



# Object Oriented Programming

## Dr Robert Harle

IA CST, PBS (CS) and NST (CS)  
Michaelmas 2014/15

# The Course

## The OOP Course

- So far you have studied **functional** programming (ML)
- Now we consider **imperative** programming (Java primarily but not exclusively).
- You have practicals in Java
  - This course **complements** the practicals
  - 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.

So far in this term you have been taught to program using the functional programming language ML. 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, especially since it directly maps to the maths functions you have met so far in your studies. In fact, if you try any other functional language (e.g. Haskell) 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. It is growing in popularity, but the dominant paradigm is undoubtedly *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<sup>1</sup> 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++ is a bit tricky...

The 'examinable' imperative language for Paper 1 is Java, and you won't be expected to program in any-

<sup>1</sup>some would say it gives you more rope to hang yourself with!

thing else. However, Java doesn't support everything we'll be looking at in this course (yet) 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 (albeit not examinable). We may also find time to try out Python and some other popular languages .

## Outline

1. Types, Objects and Classes
2. Pointers, References and Memory
3. Creating Classes
4. Inheritance
5. Polymorphism
6. Lifecycle of an Object
7. Error Handling
8. Copying Objects
9. Java Collections
10. Object Comparison
11. Design Patterns
12. Design Pattern (cont.)

## 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/current/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

## Types, Objects and Classes

### 1.1 Functional → Imperative

#### Types of Languages

- **Declarative** - specify what to do, not how to do it. i.e.
  - E.g. HTML describes what should appear on a web page, and not how it should be drawn to the screen
  - E.g. SQL statements such as "select \* from table" tell a program to get information from a database, but not how to do so
- **Imperative** – specify both what and how
  - E.g. "double x" might be a declarative instruction that you want the variable x doubled somehow. Imperatively we could have "x=x\*2" or "x=x+x"

Moving from ML to Java is fundamentally a shift from *functional* programming to *imperative* programming. Functional languages are a subclass of what is called *declarative* languages and we can summarise the differences here:

Declarative languages specify *what* should be done but not necessarily *how* it should be done. In a functional language such as ML you specify what you want to happen essentially by providing an example of how it can be achieved. The ML compiler/interpreter can do exactly that or something equivalent (i.e. it must give the same output or result).

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;

#### ML as a Functional Language

- **Functional** languages are a subset of **declarative** languages
  - ML is a functional language
  - It may appear that you tell it how to do everything, but you should think of it as providing an explicit example of what should happen
  - The compiler may **optimise** i.e. replace your implementation with something entirely different but 100% equivalent.

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 imperative programming constructs (e.g. references). 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<sup>1</sup>. So the boundaries are blurred, but ML is predominantly functional and Java predominantly imperative.

<sup>1</sup>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 you've written..!

## 1.2 Types

### Types and Variables

- Most imperative languages don't have type inference

```
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

In ML you created values of various type (*real*, *int*, etc). Sometimes you specified the type directly, but generally you tried 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. ML's type inference is a nice feature to have "baked in", although I acknowledge that ML's error messages about types could be a little less... cryptic.

In imperative languages, type inference is rare and it is more normal to manually specify the type of variables. 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.<sup>2</sup>

<sup>2</sup>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<sup>3</sup>. 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 Numerical Methods course.

## 1.3 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;

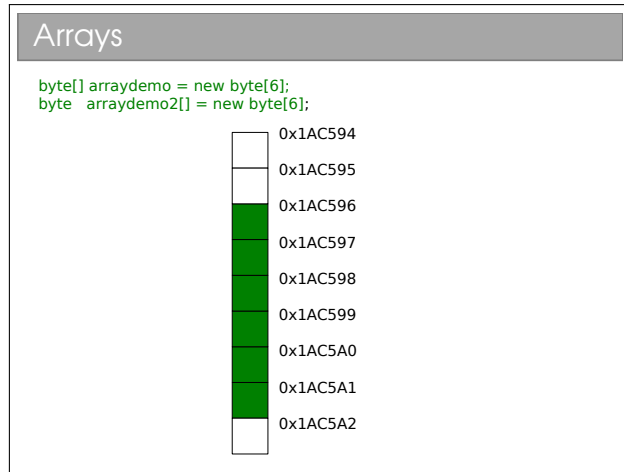
int x=9;
```

You may have noticed ML has *values* whilst other languages have *variables*. The difference is that ML's values are *immutable*—i.e. a chunk of memory is allocated for a value, and the contents of that chunk will never change. Calling it a variable would therefore be a misnomer since it can't vary!

<sup>3</sup>See workbook 1

Imperative languages are all about manipulating explicit state. That state is captured in variables (again, chunks of memory) but they can be changed to allow that manipulation.

## 1.4 Arrays



ML features tuples and lists as first class citizens of the language<sup>4</sup>. However, imperative languages feature *arrays* instead. An array is a set of values stored sequentially in a single chunk of memory and maps directly to the ML `Array` you have met, with the same properties:

- $O(1)$  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.

When you create an array, you specify the size: `float farray[] = new float[21];`. 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[]`)<sup>5</sup>.

## 1.5 Function Prototypes

Type inference meant that when you wrote a function ML you didn't specify much about the function in your

<sup>4</sup>See workbook 3

<sup>5</sup>See workbook 3

opening part. e.g.

```
fun myfun(a,b,c) = ...;
```

defines a function called `myfun` that takes three arguments. Without reading the full definition, we can't know whether the argument types are restricted, and we don't know what the possible types of the return value are. Of course, if we try using `myfun` incorrectly, the compiler will throw up a (cryptic!) error.

### Function Prototypes

- Functions are made up of a **prototype** and a **body**
  - Prototype specifies the function name, arguments and possibly return type
  - Body is the actual function code

```
fun myfun(a,b) = ...;
```

```
int myfun(int a, int b) {...}
```

In imperative programming, it is more normal to see full function prototypes (the bit before the actual code for the function, which is called the body) such as:

```
int myfun(int a,int b,int c) {...}
```

This tells us that it takes three integers as arguments and returns an integer as a result. But what if we also want it to work on floats? One option is to give a separate function prototype:

### Overloading Functions

- Same function name
- Different arguments
- Possibly different return type

```
int myfun(int a, int b) {...}
float myfun(float a, float b) {...}
double myfun(double a, double b) {...}
```

- But not just a different return type

```
int myfun(int a, int b) {...}
float myfun(int a, int b) {...}
```

✗

The compiler can easily choose the correct function to run based on the arguments you supply when you call

it. This approach is called *overloading* (because the name `myfun` has been in some sense overloaded). This is not something that you can do in ML (try it).

The obvious down side to is that you have to write out the function multiple times, one for each type! A better alternative is to parameterise the code—we'll return to that in a moment. Note that just changing the return type does not provide the compiler with a way to know which implementation to run, so can't be allowed.

## 1.6 Function Side Effects

### Function Side Effects

- Functions in imperative languages can use or alter larger system state → *procedures*

**Maths:**  $m(x,y) = xy$   
**ML:** `fun m(x,y) = x*y;`  
**Java:** `int m(int x, int y) = x*y;`

```
int y = 7;
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. By this we mean that calling it with the same arguments will always produce precisely the same result. e.g.

```
fun add(x,y)=x+y;
add(1,2);
```

will always output 3 regardless of the statements before or after it. In imperative programming, however, we have explicit system state and we could potentially use and change that state within a function. So:

```
int z=0; // this is some global state
int addimp(int x, int y) {
    'z=z+1;
    return x+y+z;
}
addimp(1,2);    // 4
addimp(1,2);    // 5
addimp(1,2);    // 6
```

Although terminologies differ, I call `addimp` a *procedure*. 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.

**Health warning:** Many languages today are imperative and many of them use the word 'function' as a synonym for 'procedure'. Even in these lectures I will use 'function' loosely. 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 introduces a layer of complexity into programming that *might* make the results better but the job harder.

### void Procedures

- A *void* procedure returns nothing:

```
int count=0;

void addToCount() {
    count=count+1;
}
```

It now makes sense for a procedure not to return *anything* at times—it may simply manipulate the external state in some way (this makes no sense in ML). In this case we label the return type `void`

If you turn back to the discussion of functional and imperative, you can hopefully see that a function with side effects (i.e. a procedure) 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 Control Flow

Generally speaking, the statements in imperative code are executed sequentially top-to-bottom. In order to handle conditional behaviour we need to support control flow. The associated statements are classified into *decision-making*, *looping* and *branching*.

### 1.7.1 Decision-Making Statements

#### Control Flow: Decision Making

```
if (boolean_expression) {  
    do_something()  
}
```

```
if (boolean_expression) {  
    do_something()  
}  
else {  
    do_something_else()  
}
```

These are familiar to you from ML, where you had `if...then...else`. In imperative languages you can also have just the `if...then` part (no `else`) because there is an option to do nothing (c.f. void procedures). In ML you had to one thing or the other: doing nothing breaks the type system since a function's return type could differ for the same input types).

Some languages support an explicit “else-if” option. E.g. python:

```
if (age<1):  
    crawl()  
elif (age<2):  
    totter()  
elif (age<75):  
    walk()  
else:  
    dodder()
```

This would look somewhat uglier in Java since we would need to *nest* multiple `if` statements<sup>6</sup>:

```
if (age<1) crawl();  
else {
```

<sup>6</sup>If you ever find needing to do this in Java or C++, check out the `switch` statement

```
if (age<2) totter();  
else {  
    if (age<75) walk();  
    else dodder();  
}  
}
```

### 1.7.2 Looping Statements

To loop some number of times in ML you have to use recursion. Whilst you obviously loved doing this, it's not the preferred way to loop in an imperative language (it's still there if you must, but it's not generally a good idea). Instead we have explicit loop constructs in the form of `for` and `while`:

#### Control Flow: Looping

```
for( initialisation; termination; increment )
```

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

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

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

```
while( boolean_expression )
```

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

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

These examples all loop eight times. The following code loops over the entirety of an array (the `for` approach is more usual for this task!):

#### Control Flow: Looping Examples

```
int arr[] = {1,2,3,4,5};
```

```
for (int i=0; i<arr.length;i++) {  
    System.out.println(arr[i]);  
}
```

```
int i=0;  
while (i<arr.length) {  
    System.out.println(arr[i]);  
    i=i+1;  
}
```



### 1.7.3 Branching

#### Control Flow: Branching I

- Branching statements interrupt the current control flow
- `return`
  - Used to return from a function at any point

```
boolean linearSearch(int[] xs, int v) {  
    for (int i=0; i<xs.length; i++) {  
        if (xs[i]==v) return true;  
    }  
    return false;  
}
```

#### Control Flow: Branching II

- Branching statements interrupt the current control flow
- `break`
  - Used to jump out of a loop

```
void linearSearch(int[] xs, int v) {  
    boolean found=false;  
    for (int i=0; i<xs.length; i++) {  
        if (xs[i]==v) {  
            found=true;  
            break; // stop looping  
        }  
    }  
    return found;  
}
```

#### Control Flow: Branching III

- Branching statements interrupt the current control flow
- `continue`
  - Used to skip the current iteration in a loop

```
void printPositives(int[] xs) {  
    for (int i=0; i<xs.length; i++) {  
        if (xs[i]<0) continue;  
        System.out.println(xs[i]);  
    }  
}
```

## 1.8 Custom types, Classes and Objects

Sooner or later, using just the built-in primitive types becomes restrictive. You saw this in ML, where you could create your own types. This is also possible in imperative programming and is, in fact, the crux of object oriented programming.

#### Custom Types

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

```
public class Vector3D {  
    float x;  
    float y;  
    float z;  
}
```

Let's take a simple example: representing 3D vectors (x,y,z). We could keep independent variables in our code. e.g.

```
float x3=0.0;  
float y3=0.0;  
float z3=0.0;
```

```
void add_vec(float x1, float y1, float z1,  
             float x2, float y2, float z2) {  
  
    x3=x1+x2;  
    y3=y1+y2;  
    z3=z1+z2;  
}
```

Clearly, this is not very elegant code. Note that, because I can only return one thing from a function, I can't return all three components of the answer (ML's tuples don't exist here: sorry!). Instead, I had to manipulate external state. In fact, you see a lot of this style of coding in procedural C coding. Yuk.

We would rather create a new type (call it `Vector3D`) that contains all three components, as per the slide. In OOP languages, the definition of such a type is called a *class*.

## 1.9 State *and* Behaviour

### State and Behaviour

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

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

public class Vector3D {
    float x;
    float y;
    float z;

    void add(float vx, float vy, float vz) {
        x=x+vx;
        y=y+vy;
        z=z+vz;
    }
}
```

What we've done so far looks a lot like procedural programming languages such as C. Here you create custom types to hold your data or state, then write a ton of functions/procedures to manipulate that state, and finally create your program by sequencing the various procedure calls appropriately. ML was similar: 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, the link is explicit since the class holds both the type and the helper functions that manipulate it.

OOP goes a step further, making the link explicit by having the class hold *both* the type and the helper functions that manipulate it. OOP classes therefore glue together both the state (i.e. variables) and the behaviour (i.e. functions or procedures).

## 1.10 Classes, Instances and Objects

### 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**

Whenever we create an *instance* of a class, we call it an *object*. The difference between a class and 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 procedures 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. Programs are made up of lots of objects, which we manipulate to get a result (hence “object-oriented programming”). e.g.<sup>7</sup>

```
// create an object of type Vector3D
Vector3D v1 = new Vector3D();
v1.x=3.5f; // set the x component

// create another object of type Vector3D
Vector3D v2 = new Vector3D();
v2.x=2.0f;
```

To summarise: Our *classes* group primitive variables and functions that operate on them to form a more complex, custom type. They act as templates to create specific *objects*.

<sup>7</sup>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**  
Fields  
Instance Variables  
Properties  
Variables  
Members

**Behaviour**  
Functions  
Methods  
Procedures

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!

## 1.11 Parameterised Types

### Parameterised Classes

- ML’s polymorphism allowed us to specify functions that could be applied to multiple types

```
> fun self(x)=x;  
val self = fn : 'a -> 'a
```

- In Java, we can achieve something similar through *Generics*; C++ through *templates*
  - Classes are defined with placeholders (see later lectures)
  - We fill them in when we create objects using them

```
LinkedList<Integer> = new LinkedList<Integer>()  
LinkedList<Double> = new LinkedList<Double>()
```

is called Generics). Initially, you will most likely encounter Generics when using Java’s built-in data structures such as `LinkedList`, `ArrayList`, `Map`, etc. For example, say you wanted a linked list of integers or `Vector3D` objects. You would declare:

```
LinkedList<Integer> lli = new LinkedList<Integer>();  
LinkedList<Vector3D> llv = new LinkedList<Vector3D>();
```

This was shoe-horned into Java relatively recently, so if you are looking at old code on the web or old books, you might see them using the non-Generics versions that ignore the type e.g. `LinkedList ll = new LinkedList()` allows you to throw almost anything into it (including a mix of types).

The astute amongst you may have noted that I used `LinkedList<Integer>` and not the more expected `LinkedList<int>`—it turns out that, in order to keep old code working, we imply can’t use primitive types directly in Generics classes. This is a java-specific irritation and we will be looking at why later on in the course. For now, just be aware that every primitive has associated with it an (immutable) *class* that holds a variable of that type. For example, `int` has `Integer`, `double` has `Double`, etc.

In ML, the type inference allowed you to write polymorphic functions; that is to write functions that could operate on different types. e.g. Your lists and sequences etc functioned for integers, reals, etc.

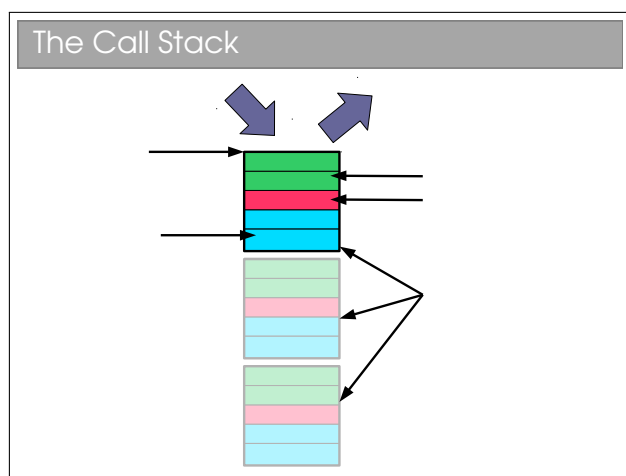
Many of the imperative/OOP languages did not have this ability originally, because they support a different type of polymorphism that we’ll come to in a later lecture. It turned out, however, there are still advantages to having the ML-style polymorphism and that has been added to some OOP languages, notably C++ (where it is called *templates* and Java (where it

## Lecture 2

# Pointers, References and Memory

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.

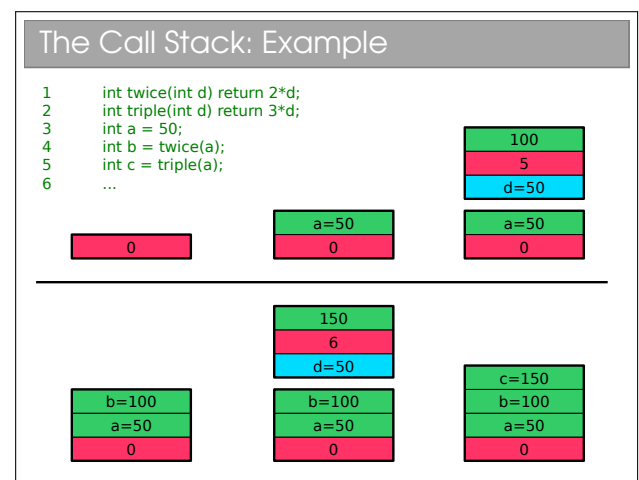
### 2.1 The Call Stack for Functions



Remember the way the fetch-execute cycle handles procedure calls<sup>1</sup>: 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*<sup>2</sup>. 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.

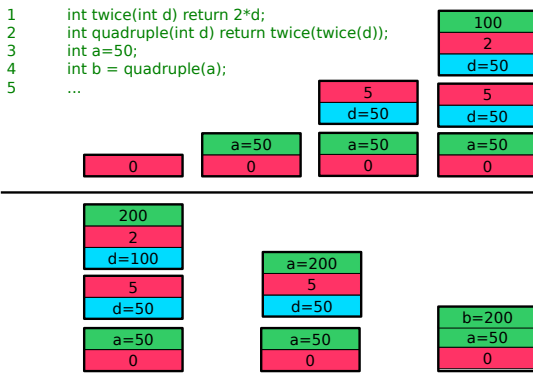


In this example I've avoided going down to assembly code and just assumed that the return address can be the code line number. This causes a small problem with e.g. line 4, which would be a couple of machine instructions (one to get the value of `twice()` and one to store it in `b`). I've just assumed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

<sup>1</sup>See the Computer Fundamentals course, lecture 2

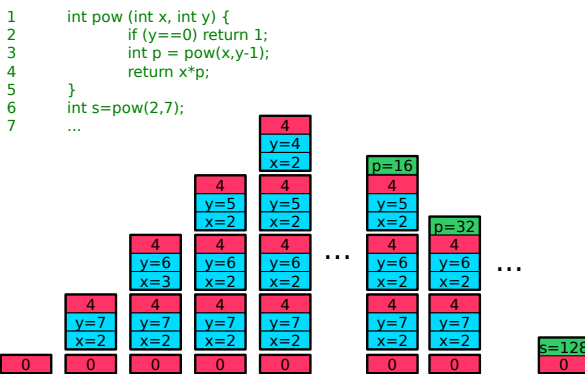
<sup>2</sup>See Algorithms next term for a full analysis

## Nested Functions



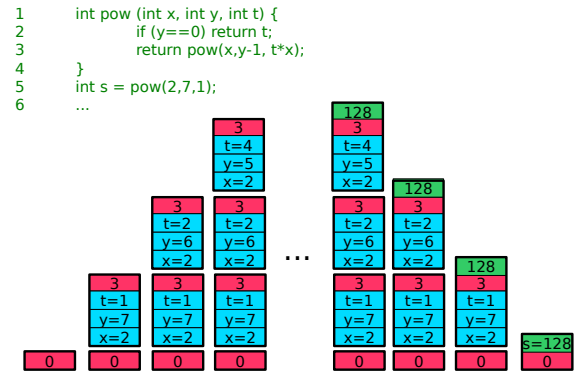
And even more interesting if we start processing recursively:

## Recursive Functions



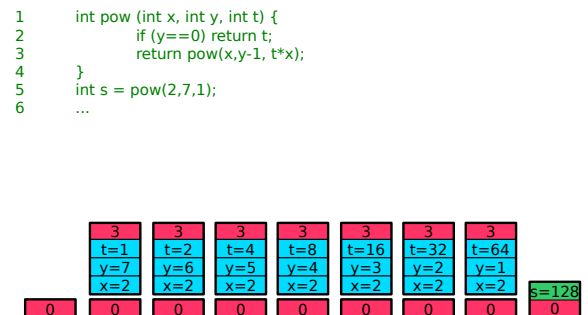
We immediately see a problem: computers only have finite memory so if our recursion is really deep, we'll be throwing lots of stack frames into memory and, sooner or later, we will run out of memory. We call this *stack overflow* and it is an unrecoverable error that you're almost certainly familiar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

## Tail-Recursive Functions I



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...<sup>3</sup>

## Tail-Recursive Functions II



## 2.2 The Heap

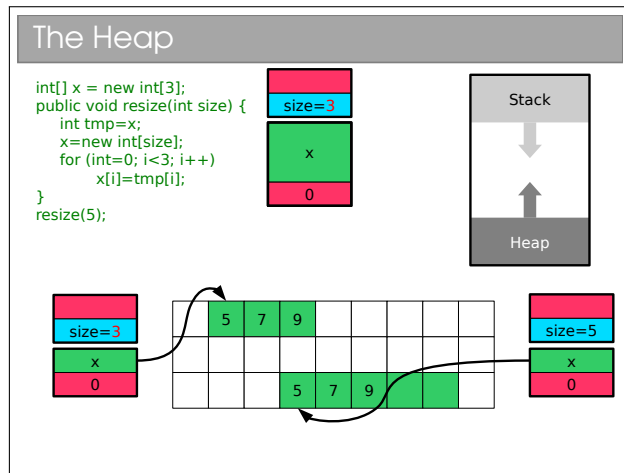
There's a subtlety with the stack that we've passed over until now. What if we want a function to create something that sticks around after the function is finished? Or to resize something (say an array)? We talk of memory being *dynamically* allocated rather than *statically* allocated as per the stack.

Why can't we dynamically allocate on the stack? Well, imagine that we do everything on a stack and you have a function that resizes an array. We'd have to grow the stack, but not from the top, but where the stack was put. This rather invalidates our stack and means that

<sup>3</sup>Language designers usually speak of 'tail-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.

every memory address we have will need to be updated if it comes after the array.

We avoid this by using a *heap*<sup>4</sup>. 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.



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

### Memory and Pointers

- In reality the compiler stores a mapping from variable name to a specific memory address, along with the type so it knows how to interpret the memory (e.g. “`x` is an `int` so it spans 4 bytes starting at memory address 43526”).
- Lower level languages often let us work with memory addresses directly. Variables that store memory addresses are called **pointers** or sometimes **references**.
- Manipulating memory directly allows us to write fast, efficient code, but also exposes us to bigger risks
  - Get it wrong and the program ‘crashes’.

The compiler must manipulate the computer’s mem-

<sup>4</sup>Note: you meet something called a ‘heap’ in Algorithms: it is NOT the same thing

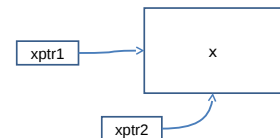
ory, but the notion of type doesn’t exist at the lowest level. Memory is simply a vast sequence of bits, split up (usually) into bytes, and the compiler must manually specify the byte it wants to read or change by it’s memory address. This is little more than a number uniquely identifying that byte. So when you ask for an `int` to be created, the compiler knows to find a 4-byte chunk of memory that isn’t being used (assuming `ints` are 32 bits) and change the bytes appropriately.

Some languages allow us, as programmers, to move beyond the abstraction of memory provided by explicit variable creation. They allow us to have variables that contain the actual memory addresses and even to manipulate them. We call such variables *pointers* and the traditional way to understand them is the “box and arrow” model:

### Pointers: Box and Arrow Model

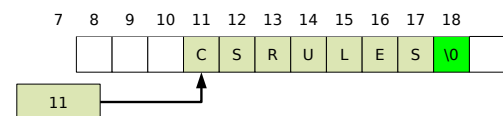
- A pointer is just the memory address of the first memory slot used by the variable
- The pointer **type** tells the compiler how many slots the whole object uses

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



### Example: Representing Strings I

- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka ‘\0’)
- So now we need to be able to store memory addresses → use **pointers**

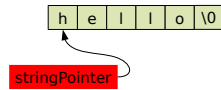


- We think of there being an **array** of characters (single letters) in memory, with the string pointer pointing to the first element of that array

## Example: Representing Strings II

```
char letterArray[] = {'h','e','l','l','o','\0'};
char *stringPointer = &(letterArray[0]);
printf("%s\n",stringPointer);

letterArray[3]='\0';
printf("%s\n",stringPointer);
```



In FoCS you encountered references, which were (sensibly back then) 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<sup>5</sup> 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.

<sup>5</sup>See workbook 3

## References vs Pointers

	Pointers	References
Represents a memory address	Yes	Yes
Can be arbitrarily assigned	Yes	<b>No</b>
Can be assigned to established object	Yes	Yes
Can be tested for validity	<b>No</b>	Yes

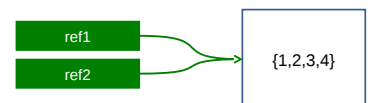
The last point is particularly important. A pointer points to something valid, something invalid, or `null` (a special zero-pointer that indicates it's not initialised). References, however, either point to something valid or to `null`. With a non-null reference, you know it's valid. With a non-null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

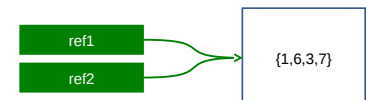
Sun decided that Java would have *only* references and no explicit pointers. Whilst slightly limiting, this makes programming much safer (and it's one of the many reasons we teach with Java). Java has two classes of types: *primitive* and *reference*. A primitive type is a built-in type<sup>6</sup>. Everything else is a reference type, including arrays and objects<sup>7</sup>.

## 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

<sup>6</sup>See Workbook 1

<sup>7</sup>See Workbook 3

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);
```

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<sup>8</sup>, 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.

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<sup>9</sup>.

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.

<sup>8</sup>See workbook 3

<sup>9</sup>Don't believe me? See the Java specification, section 8.4.1.



### 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"

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!

### Passing Procedure Arguments In C

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

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

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

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

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

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

Here I have made a fictional type **HugeInt** which is meant to represent something that takes a lot of space in memory. Calling either of these functions will give

# Lecture 3

## Creating Classes

### 3.1 Identifying Classes

#### What Not to Do

- Your ML has doubtless been one big file where you threw together all the functions and value declarations
- Lots of C programs look like this :-(
  - *We could emulate this in OOP by having one class and throwing everything into it*
- We can do (much) better

Having one massive class, MyApplication perhaps, with all the state and behaviour in it, is a surprisingly common novice error. This achieves nothing (in fact it just adds boilerplate code). Instead we aim to have multiple classes, each embodying a well-defined *concept*.

#### Identifying Classes

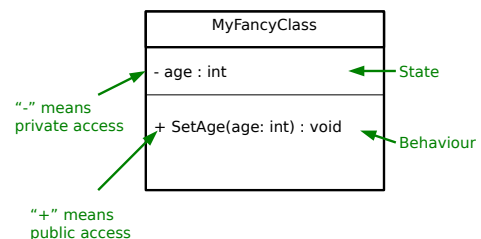
- We want our class to be a *grouping of conceptually-related state and behaviour*
- One popular way to group is using grammar
  - *Noun → Object*
  - *Verb → Method*

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

might have an object to represent the table; to represent each ball; to represent the cue; etc. Identifying the best possible set of classes for your program is more of an art than a science and depends on many factors. However, it is usually straightforward to develop sensible classes, and then we keep on refining them on them ("refactoring") until we have something better.

A helpful way to break your program down is in term of tangible things—represented by the nouns you would use when describing the program. Similarly, the verbs often map well to the behaviour required of your classes. Think of these as guidelines or rules of thumb, not rules.

#### UML: Representing a Class Graphically

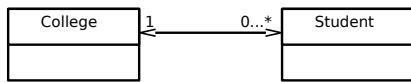


The graphical notation used here is part of UML (Unified Modeling Language). UML is a standardised set of diagrams that can be used to describe software independently of any programming language used to implement it.

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.

Very often classes follow naturally from the problem domain. So, if you are making a snooker game, you

## 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();
    }
}
  
```

Annotations:

- Class name: `MyFancyClass`
- Access modifier: `public`
- Class state (properties that an object has such as colour or size): `public int someNumber;` and `public String someText;`
- Class behaviour (actions an object can do): `public void someMethod() { }`
- 'Magic' start point for the program (named main by convention): `public static void main(String[] args) { }`
- Create a reference to a MyFancyClass object and call it c: `MyFancyClass c =`
- Create an object of type MyFancyClass in memory: `new MyFancyClass();`

## 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();
}
  
```

Annotations:

- Class name: `MyFancyClass`
- Access modifier: `public`
- Class state: `int someNumber;` and `public String someText;`
- Class behaviour: `void someMethod() { }`
- 'Magic' start point for the program: `void main(int argc, char **argv) { }`
- Create an object of type MyFancyClass and call it cc: `MyFancyClass c;`
- Create a pointer to a MyFancyClass object and call it cp: `MyFancyClass *cp =`
- Create an object of type MyFancyClass and return a reference to it: `new MyFancyClass();`

## 3.2 OOP Concepts

### OOP Concepts

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

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

### 3.2.1 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 a common Computer Science 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. Then, when we sequence them together to make our larger program, there are far fewer places where it can go wrong.

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*.

As an aside, modularity often goes further than the classes/objects. Java has the notion of **packages** to group together classes that are conceptually linked. C++ has a similar concept in the form of namespaces.

### 3.2.2 Encapsulation and Information Hiding

#### 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 directly<sup>1</sup>.

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 that allows full interaction with it, but should expose nothing about its internal state. The general rule you can follow 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 can only ever store sensible values.

#### Encapsulation III

<pre>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;} }</pre>	<pre>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);} }</pre>
---	---

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.

<sup>1</sup>See workbook 3

## Access Modifiers

	Everyone	Subclass	Same package (Java)	Same Class
<b>private</b>				X
<b>package (Java)</b>			X	X
<b>protected</b>		X	X	X
<b>public</b>	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**. Don't worry if you don't yet know what a "Subclass" is—that's in the next lecture.

### 3.3 Immutability

The discussion of access modifiers leads us naturally to talk about immutability. You should recall from FoCS that every value in ML is immutable: once it's set, it can't be changed. From a low-level perspective, writing `val x=7;` allocates a chunk of memory and sets it to the value 7. Thereafter you can't change that chunk of memory. You *could* reassign the *label* by writing `val x=8;` but this sets a new chunk of memory to the value 8, rather than changing the original chunk (which sticks around, but can't be addressed directly now since `x` points elsewhere).

It turns out that immutability has some serious advantages when concurrency is involved—knowing that nothing can change a particular chunk of memory means we can happily share it between threads without worry of contention issues. It also has a tendency to make code less ambiguous and more readable. It is, however, more efficient to manipulate allocated memory rather than constantly allocate new chunks. In OOP, we can have the best of both worlds.

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

### 3.4 Creating Parameterised Types

#### Creating Parameterised Types

- These just require a placeholder type

```
class Vector3D<T> {
    private T x;
    private T y;

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

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

We already saw how to use Generics types in Java (e.g. `LinkedList<Integer>`). Declaring them is not much

harder than a ‘normal’ class. The T is just a placeholder (and I could have used any letter or word—T is just the de-facto choice). Once declared we can create Vector3D objects with different underlying storage types, just like with LinkedList:

```
// Vector of integers
Vector3D<Integer> vi = new Vector3D<Integer>();
// Vector of single precision reals
Vector3D<Float> vi = new Vector3D<Float>();
// Vector of double precision reals
Vector3D<Double> vi = new Vector3D<Double>();
```

There is no problem having parameterised types as parameters—for example `LinkedList<Vector3D<Integer>>` declares a list of integer vector objects. And we can have multiple parameters in our definitions:

```
public class Pair<U,V> {
    private U mFirst;
    private V mSecond;
    ...
}
```

You see this most commonly with Maps in Java, which represent dictionaries, mapping keys of some type to values of (potentially) some other type. e.g. a `TreeMap<String,Integer>` could be used to map names to ages).

## 3.5 Static Data

## 3.6 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;
    ...
}
```

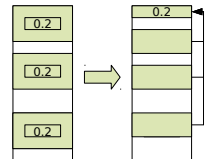
One of these created every time a new ShopItem is instantiated. Nothing keeps them all in sync.

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) {...}
}
```

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

vs

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

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

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

# Lecture 4

## Inheritance

### Inheritance I

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

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

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

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

### Inheritance II

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

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

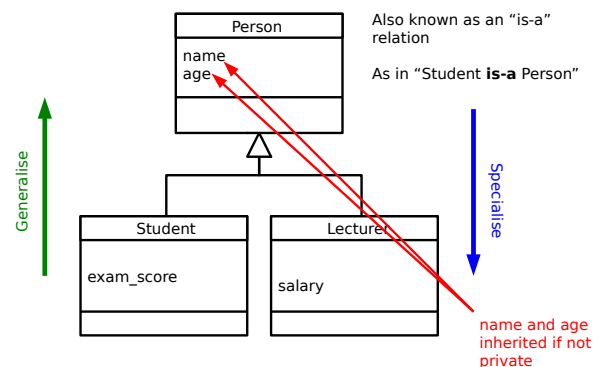
```
class Lecturer extends Person {  
    public int salary;  
}
```

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

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 {...};
```

### Representing Inheritance Graphically



Inheritance<sup>1</sup> 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 B extends A, 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’.

<sup>1</sup>See workbook 5

## 4.1 Casting

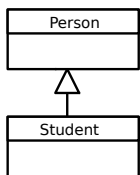
### Casting

- Many languages support *type casting* between numeric types

```
int i = 7;
float f = (float) i; // f==7.0
double d = 3.2;
int i2 = (int) d; // i2==3
```

- With inheritance it is reasonable to type cast an object to any of the types above it in the inheritance tree...

### Widening



- 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;
```

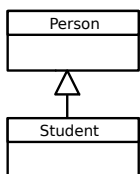
"Casting"

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

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

Implicit cast

### Narrowing



- 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) s;
Student s2 = (Student) p;
```

OK because underlying object  
really 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.

Note the example of casting primitive numeric types in the slide is a bit different, since a new variable of the primitive type is created and assigned the relevant value.

## 4.2 Shadowing

### 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<sup>2</sup>.

### 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;
    }
}
```

<sup>2</sup>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.



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* non-private field is inherited by a subclass.

There are two new keywords that have appeared here: **super** and **this**. The **this** keyword can be used in any class method<sup>3</sup> 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.

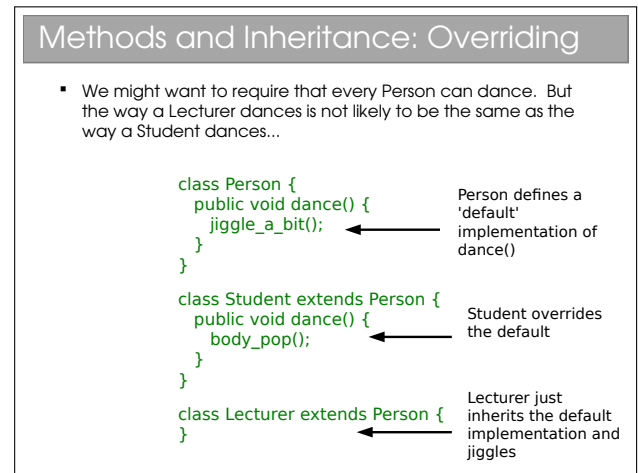
## 4.3 Overloading

We have already discussed function overloading, where we had multiple functions with the same name, but a different prototype (i.e. set of arguments). The same is possible within classes.

<sup>3</sup>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.

## 4.4 Overriding

The remaining question is what happens to methods when they are inherited and rewritten in the child class. The obvious possibility is that they are treated the same as fields, and shadowed. When this occurs we say that the method is *overridden*. As it happens, we can't do this in Java, but it is the default in C++ so we can use that to demonstrate:



Every object that has **Person** for a parent must have a **dance()** method since it is defined in the **Person** class and is inherited. If we override it in **Child** then **Child** objects will behave differently. There are some subtleties to this that we'll return to next lecture.

A useful habit to get into is to annotate every function you override using **@Override**. This serves two purposes: firstly it tells anyone reading the code that it's an overridden method; secondly it allows the compiler to check it really does override something. It's surprisingly easy to make a typo and think you've overridden but actually not. We'll see this later when we discuss object comparison.

## 4.5 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.

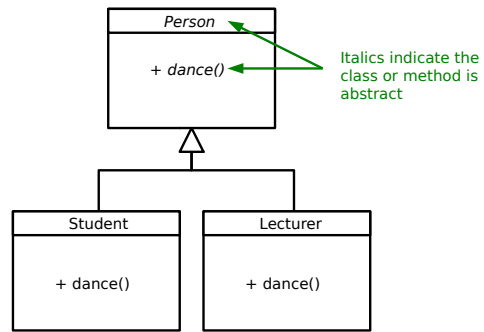
<pre>public <b>abstract</b> class Person {     public <b>abstract</b> void dance(); }</pre>	<pre>class Person {     public:         virtual void dance()=0; }</pre>
Java	C++

- 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.

# Lecture 5

## Polymorphism

You should be comfortable with the polymorphism<sup>1</sup> that you met in FoCS, where you wrote functions that could operate on multiple types. It turns out that is just one type of polymorphism in programming, and it isn't the form that most programmers mean when they use the word. To understand that, we should look back at our overridden methods:

### 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

If we can get different method implementations by casting the same object to different types, we have

<sup>1</sup>The etymology of the word polymorphism is from the ancient Greek: *poly* (many)–*morph* (form)–ism

static polymorphism. In general static polymorphism refers to anything where decisions are made at compile-time (so-called early binding). You may realise that all the polymorphism you saw in ML was static polymorphism. The shadowing of fields also fits this description.

### Polymorphic Concepts II

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

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

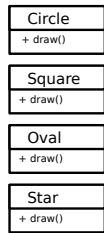
- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in Student

Here we get the same method implementation regardless of what we cast the object to. In order to be sure that it gets this right, we can't figure out which method to run when we are compiling. Instead, the system has to run the program and, when a decision needs to be made about which method to run, it must look at the actual object in memory (regardless of the type of the reference, which may be a cast) and act appropriately.

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 (so-called late binding) 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

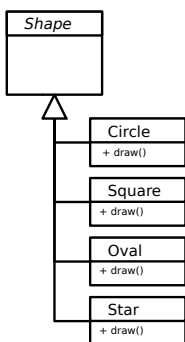


- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects
- **Option 1**
  - Keep a list of Circle objects, a list of Square objects,...
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?

## 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...

## The Canonical Example II



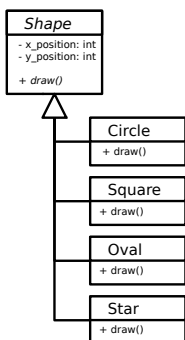
- **Option 2**
    - Keep a single list of Shape references
    - Figure out what each object really is, narrow the reference and then draw()
- ```

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?

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!

## The Canonical Example III



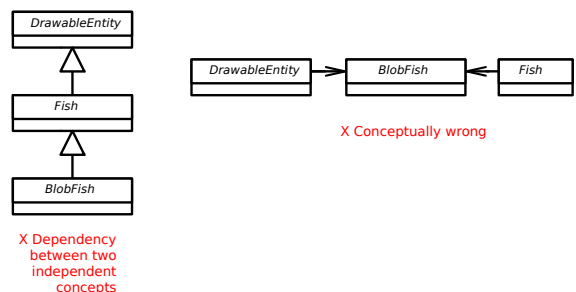
- **Option 3 (Polymorphic)**
    - Keep a single list of Shape references
    - Let the compiler figure out what to do with each Shape reference
- ```

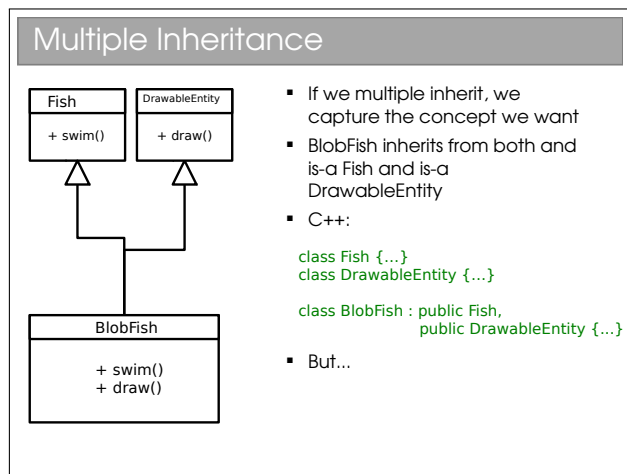
For every Shape s in myShapeList
  s.draw();
  
```
- What if we want to add a new shape?

## 5.1 Multiple Inheritance and Interfaces

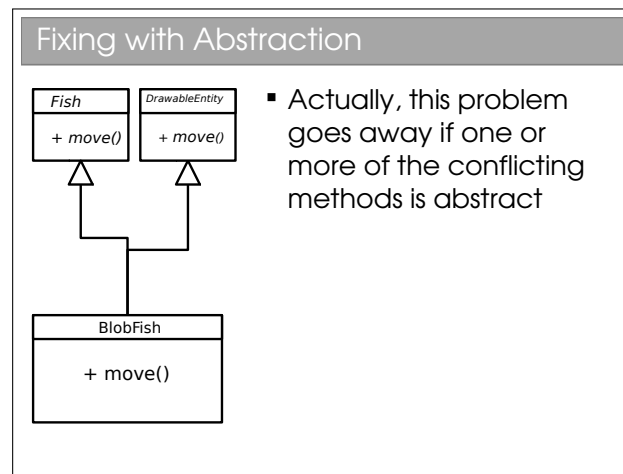
### Harder Problems

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

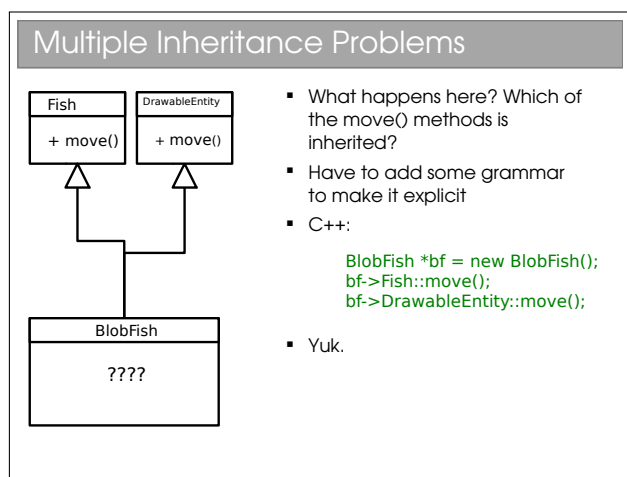




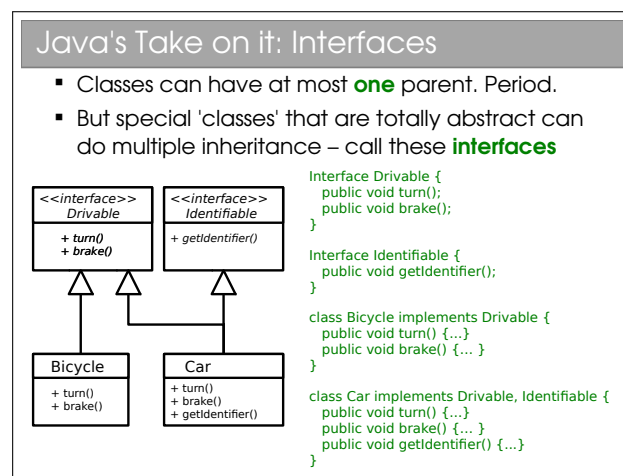
This is the obvious and (perhaps) sensible option that manages to capture the concept nicely.



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



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.



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*<sup>2</sup> 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.

<sup>2</sup>See workbook 5

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...

# Lecture 6

## Lifecycle of an Object

### Constructors

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

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.
- It's a method that gets called when the object is constructed, and it goes by the name of a **constructor** (it's not rocket science). 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	C++
<pre>public class Person {     private String mName;      // Constructor     public Person(String name) {         mName=name;     }      public static void main(         String[] args) {         Person p =             new Person("Bob");     } }</pre>	<pre>class Person {     private:         std::string mName;      public:         Person(std::string &amp;name){             mName=name;         } };  int main (int argc,           char ** argv) {     Person p ("Bob"); }</pre>

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 ref-

erence 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)

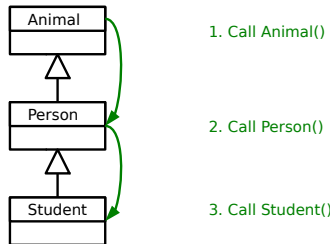
Again, not all languages support this. Python doesn't support multiple overloaded `__init__` methods, and this can be a bit frustrating,

**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++:

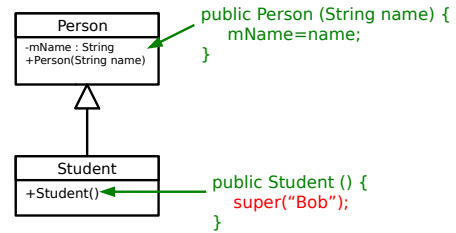
```

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:



## 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.

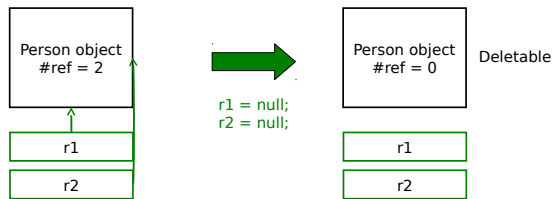
## Cleaning Up

- 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.

# Lecture 7

## Error Handling

One of the more difficult problems in programming is how and when to deal with things that go wrong.

### 7.1 Return Codes

#### Return Codes

- 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

Originally there was no explicit mechanism for error handling. The trick that was used was to return the error state via the normal return type. For this to work, the range of potential results from a procedure must be smaller than the range of the return type. For example, we might have a function for square root with the prototype `float sqrt(float a)`. The convention is to return the positive root, and so we can return `-1.0` to signify an error:

```
float sqrt(float a) {  
    if (a<0.0) return -1.0;  
    else {  
        ...  
    }  
}
```

If the return type isn't something we can repurpose (e.g. a custom class) then we can instead pass the output by reference and have the function return an integer to indicate the error state. E.g.,

```
SomeCustomClass sqrt(float a) {  
    return new SomeCustomClass(...);  
}
```

becomes

```
int func(float a, SomeCustomClass result ) {  
    if (a<0.0) return -1.0;  
    else result.set(...);  
    return 0;  
}
```

You might see functions that return `null` if they have an error. This is a very bad practice since it relies on the programmer using the function to check for `null`. If they don't, they'll likely try to dereference `null` and their program will die...

In fact, this is a larger problem with the general approach. We are dependent on the programmer testing the return value. Two problems arise: firstly, they could neglect to check (really common); secondly, they end up with really nasty looking code such as:

```
int retval = somefunc();  
  
if (retval==-1) {  
    // handle error type 1  
}  
else if (retval==-2) {  
    // handle error type 2  
}  
else if (retval==-3) {  
    // handle error type 3  
}
```

Here, just writing one line to call one function results in a screen-worth of error handling code. This constant mixing of code and error handling makes the code all but unreadable.

## 7.2 Deferred Error Handling

### Deferred Error Handling

- A similar idea (with the same issues) is to set some state in the system that needs to be checked for errors.
- C++ does this for streams:

```
ifstream file( "test.txt" );
if ( file.good() )
{
    cout << "An error occurred opening the file" << endl;
}
```

### Flow Control During Exceptions

- When an exception is thrown, any code left to run in the try block is skipped

```
double z=0.0;
boolean failed=false;
try {
    z = divide(5,0);
    z = 1.0;
}
catch(DivideByZeroException d) {
    failed=true;
}
z=3.0;
System.out.println(z+" "+failed);
```

## 7.3 Exceptions

### Exceptions

- 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
- Example usage:

```
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.

### Creating Exceptions

- Just extend Exception (or RuntimeException if you need it to be unchecked). Good form to add a detail message in the constructor but not required.

```
public class DivideByZero extends Exception {}

public class ComputationFailed extends Exception {
    public ComputationFailed(String msg) {
        super(msg);
    }
}
```

- You can also add more data to the exception class to provide more info on what happened (e.g. store the numerator and denominator of a failed division)

### Throwing Exceptions

- An exception is an object that has Exception as an ancestor
- So you need to create it (with new) before throwing

```
double divide(double x, double y) throws DivideByZeroException {
    if (y==0.0) throw new DivideByZeroException();
    else return x/y;
}
```

## Multiple Handlers

- A try block can result in a range of different exceptions. We test them in sequence

```
try {
    FileReader fr = new FileReader("somefile");
    int r = fr.read();
}
catch(FileNotFoundException fnf) {
    // handle file not found with FileReader
}
catch(IOException d) {
    // handle read() failed
}
```

Note: once a catch block is matched, the remaining catch blocks are skipped.

## Exception Hierarchies

- You can use inheritance hierarchies

```
public class MathException extends Exception {...}
public class InfiniteResult extends MathException {...}
public class DivByZero extends MathException {...}
```

- And catch parent classes

```
try {
    ...
}
catch(InfiniteResult ir) {
    // handle an infinite result
}
catch(MathException me) {
    // handle any MathException or DivByZero
}
```

## Checked vs Unchecked Exceptions

- **Checked:** must be handled or passed up.
  - Used for recoverable errors
  - Java requires you to declare checked exceptions that your method throws
  - Java requires you to catch the exception when you call the function

```
double somefunc() throws SomeException {}
```

- **Unchecked:** not expected to be handled. Used for programming errors
  - Extends RuntimeException
  - Good example is NullPointerException

There is an ongoing debate about the value of checked exceptions and they feature in some OOP languages but not others. Most of the time you'll be writing and dealing with checked exceptions in Java. You'll encounter unchecked exceptions only when you mess up in your code.

Aside: It turns out with Java they decided that `RuntimeException` should inherit from `Exception`. This means that if you ever write `catch(Exception e) {...}` then you will also catch the unchecked exceptions. So don't ever write that!

## finally

- With resources we often want to ensure that they are closed whatever happens

```
try {
    fr.read();
    fr.close();
}
catch(IOException ioe) {
    // read() failed but we must still close the FileReader
    fr.close();
}
```

## finally II

- The finally block is added and will *always* run (after any handler)

```
try {
    fr.read();
}
catch(IOException ioe) {
    // read() failed
}
finally {
    fr.close();
}
```

## Evil I: Exceptions for Flow Control

- At some level, throwing an exception is like a GOTO
- Tempting to exploit this

```
try {
    for (int i=0; ; i++) {
        System.out.println(myarray[i]);
    }
}
catch (ArrayOutOfBoundsException ae) {
    // This is expected
}
```

- This is not good. Exceptions are for exceptional circumstances only
  - Harder to read
  - May prevent optimisations

The code readability argument should be obvious but the second argument warrants more discussion. If you

Google the notion of flow control with exceptions, you will probably find many comments that suggest exception throwing is very slow compared to ‘normal’ code execution. This is attributed variously to the need to create an Exception object; the need to create a stack trace; or even just the need to create a message string. Some people report Exception handling was 50 times slower on the first JVMs!

Now, you *could* write a JVM that handled exception throwing efficiently, such that code like that in the slide would carry little performance penalty. But the crucial point is that there is no guarantee that a JVM will do so (and many still don’t). Exceptions are intended to be rare occurrences and it is perfectly reasonable (if not natural) for a JVM creator to assume this and therefore not need to worry about optimising exception handling. Bottom line: this smells bad.

### Evil II: Blank Handlers

- Checked exceptions must be handled
- Constantly having to use try...catch blocks to do this can be annoying and the temptation is to just gaffer-tape it for now

```
try {  
    FileReader fr = new FileReader(filename);  
}  
catch (FileNotFoundException fnf) {  
}
```

- ...but we never remember to fix it and we could easily be missing serious errors that manifest as bugs later on that are extremely hard to track down

This is a bad habit that novices tend to adopt—try not to develop it yourself. Eclipse at least discourages blank handlers, automatically filling in `e.printStackTrace()` so there’s some record of the problem printed to the screen. However, in large programs, where there’s often lots of debug output flowing to the console, these messages are easily missed... Better to fill in your handlers!

### Advantages of Exceptions

- 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**

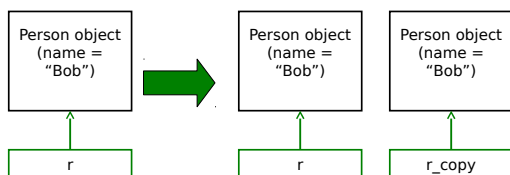
<http://java.sun.com/docs/books/tutorial/essential/exceptions/>

# Lecture 8

## Copying Objects

### Cloning I

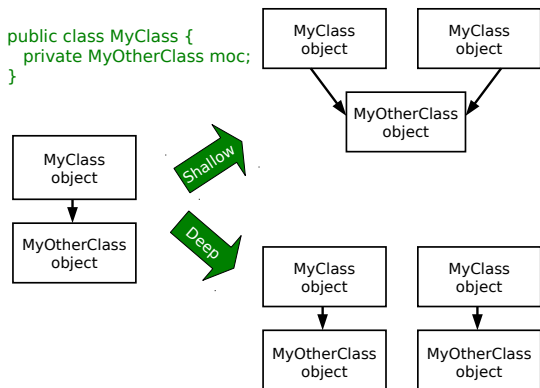
- Sometimes we really do want to copy an object



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

### Shallow and Deep Copies

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



### 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 implements 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.

### Cloning Arrays

- Arrays have built in cloning but the contents are only cloned *shallowly*

```
int intarray[] = new int[100];
Vector3D vecarray = new Vector3D[10];

...

int intarray2[] = intarray.clone();
Vector3D vecarray2 = vecarray.clone();
```

### Covariant Return Types

- The need to cast the clone return is annoying

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

- Recent versions of Java allow you to override a method in a subclass and change its return type to a subclass of the original's class

```
class A {}
class B extends A {}

class C {
    A mymethod() {}
}

class D extends C {
    B mymethod() {}
}
```

### 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).

The `clone()` approach is unique to Java. It can be a bit of a headache, but it was meant to address the shortcomings of the de-facto copying approach in OOP, which is the use of copy constructors:

### Copy Constructors

- Another way to create copies of objects is to define a **copy constructor** that takes in an object of the same type and manually copies the data
- See examples sheet

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

I won't go into detail on these here. Instead they are on the examples sheet.



# Lecture 9

## Collections

### 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:

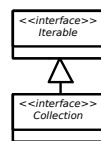
- Complex data structures and algorithms
- I/O (input/output: reading and writing files, etc)
- Networking
- Graphical interfaces

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.

## 9.1 Collections and Generics

### Java's Collections Framework



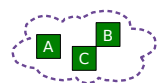
- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("iterate over it")
- The Collections framework has two main interfaces: *Iterable* and *Collection*. They define a set of operations that all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

### Sets

`<<interface>> Set`

- A collection of elements with no duplicates that represents the mathematical notion of a set
- `TreeSet`: objects stored in order
- `HashSet`: objects in unpredictable order but fast to operate on (see Algorithms course)

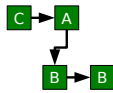


```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(15);
ts.add(12);
ts.contains(7); // false
ts.contains(12); // true
ts.first(); // 12 (sorted)
```

## Lists

### <<interface>> List

- An ordered collection of elements that may contain duplicates
- LinkedList: linked list of elements
- ArrayList: array of elements (efficient access)
- Vector: Legacy, as ArrayList but threadsafe



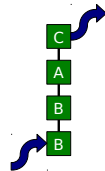
```

LinkedList<Double> ll = new LinkedList<Double>();
ll.add(1.0);
ll.add(0.5);
ll.add(3.7);
ll.add(0.5);
ll.get(1); // get element 2 (==3.7)
  
```

## Queues

### <<interface>> Queue

- An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
- offer() to add to the back and poll() to take from the front
- LinkedList: supports the necessary functionality
- PriorityQueue: adds a notion of priority to the queue so more important stuff bubbles to the top



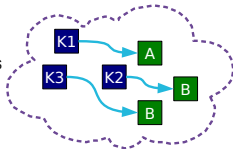
```

LinkedList<Double> ll = new LinkedList<Double>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
  
```

## Maps

### <<interface>> Map

- Like dictionaries in ML
- Maps **key** objects to **value** objects
- Keys must be unique
- Values can be duplicated and (sometimes) null.
- TreeMap: keys kept in order
- HashMap: Keys not in order, efficient (see Algorithms)



```

TreeMap<String, Integer> tm = new TreeMap<String,Integer>();
tm.put("A",1);
tm.put("B",2);
tm.get("A"); // returns 1
tm.get("C"); // returns null
tm.contains("G"); // false
  
```

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.

Now, don't worry about too much 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. You should get into the habit of reading the API descriptions to know which is the right choice for your problem.

## Iteration

### ▪ for loop

```

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

### ▪ foreach loop (Java 5.0+)

```

LinkedList list = new LinkedList();
...
for (Integer i : 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 = studentlist.get(i);
    for (int j=0; j<subjectlist.size(); j++) {
        Subject sub = 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<Integer> it = list.iterator();
while(it.hasNext()) {Integer i = it.next();}
for (; it.hasNext(); ) {Integer i = 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.

## The Origins of Generics

```
// 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

## The Origins of Generics 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)

This is pretty nasty. The OOP paradigm has let us write a flexible data structure that can handle us wrapping around various types, but it can't apply the restriction that all the types in one object should be the same. Additionally, all this casting makes for ugly code. This is what convinced the Java designers that parameterised types (Generics) were needed. But it was already a bit late: there was tons of established code using Collections (and still is). The Java designers were faced with the problem of updating the language to support parameterised types without breaking everything that went before.

## The Generics Solution

- Java implements *type erasure*
  - Compiler checks through your code to make sure you only used a single type with a given Generics object
  - Then it deletes all knowledge of the parameter, converting it to the old code invisibly

```
LinkedList<Integer> ll =
new LinkedList<Integer>();

...
for (Integer i : ll) {
    do_something(i);
}
```

→

```
LinkedList ll =
new LinkedList();

...
for (Object i : ll) {
    do_something((Integer)i);
}
```

So now we see why we can't use primitives as parameters: whatever we put there must be castable to Object, which primitives simply aren't.

## The C++ Templates Solution

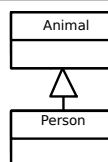
- Compiler first generates the class definitions from the template

```
class MyClass<T> {
    T membervar;
};

class MyClass_float {
    float membervar;
};
class MyClass_int {
    int membervar;
};
class MyClass_double {
    double membervar;
};
...
```

C++ doesn't suffer from the same problem since it just generates a special class for each instance you request.

## Generics and SubTyping



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

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

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

# Lecture 10

## 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.

### Value Equality

- Use the `equals()` method in `Object`
- Default implementation just uses reference equality (`==`) so we have to override the method

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

    @Override
    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 try to limit my use of `equals()` on objects from core Java classes, where I trust it to have been done properly.

## 10.1 Object Equality

### Reference Equality

- `r1==r2, r1!=r2`
- These test *reference equality*
- i.e. do the two references point at the same chunk of memory?  
`Person p1 = new Person("Bob");`  
`Person p2 = new Person("Bob");`

`(p1==p2);` ← False (references differ)

`(p1!=p2);` ← True (references differ)

`(p1==p1);` ← True

### Aside: Use The Override Annotation

- It's so easy to mistakenly write:

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

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

    public static void main(String args[]) {
        EqualsTest t1 = new EqualsTest();
        EqualsTest t2 = new EqualsTest();
        Object o1 = (Object) t1;
        Object o2 = (Object) t2;
        System.out.println(t1.equals(t2));
        System.out.println(o1.equals(o2));
    }
}
```

### Aside: Use The Override Annotation II

- Annotation would have picked up the mistake:

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

    @Override
    public boolean equals(EqualsTest e) {
        return (this.x==e.x);
    }

    public static void main(String args[]) {
        EqualsTest t1 = new EqualsTest();
        EqualsTest t2 = new EqualsTest();
        Object o1 = (Object) t1;
        Object o2 = (Object) t2;
        System.out.println(t1.equals(t2));
        System.out.println(o1.equals(o2));
    }
}
```

What's happening here is that the signature of our overriding method doesn't match the one in **Object**. So, Java actually *overloads* it, keeping both methods. By using **@Override** when we mean to override not overload, the compiler will spot our error.

For the geeks out there (i.e. non-examinable), we could write a compiler that spots that **EqualsTest** is a subclass of **Object** and therefore do overriding. This is called *covariant parameter types* and is *not* supported by Java. this matches back to where we introduced *covariant return types*, where Java spotted that the return type was a subclass and allowed it. So if we have:

```
public class A {
    Object void work(Object o) {...}
}
```

then this is not allowed (covariant parameter types):

```
public class B extends A {
    @Override
    public Object work(Person p) {...}
}
```

but this is (covariant return types):

```
public class C extends A {
    @Override
    public Person work(Object o) {...}
}
```

### Java Quirk: hashCode()

- Object also gives classes hashCode()
- Code assumes that if equals(a,b) returns true, then a.hashCode() is the same as b.hashCode()
- So you should override hashCode() at the same time as equals()

I don't want to go into this in too much detail since you haven't yet met hashes (it's in the Algorithms course next term). For now, just accept that a hash is a function that takes in chunks of information (e.g. all the fields in an object) and spits out a number. Java uses this in its **HashMap** implementation and other places as a shortcut to having to sequentially compare each field. I mention it here really for completeness so that if any of you override **equals()** in production code then you know you should also override **hashCode()**. Details of doing so are easily found on the web and in books (because it's a very common mistake to make!).

## 10.2 Less Than and Greater Than

In order to sort your classes using the built in classes, you need to write something that allows two objects to be ordered. Often our classes have a *natural ordering* e.g. people are usually sorted first by surname and then by forename. We can build-in natural ordering to our classes using the **Comparable** interface:

### 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

## 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>();
```

This is all very well, but sometimes we might want to sort with a different ordering (e.g. sort just by fore-name). Java Collections lets us do this by supplying a custom piece of code for the ordering: a *Comparator*:

## Comparator<T> Interface I

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

- Also part of the Collections framework and allows us to specify a specific ordering for a particular job
- E.g. a Person might have natural ordering that sorts by surname. A Comparator could be written to sort by age instead...

## Comparator<T> Interface II

```
public class Person implements Comparable<Person> {
    private String mSurname;
    private int mAge;
    public int compareTo(Person p) {
        return mSurname.compareTo(p.mSurname);
    }
}

public class AgeComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return (p1.mAge-p2.mAge);
    }
}

...
ArrayList<Person> plist = ...;
...
Collections.sort(plist); // sorts by surname
Collections.sort(plist, new AgeComparator()); // sorts by age
```

Note that a natural ordering uses `compareTo()` whilst a comparator uses `compare()`.

## 10.3 Operator Overloading

### Operator Overloading

- Some languages have a neat feature that allows you to overload the comparison operators. e.g. in C++

```
class Person {
public:
    int mAge;
    bool operator==(Person &p) {
        return (p.mAge==mAge);
    };
}

Person a, b;
b == a; // Test value equality
```

Java doesn't have this, but it's good to know what it is at this stage.

# Lecture 11

## Design Patterns

### 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.

### 11.1 So Design Patterns are like coding recipes?

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

### 11.2 Why Bother Studying Them?

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

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

The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments<sup>1</sup> 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.

<sup>1</sup>You are commenting your code liberally, aren't you?

## 11.3 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.



## 11.4 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?

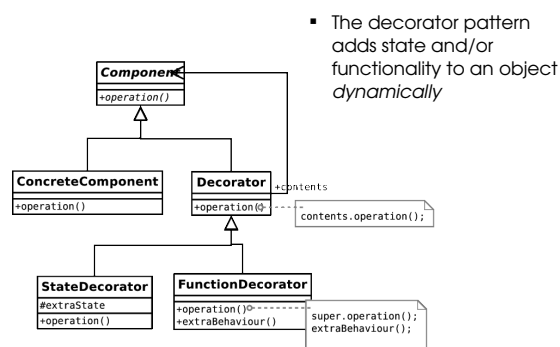
**Solution 1:** Add variables to the established Book class that describe whether or not the product is to be gift wrapped.

**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.

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`.

### Decorator in General



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

## 11.5 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...).

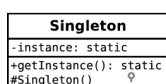
Instead we exploit our access modifiers and create a **private** constructor (to ensure no-one can create objects at will) and add in a **static** member (the only instance we will ever have). Finally, we include a static getter for this member.

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

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?

### Singleton in General



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

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

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

## 11.6 The State Pattern

### State

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

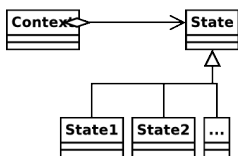
**Example problem:** Representing academics as they progress through the rank

**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

## 11.7 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.

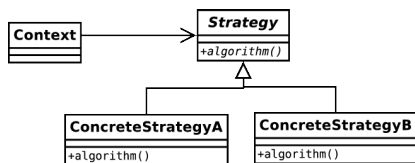
**Solution 2:** (Strategy) Create an abstract `ChangeFinder` class. Derive a new class for each of our algorithms.

- Strategy is about encapsulating behaviour in a class. This behaviour does not depend on internal variables.
- Different concrete Strategies 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 `set-Strategy(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.

### 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 `set-State(State s)` function.

## 11.8 The Composite Pattern

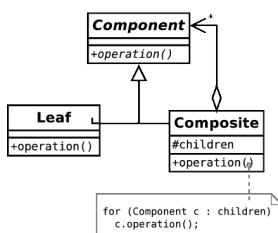
### Composite

**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

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.

## 11.9 The Observer Pattern

### Observer

**Abstract problem:** When an object changes state, how can any interested parties know?

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

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);
```

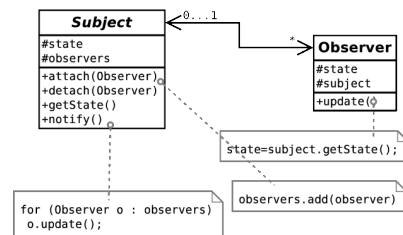
We then register with it with a line like:

```
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.

### Observer in General

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



## 11.10 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).

## 11.11 Other Patterns

You've now met a few Design Patterns. There are plenty more (23 in the original book and many, many more identified since), 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 *refac-*

*toring* your code as new situations arise. However, if a Design Pattern *is* appropriate, you should probably use it.

## 11.12 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!].

# Appendix I: Java, the JVM and Bytecode

Java is known for its cross-platform abilities, which has given it strong internet credentials. Being able to send a file compiled on one machine to another machine with a different architecture and have it run is a neat trick. It shouldn't work because the machine code for one machine shouldn't make sense to another.

## Interpreter to Virtual Machine

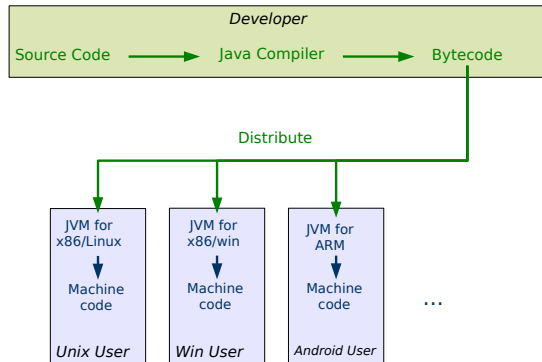
- Java was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?
- Could use an interpreter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

So the trick is to *partially* compile the Java code to a machine code for a universal machine (that doesn't actually exist). To actually *use* this special machine code ("bytecode") a machine must translate from bytecode to its own local machine code. To that it must have a Java Virtual Machine (JVM) installed that knows the translation.

## Java Bytecode II



## 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