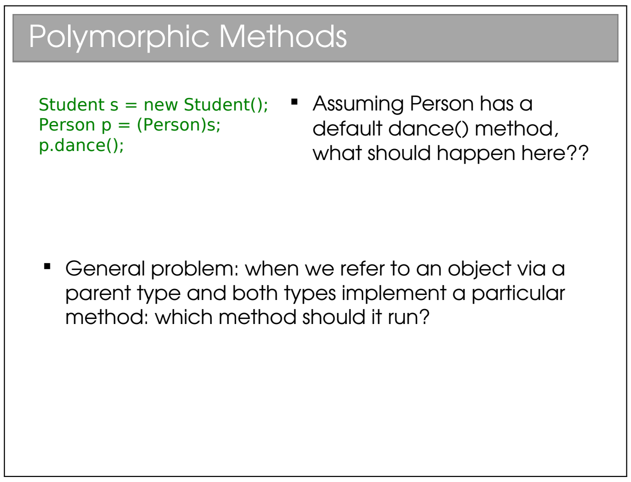
Person defines a 'default' implementation of dance()

Student overrides the default

Lecturer just inherits the default implementation and jiggles

[3]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.

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.

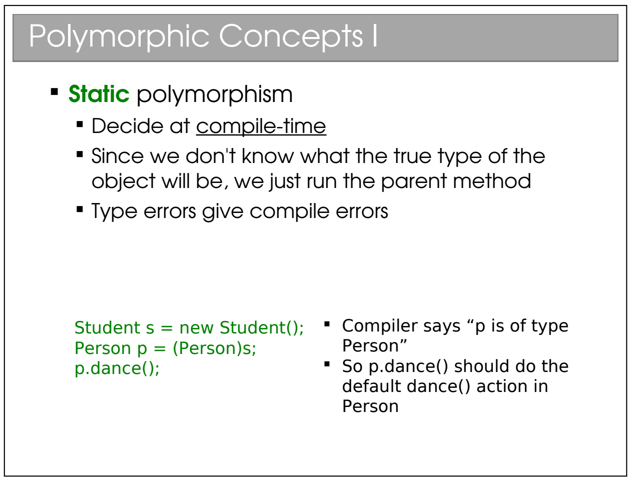


Polymorphic Methods

```
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

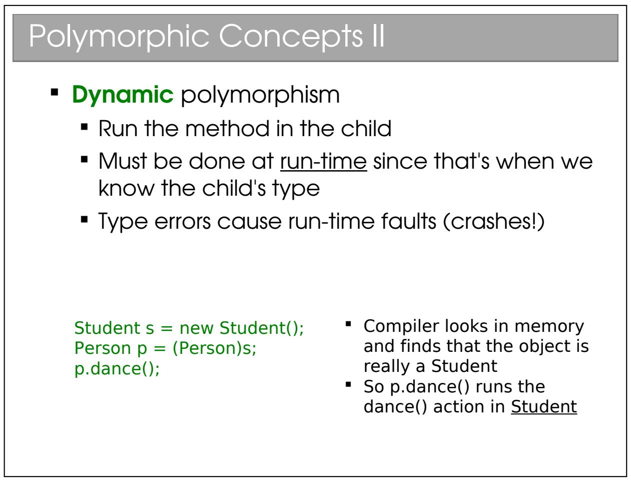
- **Static** polymorphism
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we just run the parent method
  - Type errors give compile errors

```
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[4] refers to anything where decisions are made at compile-time. You may realise that all the polymorphism you saw in ML was static polymorphism. The shadowing of fields also fits this description.

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



Polymorphic Concepts II

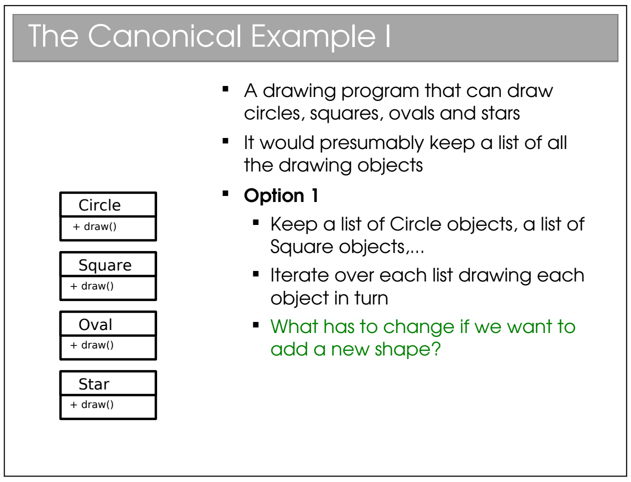
- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

```
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 *sub-type* 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.

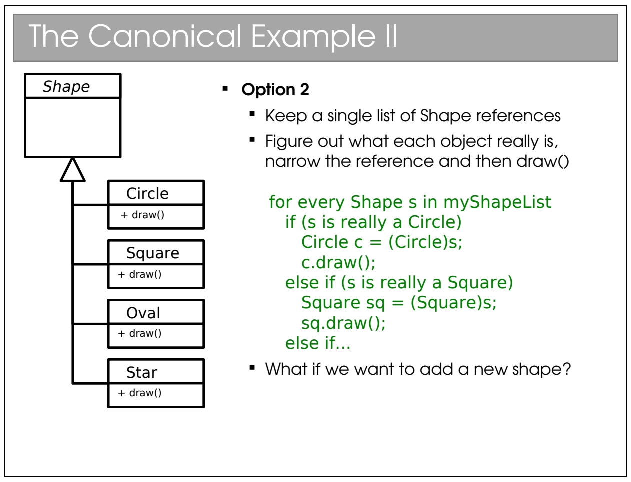
Beware: Most programmers use the word 'polymorphism' to refer to dynamic polymorphism.



The Canonical Example I

- 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?

Circle
+ draw()
Square
+ draw()
Oval
+ draw()
Star
+ draw()



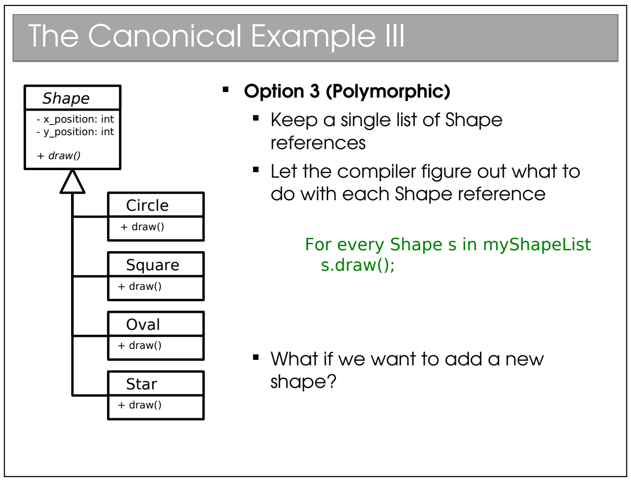
The Canonical Example II

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

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

  - What if we want to add a new shape?

Shape
Circle
+ draw()
Square
+ draw()
Oval
+ draw()
Star
+ draw()



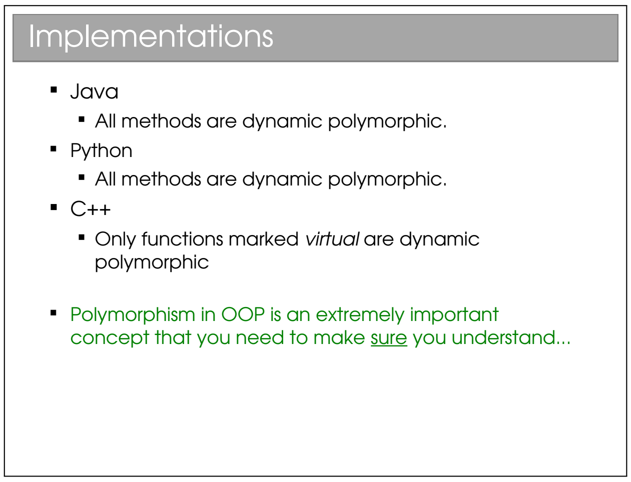
The Canonical Example III

- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

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

  - What if we want to add a new shape?

Shape
- x_position: int
- y_position: int
+ draw()
Circle
+ draw()
Square
+ draw()
Oval
+ draw()
Star
+ draw()



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...

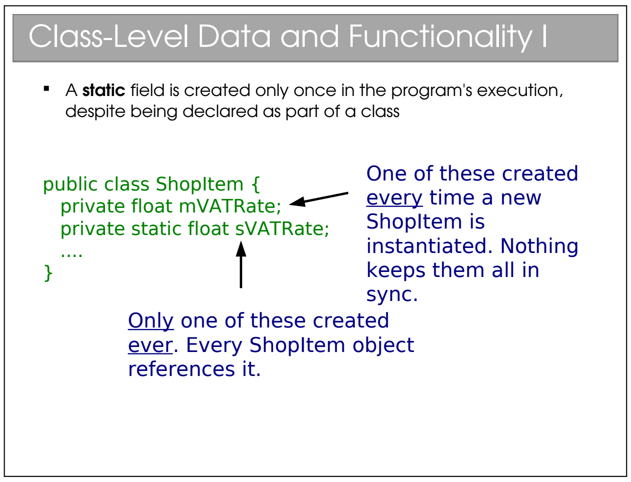
C++ allows you to choose whether methods are inherited statically (default) or dynamically (explicitly labelled with the keyword virtual). This can be good for performance (you only incur the dynamic overhead when you need to) but gets complicated, especially if the base method isn't dynamic but a derived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

The Java designers avoided the problem by enforcing dynamic polymorphism. You may find reference to final methods being Java's static polymorphism since this gives a compile error if you 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!

Lecture 5

Static Data, Abstract Classes and Interfaces

5.1 Class-Level Data



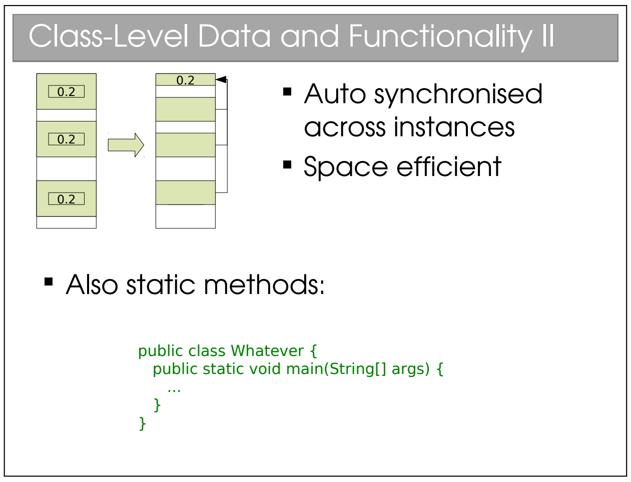
Class-Level Data and Functionality I

- A **static** field is created only once in the program's execution, despite being declared as part of a class

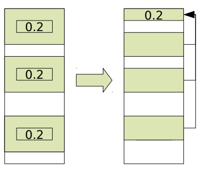
```
public class ShopItem {
   private float mVATRate;
   private static float sVATRate;
   ....
}
```

One of these created every time a new ShopItem is instantiated. Nothing keeps them all in sync.
Only one of these created ever. Every ShopItem object references it.

You don't even need to instantiate a class to access a static member. just writing ShopItem.sVATRate would give you access. You see examples of this in the Math class provided by Java: you can just call Math.PI to get the value of pi, rather than creating a Math object first.



Class-Level Data and Functionality II

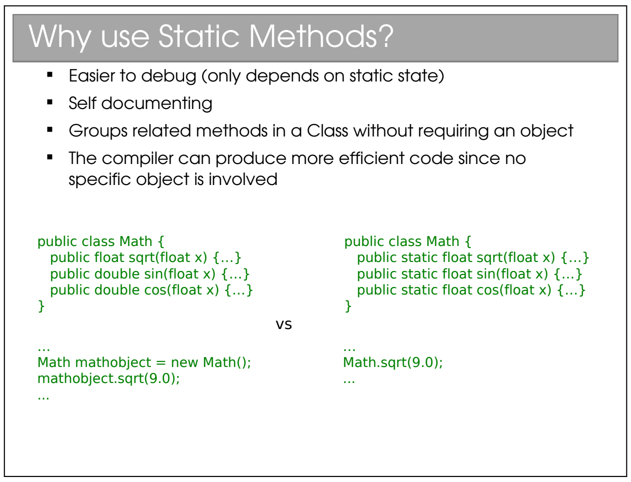


- Auto synchronised across instances
- Space efficient

- Also static methods:

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

In order for a method to be static, it must not make use of anything other than local or static variables. So it can't use anything that is instance-specific (i.e. non-static member variables are out).



Why use Static Methods?

- Easier to debug (only depends on static state)
- Self documenting
- Groups related methods in a Class without requiring an object
- 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);
...
```

In your first few practicals you were encouraged to write static methods to avoid having to instantiate objects all over the place.

5.2 Abstract Methods and Classes

Abstract Methods

- Sometimes we want to force a class to implement a method but there isn't a convenient default behaviour
- An **abstract** method is used in a base class to do this
- It has no implementation whatsoever

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

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

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

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

- Note that I had to declare the class abstract too. This is because it has a method without an implementation so we can't directly instantiate a Person.

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

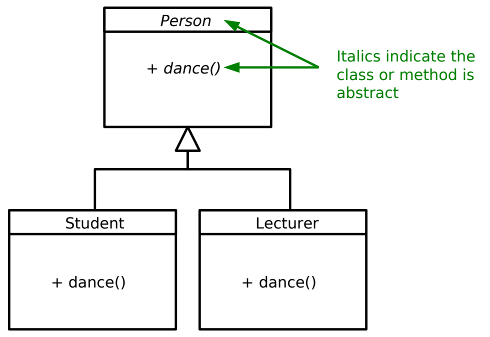
```
class Person {
  public:
    virtual void dance()=0;
}
                          C++
```

- All state and non-abstract methods are inherited as normal by children of our abstract class
- Interestingly, Java allows a class to be declared abstract even if it contains no abstract methods!

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

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

Representing Abstract Classes

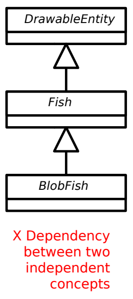


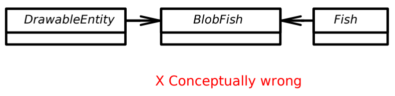
You have to look at UML diagrams carefully since the italics that represent abstract methods or classes aren't always obvious on a quick glance.

5.3 Multiple Inheritance and Interfaces

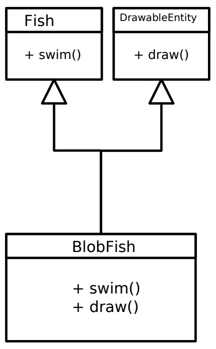
Harder Problems

- Given a class Fish and a class DrawableEntity, how do we make a BlobFish class that is a drawable fish?





Multiple Inheritance

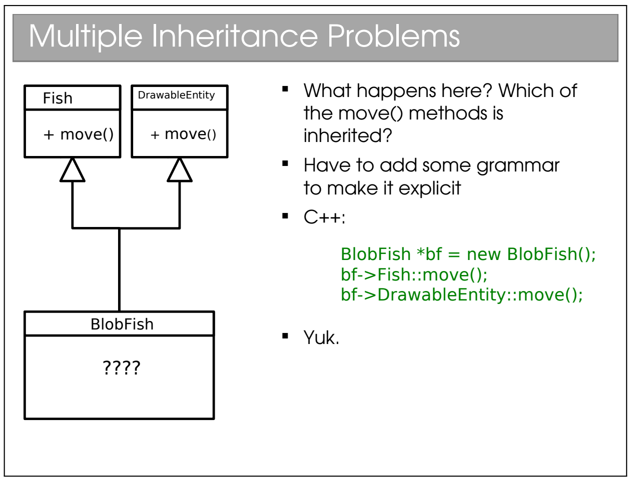


- If we multiple inherit, we capture the concept we want
- BlobFish inherits from both and is-a Fish and is-a DrawableEntity
- C++:

```
class Fish {...}
class DrawableEntity {...}

class BlobFish : public Fish,
                 public DrawableEntity {...}
```

- But...

This is the obvious and (perhaps) sensible option that manages to capture the concept nicely.



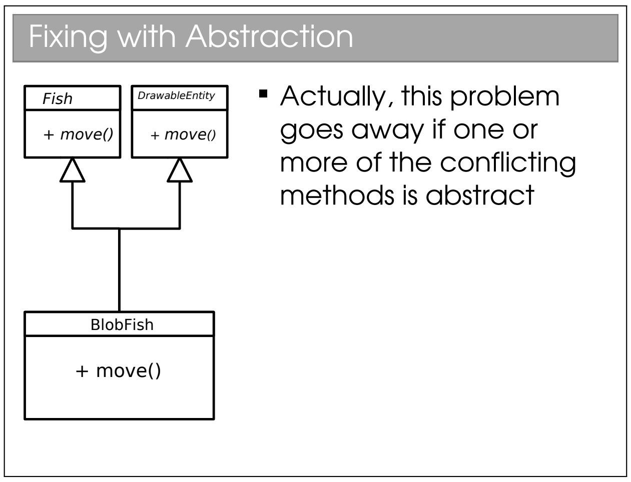
Multiple Inheritance Problems

- What happens here? Which of the move() methods is inherited?
- Have to add some grammar to make it explicit
- C++:

```
BlobFish *bf = new BlobFish();
bf->Fish::move();
bf->DrawableEntity::move();
```

- Yuk.

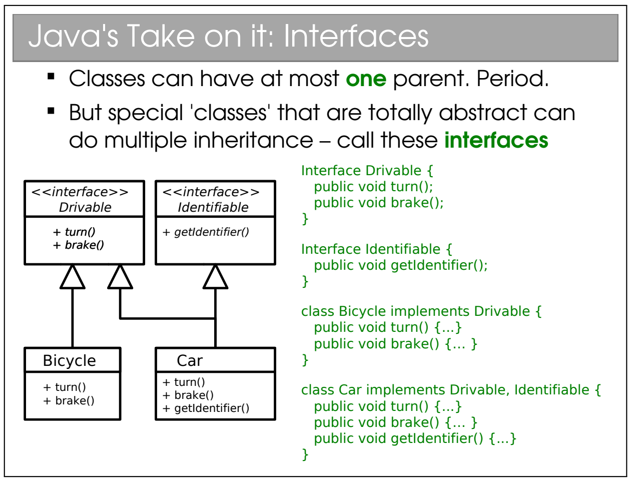
Many texts speak of the "dreaded diamond". This occurs when a base class has two children who are the parents of another class through multiple inheritance (thereby forming a diamond in the UML diagram). If the two classes in the middle independently override a method from the top class, the bottom class suffers from the problem in this slide.



Fixing with Abstraction

- Actually, this problem goes away if one or more of the conflicting methods is abstract

The problem goes away here because the methods are abstract and hence have no implementation that can conflict.



Java's Take on it: Interfaces

- Classes can have at most **one** parent. Period.
- But special 'classes' that are totally abstract can do multiple inheritance – call these **interfaces**

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

Interface Identifiable {
  public void getIdentifier();
}

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

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

So Java allows you to inherit from one class *only* (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 (often a better way, although not always!).

A Java *interface*[1] is essentially just a class that has:

- *No* state whatsoever; and
- *All* methods abstract.

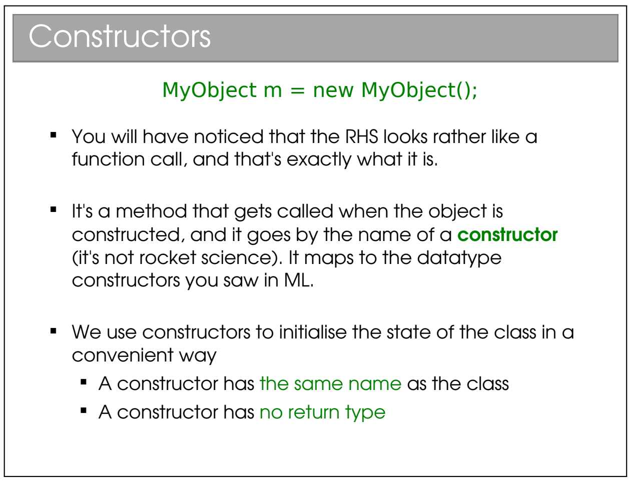
This is a greatly simplified concept that allows for multiple inheritance without any chance of conflict. Interfaces are represented in our UML class diagram with a preceding <<interface>> label and inheritance occurs via the implements keyword rather than through extends.

Interfaces are so important in 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, it is possible to overdo it, ending up with 20 files where one would do...

[1]See workbook 5

Lecture 6

Construction, Destruction and Error Handling

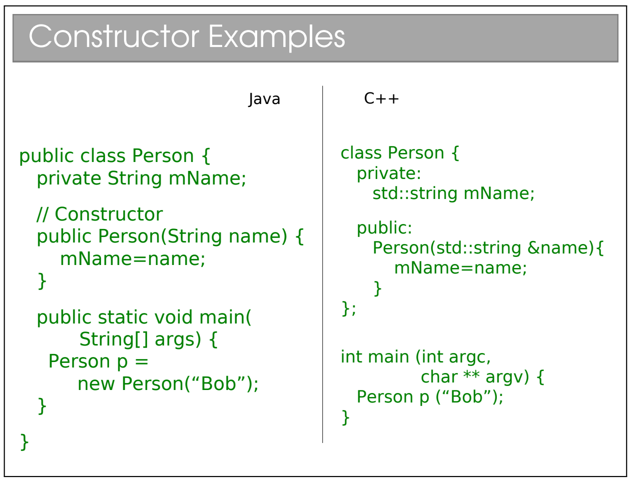


Constructors

```
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). It maps to the datatype constructors you saw in ML.
- 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 usual way for a method.



Constructor Examples

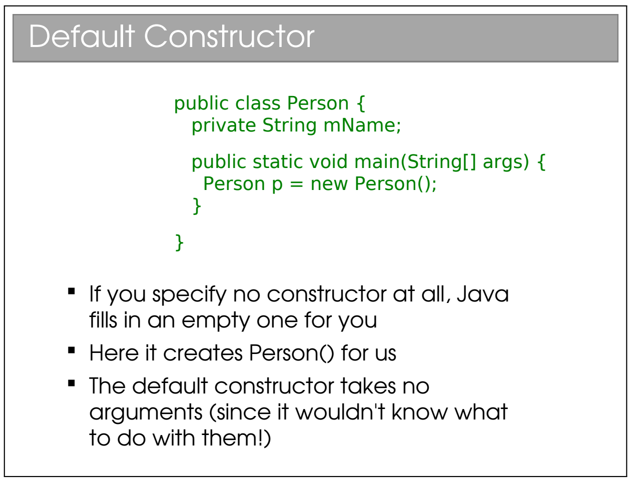
Java

```
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”);
}
```

As with many OOP features, not all languages support it. Python, for example, doesn't have constructors. It *does* have a single __init__ method in each class that acts a bit like a constructor but technically isn't (python fully constructs the object, and returns a reference that gets passed to __init__ if it exists—similar, but not quite the same thing.

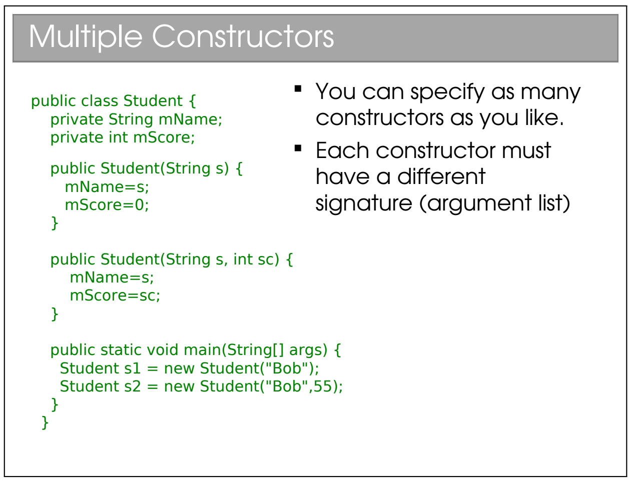


Default Constructor

```
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!)

In languages such as Java and C++ *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.



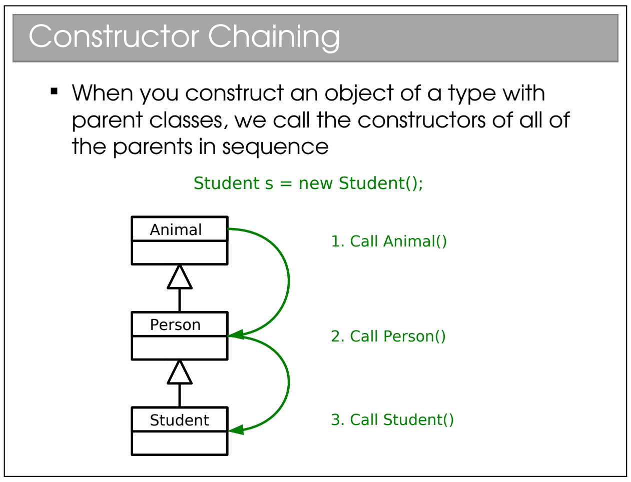
Multiple Constructors

```
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)

Again, not all languages support this. Python doesn't support multiple overloaded __init__ methods, and this can be a bit frustrating,

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?



Constructor Chaining

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

Student s = new Student();

In reality, Java asserts 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:

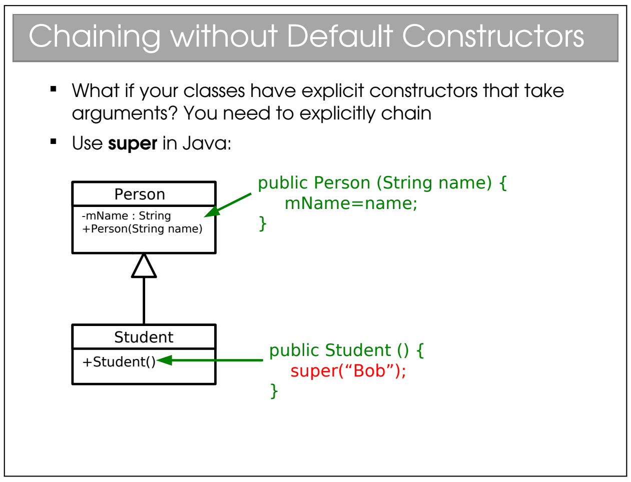
```
public class Person {
  public Person() {

  }
}
```

becomes:

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

In other languages that support multiple inheritance, this becomes more complex since there may be more than one parent and a simple keyword like super isn't enough. Instead they support manually specifying the constructor parameters for the parents. E.g. for C++:

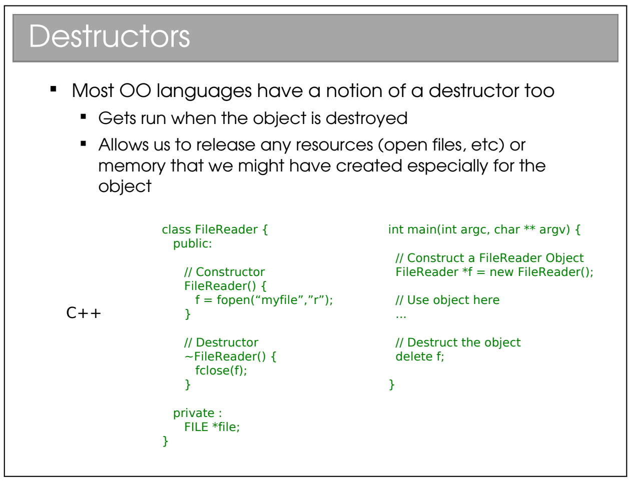
```
class Child : public Parent1, Parent2 {
  public:
    Child() : Parent1("Alice"), Parent2("Bob") {...
}
```



Chaining without Default Constructors

- What if your classes have explicit constructors that take arguments? You need to explicitly chain
- Use **super** in Java:

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

public Student () {
   super("Bob");
}
```



Destructors

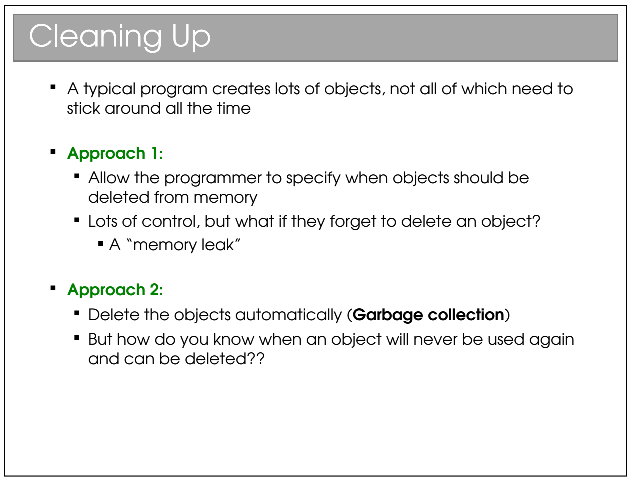
- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

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

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

  private :
    FILE *file;
}
```

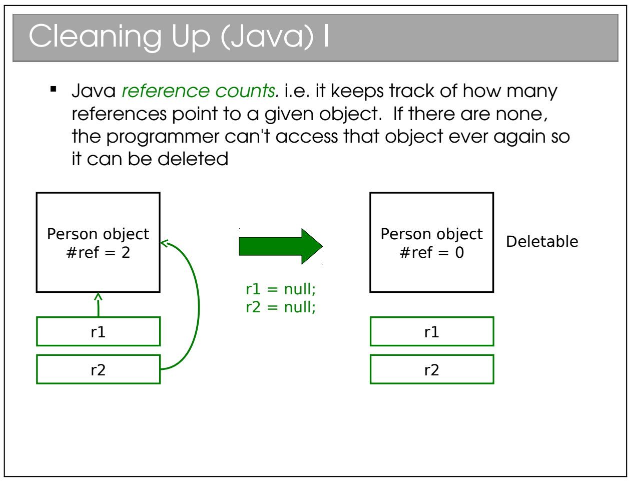
```
int main(int argc, char ** argv) {

  // 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.



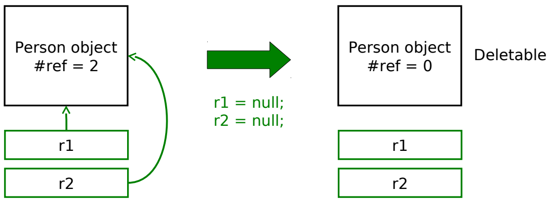
Cleaning Up

- 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??

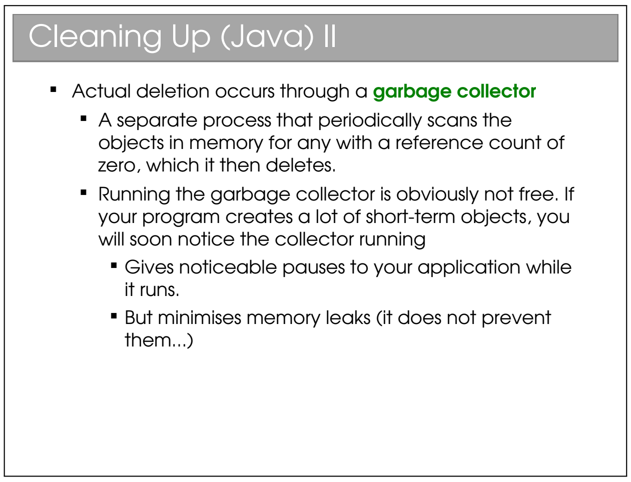


Cleaning Up (Java) I

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

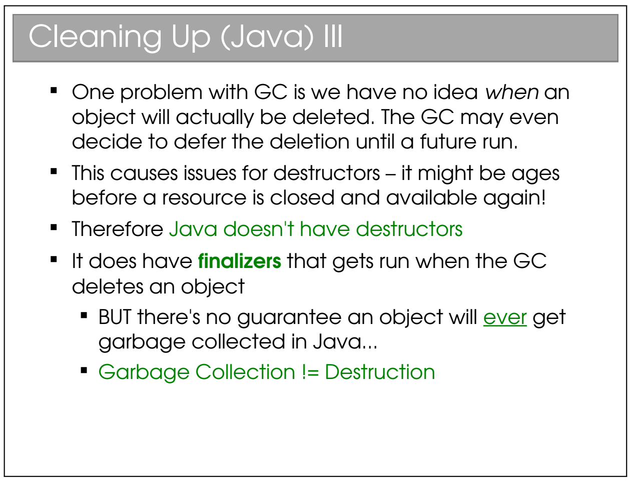


Note that reference counting has an associated cost - every object needs more memory (to store the reference count) and we have to monitor changes to all references to keep the counts up to date.



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…)

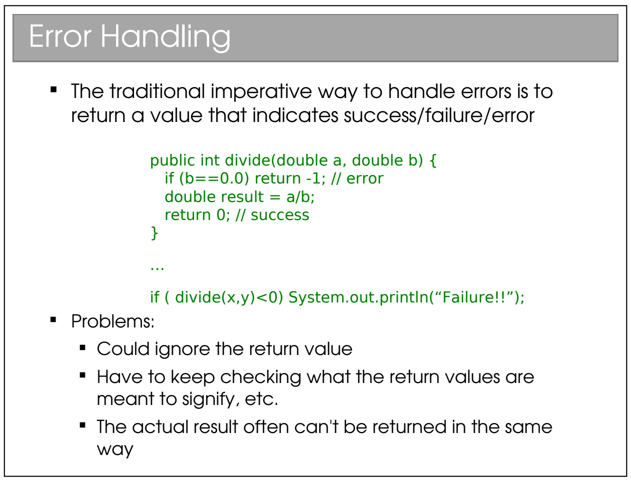


Cleaning Up (Java) III

- 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

Because you can't tell when finalizer methods will get called in Java, their value is greatly reduced. It's actually quite rare to see them in Java in my experience.

6.1 Exceptions



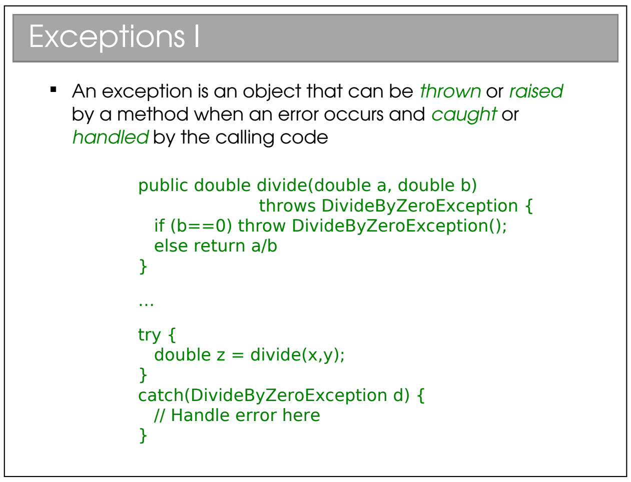
Error Handling

- The traditional imperative way to handle errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
  if (b==0.0) 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!

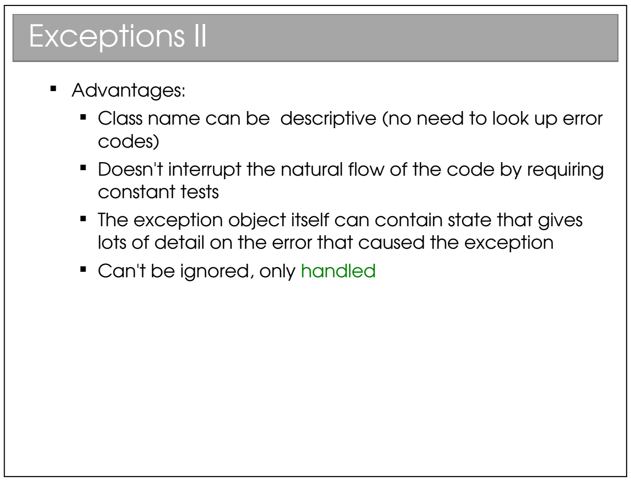


Exceptions I

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

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

...

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

Of course, you met exceptions in ML and there isn't much difference here. There is a tendency to use the terminology throw/catch rather than raise/handle in OOP languages—I don't know why.


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

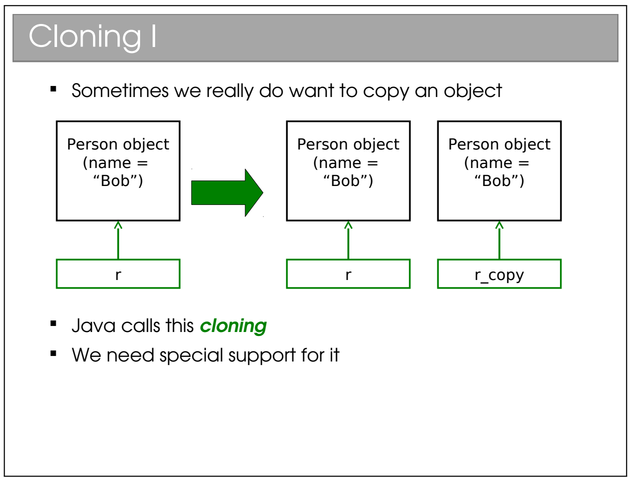
There is a *lot* more we could say about exceptions, but you have the basic tools to understand them and they will be covered in your practical Java course[1]. Just be aware that exceptions are very powerful and very popular in most modern programming languages. If you're struggling to understand them, take a look at:

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

[1]See workbook 4

Lecture 7

Copying and Cloning

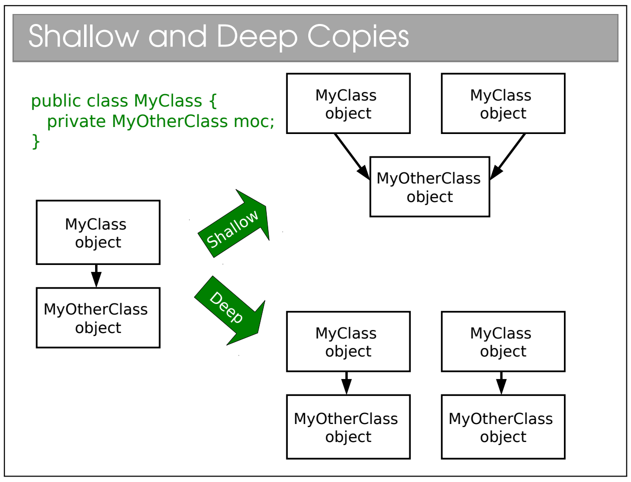


Cloning I

- Sometimes we really do want to copy an object

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

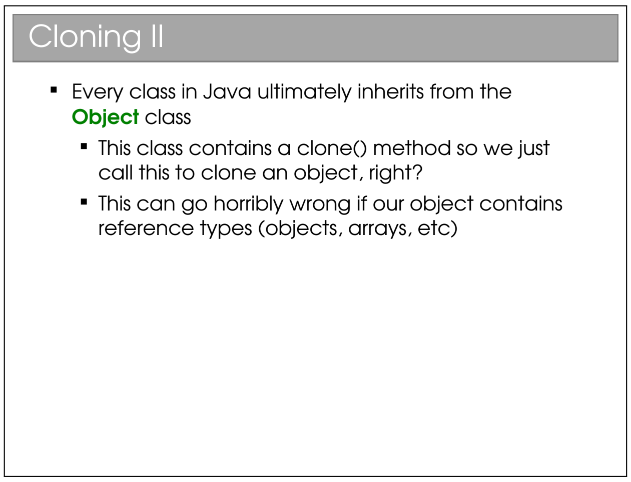
- Java calls this ***cloning***
- We need special support for it



Shallow and Deep Copies

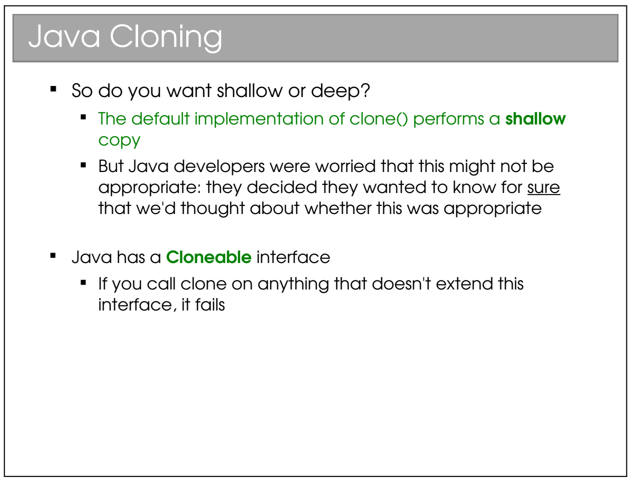
```
public class MyClass {
  private MyOtherClass moc;
}
```

MyClass object
MyOtherClass object
Shallow
Deep
MyClass object
MyClass object
MyOtherClass object
MyClass object
MyClass object
MyOtherClass object
MyOtherClass object



Cloning II

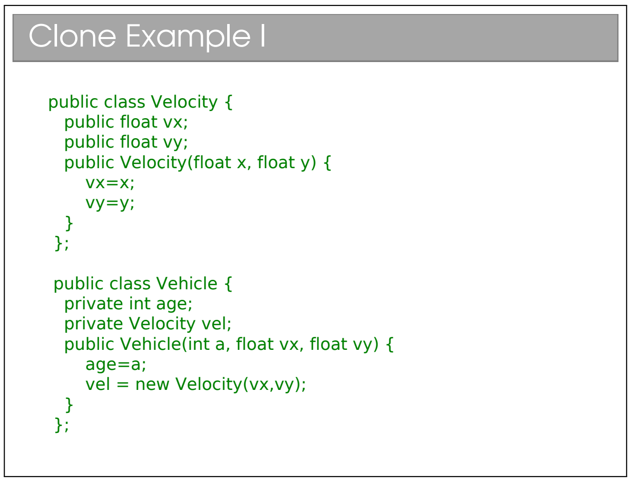
- Every class in Java ultimately inherits from the **Object** class
  - This class contains a clone() method so we just call this to clone an object, right?
  - This can go horribly wrong if our object contains reference types (objects, arrays, etc)



Java Cloning

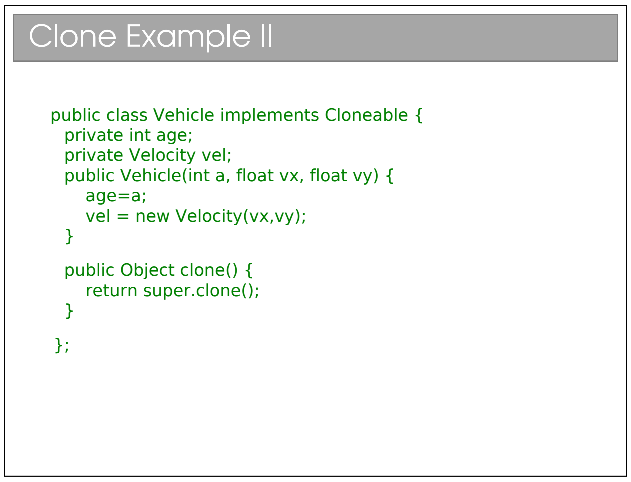
- So do you want shallow or deep?
  - The default implementation of clone() performs a **shallow** copy
  - But Java developers were worried that this might not be appropriate: they decided they wanted to know for sure that we'd thought about whether this was appropriate
- Java has a **Cloneable** interface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

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...



Clone Example I

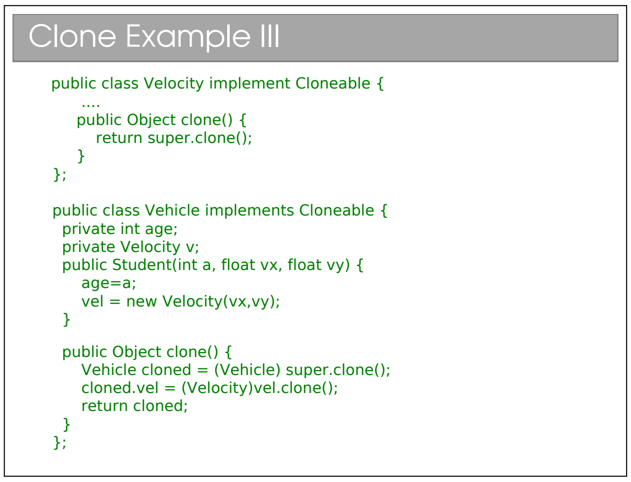
```
public class Velocity {
  public float vx;
  public float vy;
  public Velocity(float x, float y) {
     vx=x;
     vy=y;
  }
};

 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();
  }

 };
```

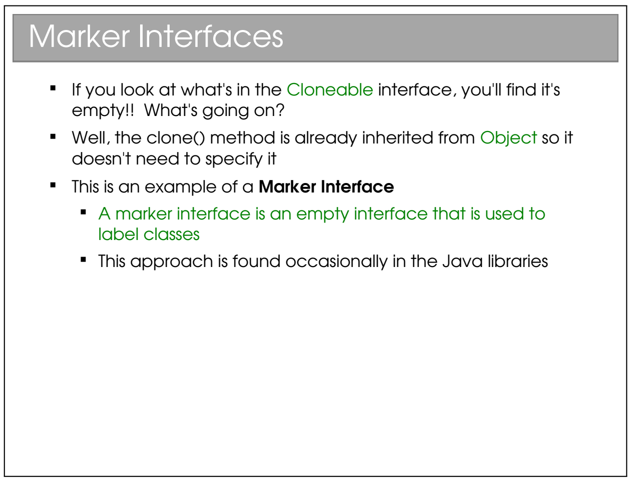
Here we fill in the clone() method using super.clone(). You can think of this as doing a byte-for-byte copy of an object in memory. Any primitive types (such as age) will therefore be copied. And references will also be copied, but not the objects they point to. Hence this much gets us a shallow copy.



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.



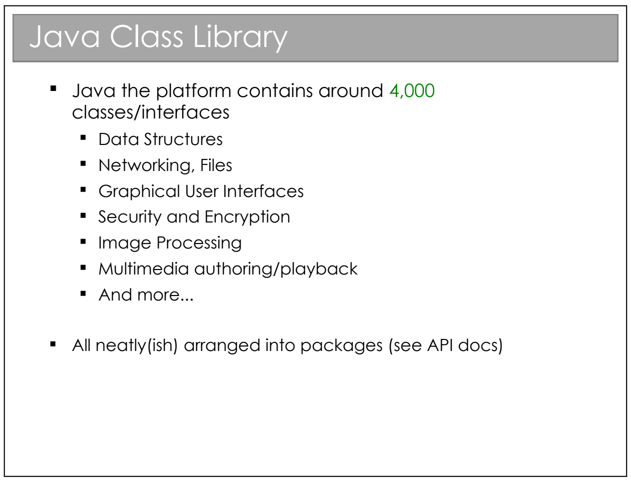
Marker Interfaces

- 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).

Lecture 8

Collections and Generics



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)

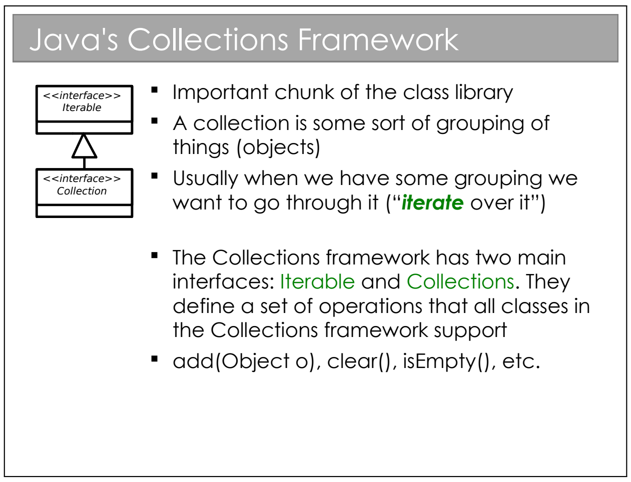
Remember Java is a *platform*, not just a programming language. It ships with a huge *class library*: that is to say that Java itself contains a big set of built-in classes for doing all sorts of useful things like:

- 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.

8.0.1 Collections and Generics

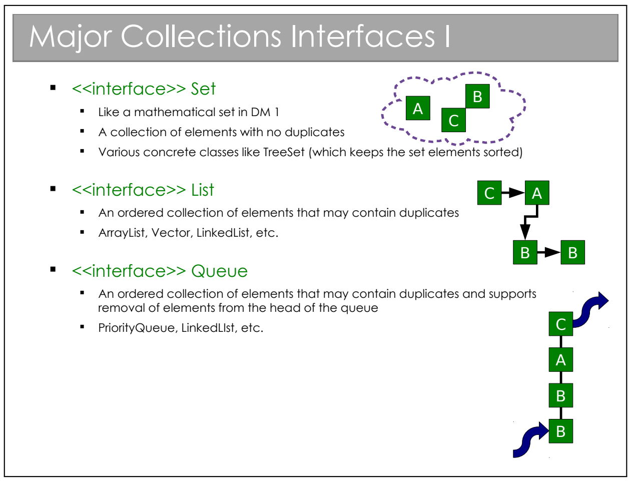


Java's Collections Framework

<<interface>>
Iterable
<<interface>>
Collection

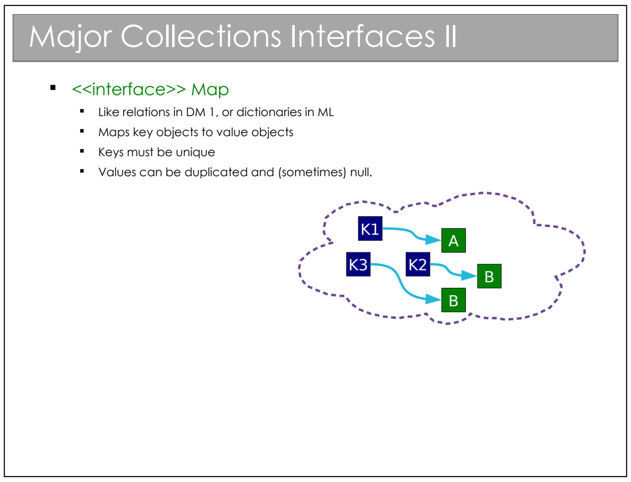
- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")
- The Collections framework has two main interfaces: Iterable and Collections. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

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



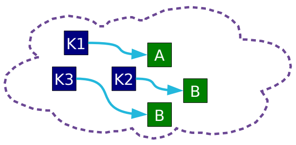
Major Collections Interfaces I

- <<interface>> Set
  - Like a mathematical set in DM 1
  - A collection of elements with no duplicates
  - Various concrete classes like TreeSet (which keeps the set elements sorted)
- <<interface>> List
  - An ordered collection of elements that may contain duplicates
  - ArrayList, Vector, LinkedList, etc.
- <<interface>> Queue
  - An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
  - PriorityQueue, LinkedList, etc.



Major Collections Interfaces II

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



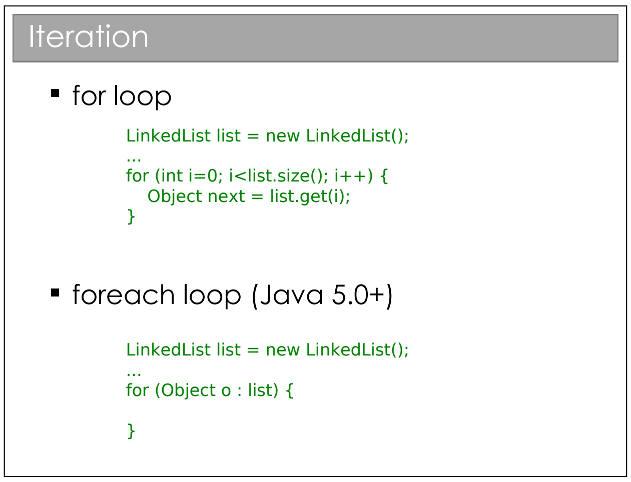
There are other interfaces in the Collections class, and you may want to poke around in the API documentation. In day-to-day programming, however, these are likely to be the interfaces you use.

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+)

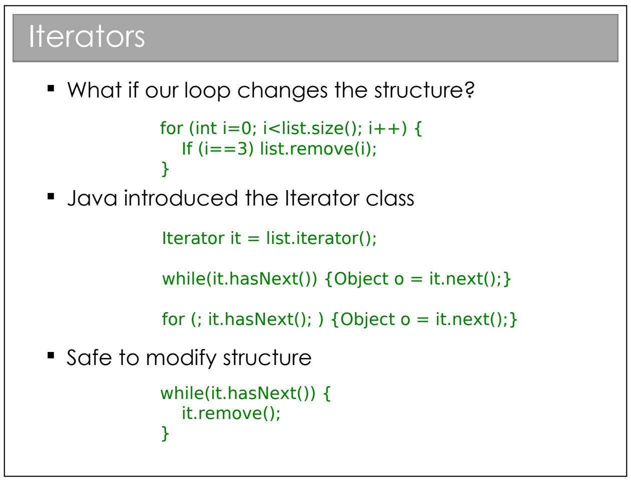
```
LinkedList list = new LinkedList();
...
for (Object o : list) {

}
```

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);
}
```

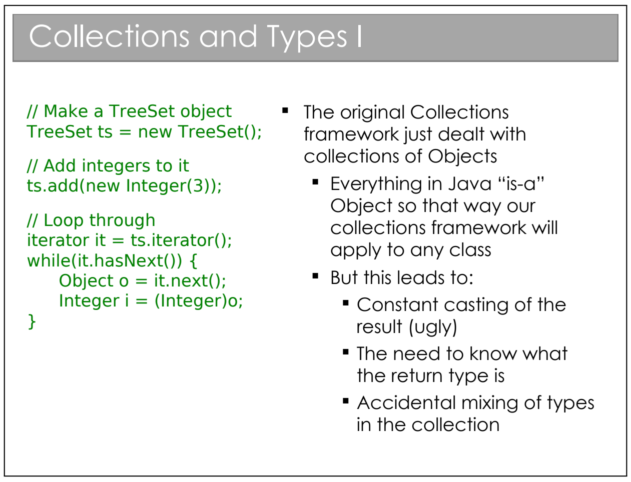
- Java introduced the Iterator class

```
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();
}
```

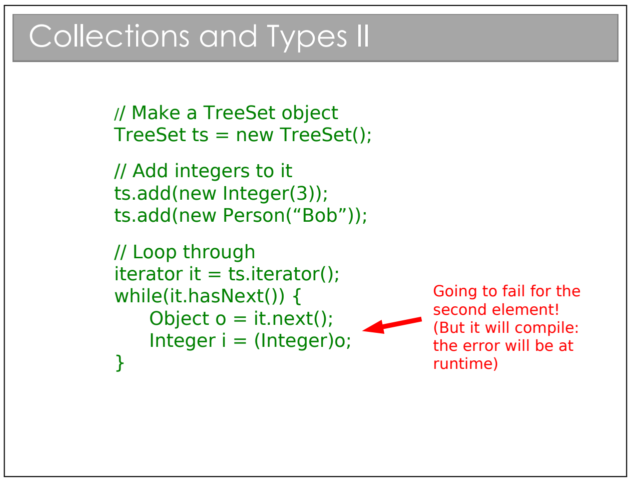
Note that the foreach structure isn't useful with Iterators. So we sacrifice some code readability for the ability to adjust the Collection's structure as we go.



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;
}
```

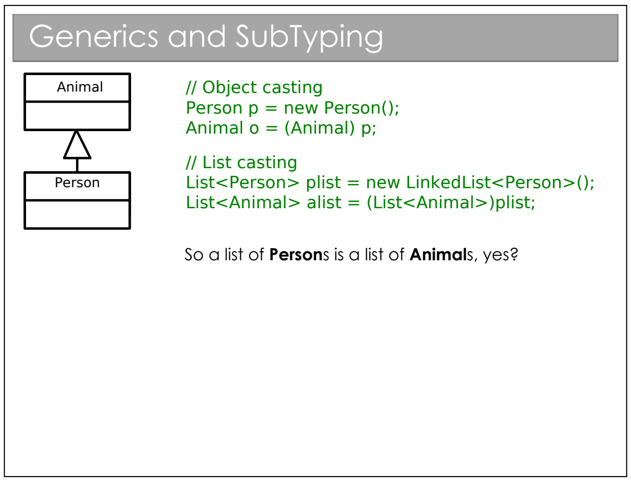
- The original Collections framework just dealt with collections of Objects
  - Everything in Java "is-a" Object so that way our collections framework will apply to any class
  - But this leads to:
    - Constant casting of the result (ugly)
    - The need to know what the return type is
    - Accidental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

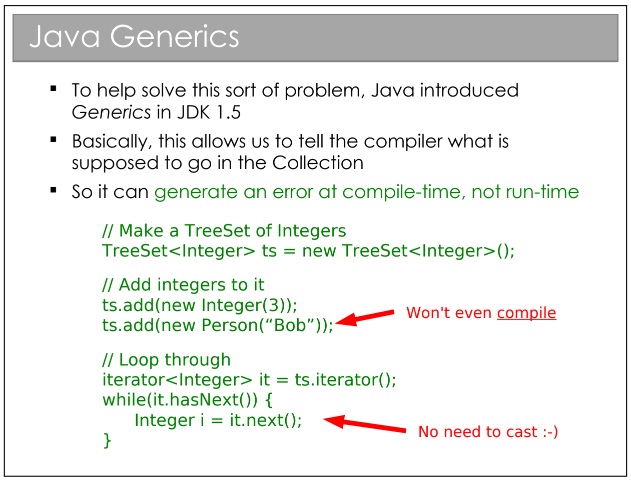


Generics and SubTyping

Animal
Person

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

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

So a list of **Person**s is a list of **Animal**s, yes?



Java Generics

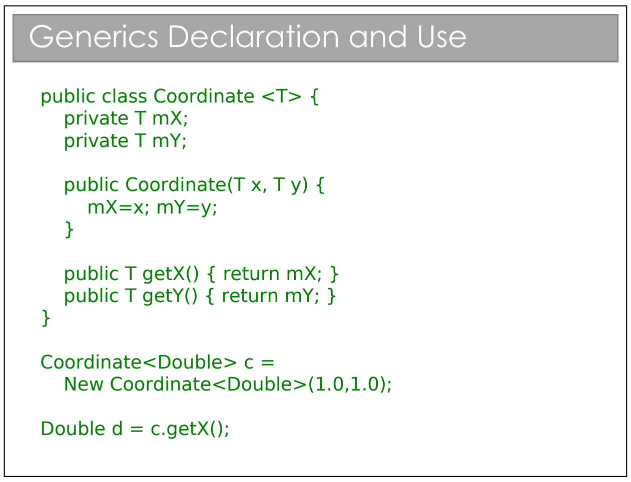
- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// 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"));

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

Won't even compile

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...

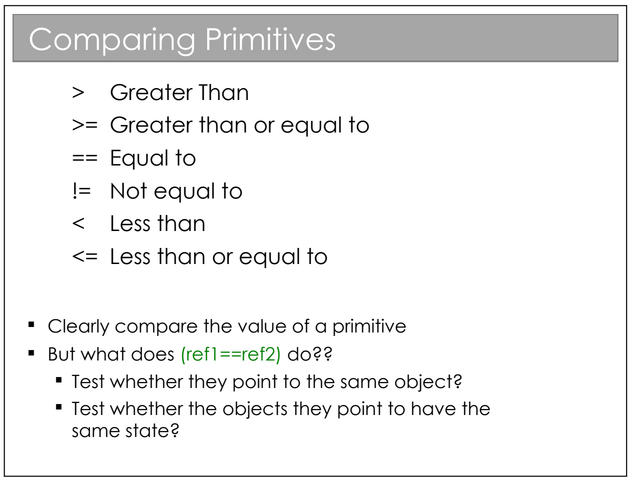


Generics Declaration and Use

```
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();
```

Lecture 9

Object Comparison

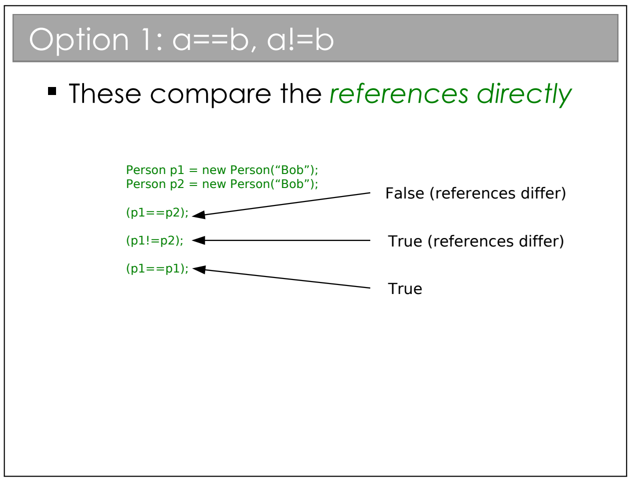


Comparing Primitives

>	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

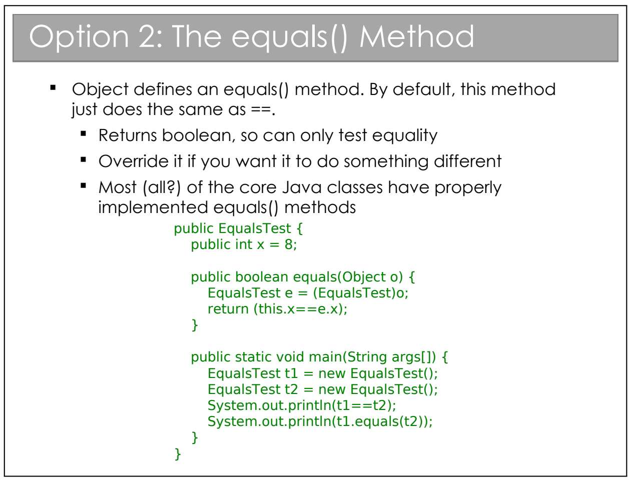
- These compare the *references directly*

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

(p1==p2);

(p1!=p2);

(p1==p1);
```

False (references differ)

True (references differ)

True

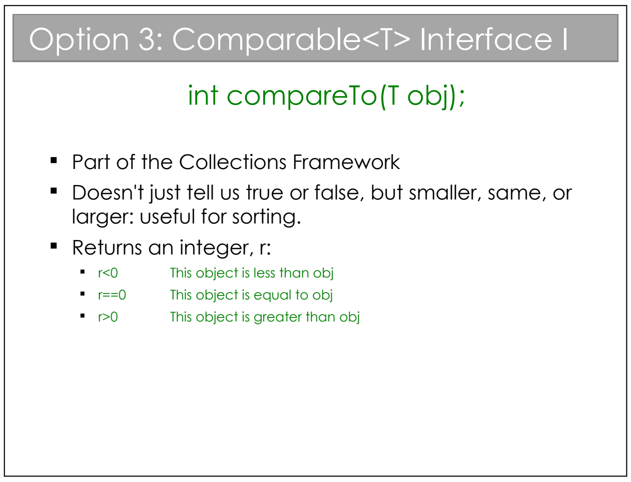


Option 2: The equals() Method

- 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));
   }
}
```

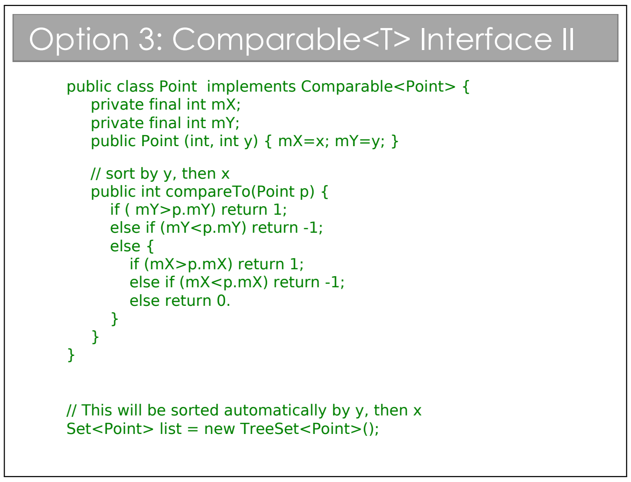
I find this mildly irritating: every class you use will support equals() but you'll have to check whether or not it has been overridden to do something other than ==. Personally, I only use equals() on objects from core Java classes, where I trust it to have been done properly.



Option 3: Comparable<T> Interface I

int compareTo(T obj);

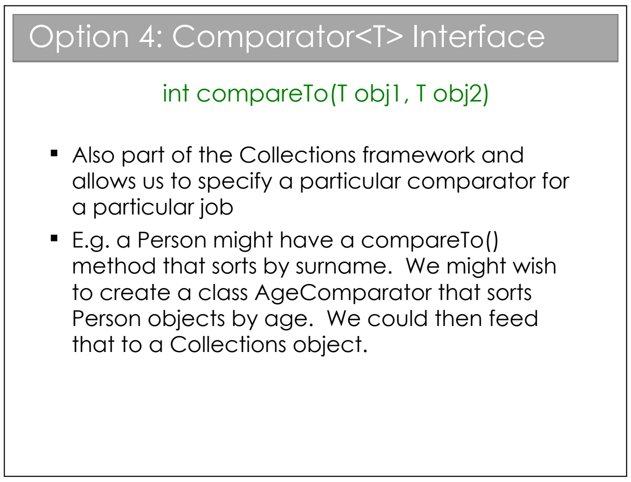
- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- Returns an integer, r:
  - r<0 This object is less than obj
  - r==0 This object is equal to obj
  - r>0 This object is greater than obj



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.

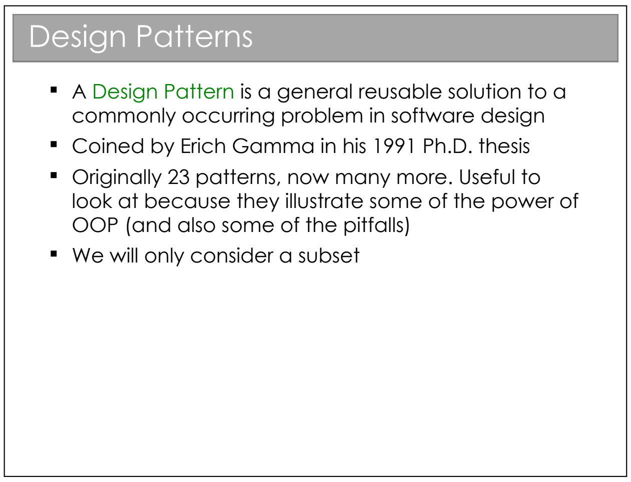
At first glance, it may seem that Comparator doesn't add much over Comparable. However it's very useful to be able to specify Comparators and apply them dynamically to Collections. If you look in the API, you will find that Collections has a *static* method sort(List l, Comparator c).

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());
```

Lecture 10

Design Patterns (2 lectures]



Design Patterns

- A Design Pattern is a general reusable solution to a commonly occurring problem in software design
- Coined by Erich Gamma in his 1991 Ph.D. thesis
- Originally 23 patterns, now many more. Useful to look at because they illustrate some of the power of OOP (and also some of the pitfalls)
- We will only consider a subset

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 solutions to them. 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 look at a few key patterns and how they are used.

10.0.2 So Design Patterns are like coding recipes?

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

10.0.3 Why Bother Studying Them?

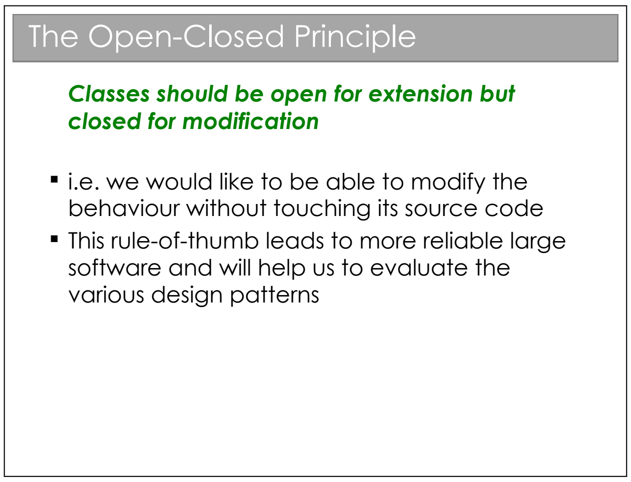
Design patterns are useful for a number of things, not least:

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?

10.0.4 The Open-Closed Principle

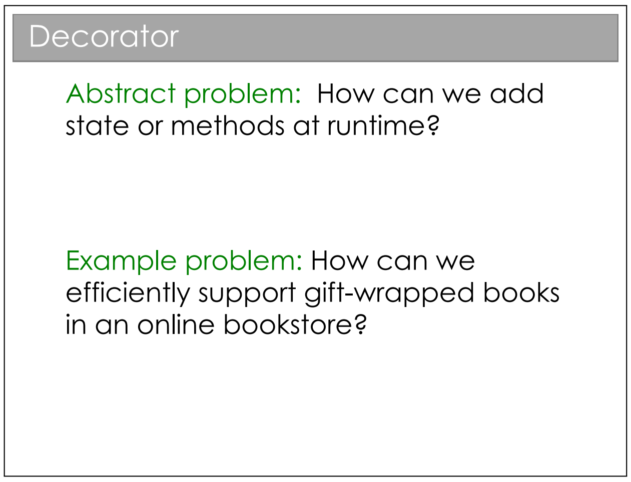

The Open-Closed Principle

Classes should be open for extension but closed for modification

- i.e. we would like to be able to modify the behaviour without touching its source code
- This rule-of-thumb leads to more reliable large software and will help us to evaluate the various design patterns

To help understand why this is helpful, it's useful to think about multiple developers using a software library. If they want to alter one of the classes in the library, they could edit its source code. But this would mean they had a customised version of the library that they wouldn't be able to update when new (bug-reduced) versions appeared. A better solution is to use the library class as a base class and implement the minor changes that are desired in the custom child. So, if you're writing code that others will use (and you should *always* assume you are in OOP) you should make it easy for them to extend your classes and discourage direct editing of them.

10.0.5 The Decorator Pattern



Decorator

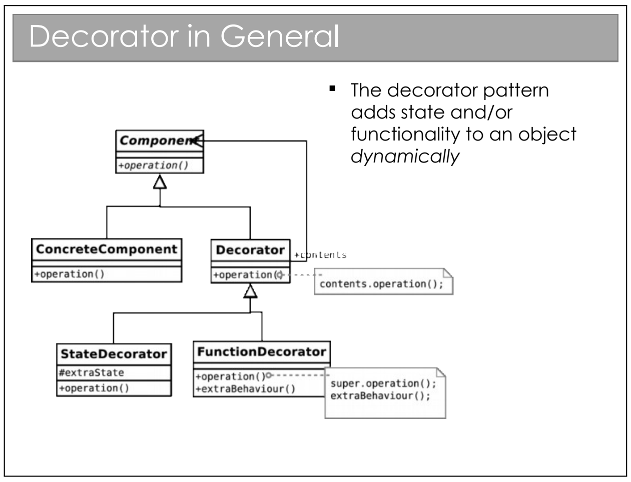
Abstract problem: How can we add state or methods at runtime?

Example problem: How can we efficiently support gift-wrapped books in an online bookstore?

Solution 1: Add variables to the established Book class that describe whether or not the product is to be gift wrapped.

Solution 2: Extend Book to create WrappedBook.

Solution 3: (Decorator) Extend Book to create WrappedBook and also add a member reference *to* a Book object. Just pass through any method calls to the internal reference, intercepting any that are to do with shipping or price to account for the extra wrapping behaviour.



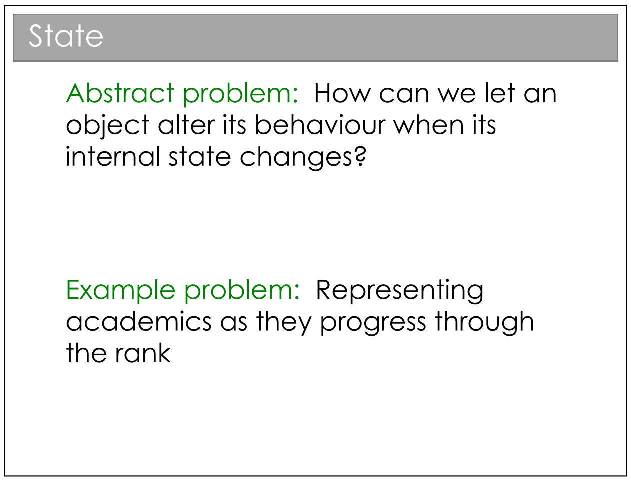
Decorator in General

- The decorator pattern adds state and/or functionality to an object *dynamically*

So we take an object 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.

10.0.6 The State Pattern



State

Abstract problem: How can we let an object alter its behaviour when its internal state changes?

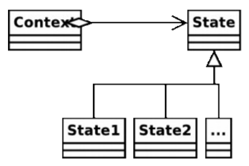
Example problem: Representing academics as they progress through the rank

Solution 1: Have an abstract Academic class which acts as a base class for Lecturer, Professor, etc.

Solution 2: Make Academic a concrete class with a member variable that indicates rank. To get rank-specific behaviour, check this variable within the relevant methods.

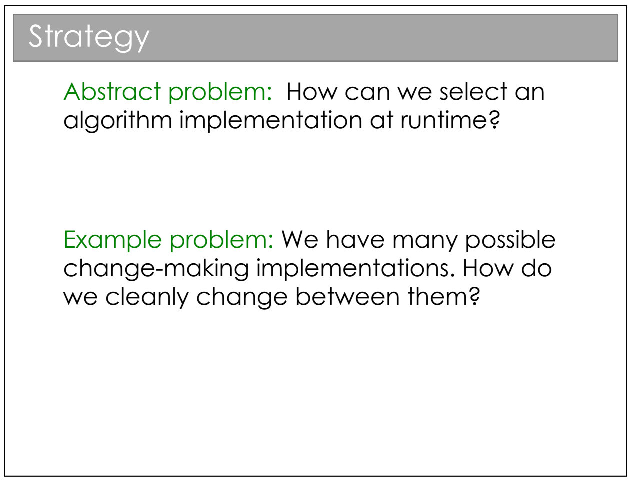
Solution 3: (State) Make Academic a concrete class that has-a AcademicRank as a member. Use AcademicRank as a base for Lecturer, Professor, etc., implementing the rank-specific behaviour in each..

State in General



- The state pattern allows an object to cleanly alter its behaviour when internal state changes

10.0.7 The Strategy Pattern



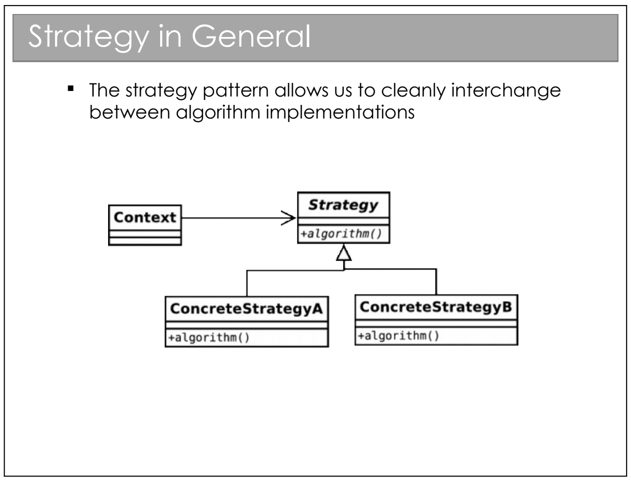
Strategy

Abstract problem: How can we select an algorithm implementation at runtime?

Example problem: We have many possible change-making implementations. How do we cleanly change between them?

Solution 1: Use a lot of if...else statements in the getChange(...) method.

Solution 2: (Strategy) Create an abstract ChangeFinder class. Derive a new class for each of our algorithms.



Strategy in General

- The strategy pattern allows us to cleanly interchange between algorithm implementations

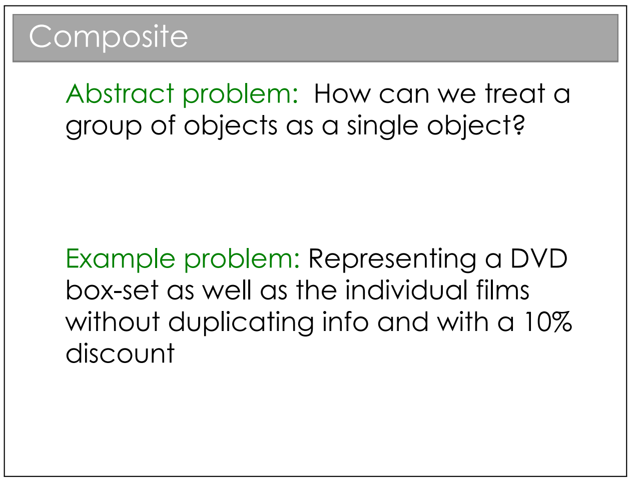
Note that this is essentially the same UML as the State pattern! The *intent* of each of the two patterns is quite different however:

- 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.

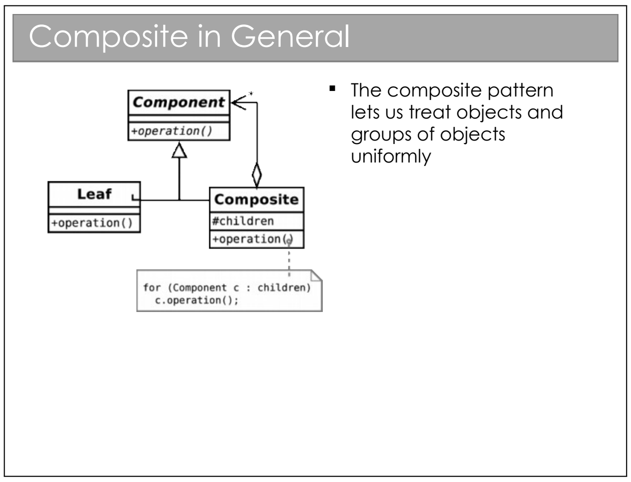
10.0.8 The Composite Pattern


Composite

Abstract problem: How can we treat a group of objects as a single object?

Example problem: Representing a DVD box-set as well as the individual films without duplicating info and with a 10% discount

The solution is fairly straightforward. We want to be able to treat a group of DVDs to just like a single DVD, so BoxSet inherits from DVD. To avoid repeating the description information and to keep pricing in sync, BoxSet must also have access to the constituent DVD objects.


Composite in General

- The composite pattern lets us treat objects and groups of objects uniformly

```
for (Component c : children)
  c.operation();
```

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

If you try to make a graphical representation of composites, you'll end up with some form of tree with each composite a node and each single entity a leaf. Many texts use this terminology when discussing the composite pattern.

10.0.9 The Singleton Pattern

Singleton

Abstract problem: How can we ensure only one instance of an object is created by developers using our code?

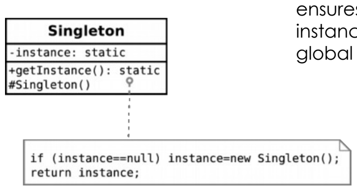
Example problem: You have a class that encapsulates accessing a database over a network. When instantiated, the object will create a connection and send the query. Unfortunately you are only allowed one connection at a time.

A valid solution to this is to make sure you close the database connection after using it, so you can just create Database objects every time you have a query. However, what if you forgot to close it? And what if making the connection was slow (they always are in computer time...).

Instead we exploit our access modifiers and create a private constructor (to ensure no-one can create objects at will) and add in a static member (the only instance we will ever have). Finally, we include a static getter for this member.

Ideally the instantiation of the Database should be *lazy*—i.e. only done on the first call to the getter.

Singleton in General



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

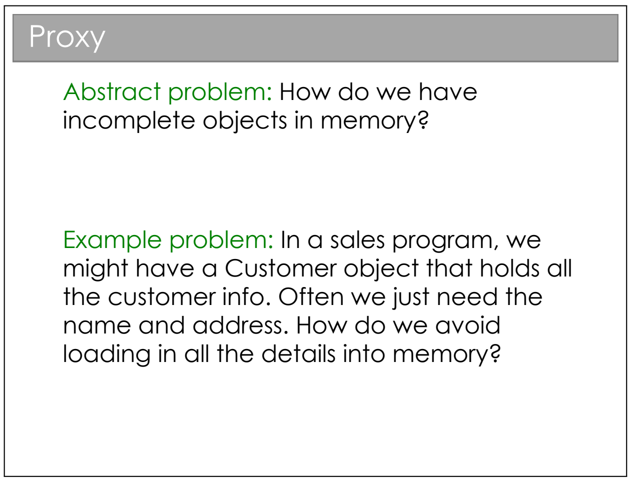
- The singleton pattern ensures a class has only one instance and provides global access to it

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. How could you ensure this doesn't happen?

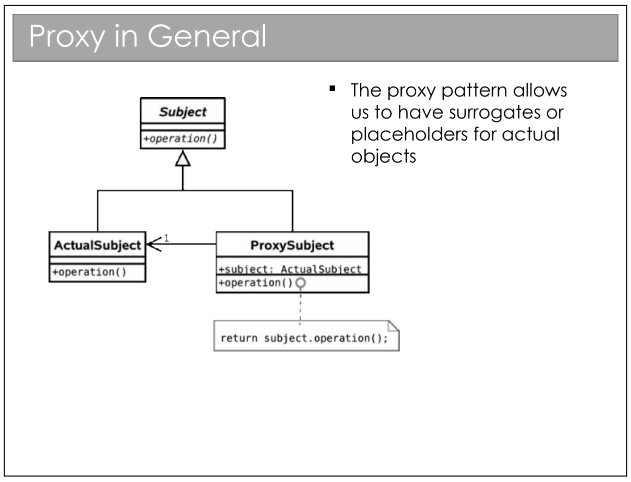
10.0.10 The Proxy Pattern



Proxy

Abstract problem: How do we have incomplete objects in memory?

Example problem: In a sales program, we might have a Customer object that holds all the customer info. Often we just need the name and address. How do we avoid loading in all the details into memory?



Proxy in General

- The proxy pattern allows us to have surrogates or placeholders for actual objects

Actually the Proxy pattern is often broken down into three possible intents, called Virtual Proxy, Remote Proxy, and Protection Proxy.

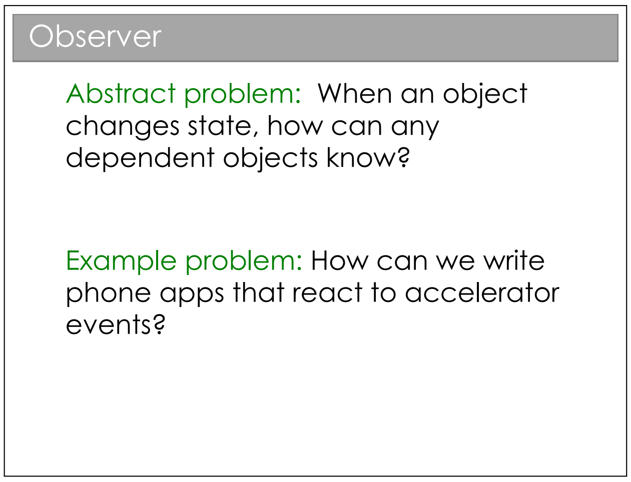
All three are based on the same general idea: we can have a placeholder class that has the same interface as another class, but actually acts as a pass through for some reason.

In the example here we need a virtual proxy, which stores some details locally, and passes the rest to the real object (which may need to be loaded from disk or something).

A protection proxy is essentially the same, but the intent is not to save memory so much as protect sensitive information. Perhaps our Customer class consists only of a name and a credit card number, the latter of which is clearly sensitive data. We ought to only have the credit card number in memory when we need it. A protection proxy would store the name locally, but override the getCreditCardNumber() method such that it only loads the information on-demand (possibly with an extra authentication requirement).

A remote proxy is used when we want the same object on multiple systems. E.g. Our sales system might have four or five back-end servers to spread the load of incoming account requests at any moment. Rather than copy all of the data to each machine, we might use proxy objects that simply pass through any requests to the 'head' server. Obviously, just doing this doesn't spread the load so we also have our proxy objects store anything that they get (this is *caching*).

10.0.11 The Observer Pattern



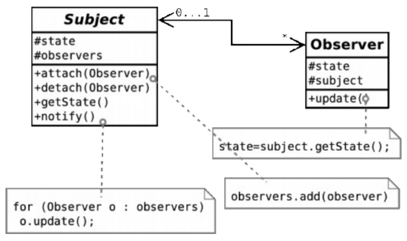
Observer

Abstract problem: When an object changes state, how can any dependent objects know?

Example problem: How can we write phone apps that react to accelerator events?

Observer in General

- The observer pattern allows an object to have multiple dependents and propagates updates to the dependents automatically.



This pattern is used regularly, but is particularly useful for event-based programs. The process is analogous to a magazine subscription: you *subscribe* with the publisher in order to receive *publish* events (magazines) as soon as they are available. In design patterns parlance, you are an observer of the publisher, who is the subject. It should be clear that this is also a very important pattern for the various proxy implementations if the source information might change during use.

In an Android smartphone, the system provides a subject in the form of a SensorManager object, which is actually a singleton (only one manager at any time). So we get it by calling:

```
SensorManager sManager = (SensorManager)
   getSystemService(SENSOR_SERVICE);
```

We then register with it with a line like:

```
sManager.registerListener(this,
        sManager.getDefaultSensor(
          Sensor.TYPE_ACCELEROMETER),
        SensorManager.SENSOR_DELAY_NORMAL);
```

Our class must implement SensorEventListener, which forces us to specify a onSensorEvent() method. Whenever the system gets a new accelerometer reading, it cycles over all the objects that have registered with it, feeding them the new reading.

10.1 Summary

From the original Design Patterns book:

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

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

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

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

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

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

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

10.1.1 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).

10.1.2 Other Patterns

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

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

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

10.1.3 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!].