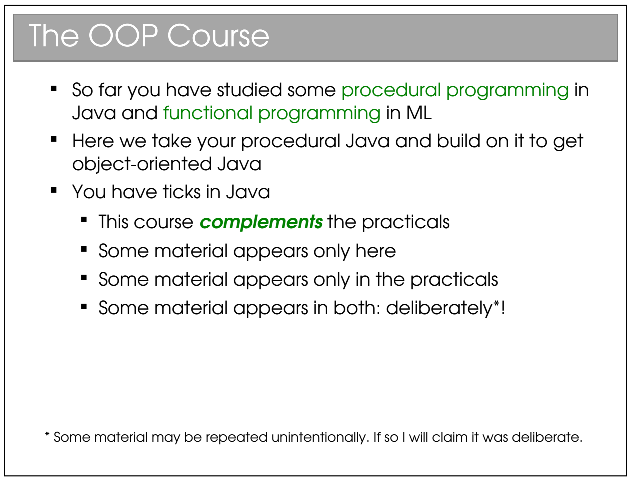
Object Oriented Programming
Dr Robert Harle

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

Michaelmas 2015/16

The Course


The OOP Course

- So far you have studied some procedural programming in Java and functional programming in ML
- Here we take your procedural Java and build on it to get object-oriented Java
- You have ticks 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*!

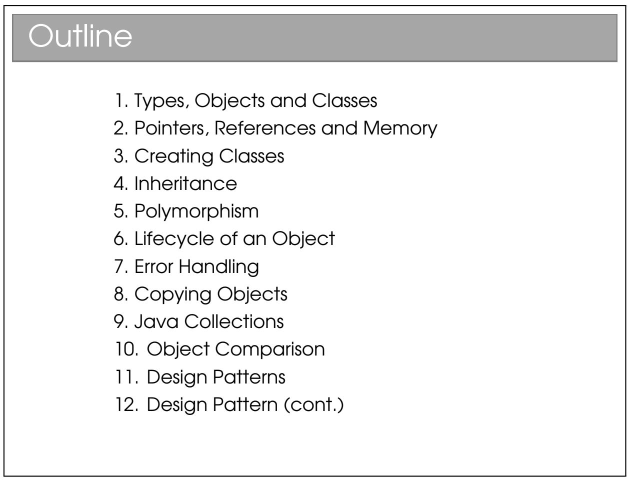
* Some material may be repeated unintentionally. If so I will claim it was deliberate.

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 since it maps directly to the functions you are so familiar with from maths. 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 real-world experience of programming, you're probably aware that functional programming is 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[1] and, crucially, not all imperative languages support all of the same language concepts in the same way. So, if you just learn one language (e.g. Java) you'll probably struggle to separate the underlying programming concepts from the Java-specific quirks and jumping ship (to, say, C++) can prove a bit challenging.

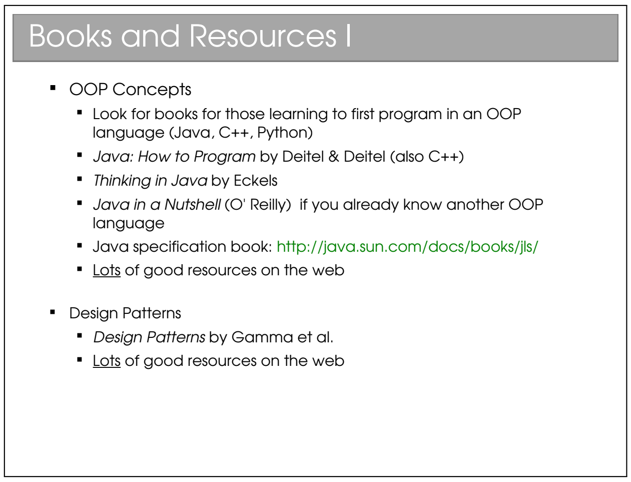
This is quite similar to learning 'natural' languages (i.e. the languages we write and speak in). We are all proficient, if not expert, in our mother tongue. Oddly we are very good at spotting text that breaks the 'rules' of that language (grammar, spelling, etc.), but almost hopeless at identifying the rules themselves. Becoming fluent in a new language forces you to break language down into its constituent rules, and you become better at your original language! Those that are multilingual often comment that once you know two languages well, picking up more is quite trivial: it's just a case of figuring out which rules to apply, possibly adding a few new rules, and learning the vocabulary. So it is with programming: once you've gone through the effort of learning a couple of languages picking up new ones is easy.

Now, the 'examinable' imperative language for Paper 1 is Java, and you won't be *required* to program in anything else. However, Java doesn't support all the interesting concepts found in imperative programming (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 non-examinable). We may also find time to try out Python and some other popular languages.


Outline

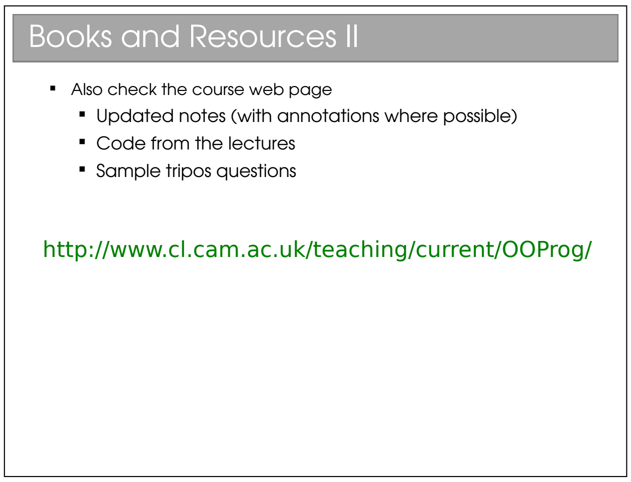
1. Types, Objects and Classes
2. Pointers, References and Memory
3. Creating Classes
4. Inheritance
5. Polymorphism
6. Lifecycle of an Object
7. Error Handling
8. Copying Objects
9. Java Collections
10. Object Comparison
11. Design Patterns
12. Design Pattern (cont.)

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



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 are many books and websites describing the basics of OOP. The concepts themselves are quite abstract, although most texts will use a specific language to demonstrate them. The books I've listed favour Java but you shouldn't see that as a dis-recommendation for other books. In terms of websites, Oracle 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.

0.1 Ticks

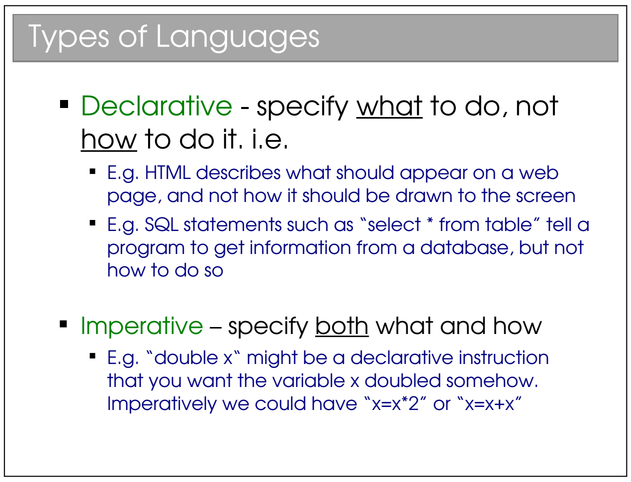
There are five OOP ticks, all in Java. They follow on from the work you did in module three of the Pre-arrival course, using the concepts from lectures to build ever-better Game of Life implementations.

If you cannot meet a tick deadline you should speak to your Director of Studies (DoS) in the first instance. If they agree there is a legitimate reason for late submission or a missed ticking slot, **they** can email me to request an extension or new slot. Any such action will always require the explicit support of the DoS.

Lecture 1

Types, Objects and Classes

1.1 Imperative, Procedural, Object Oriented


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"

Firstly a recap on declarative vs imperative:

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;

There's a nice meme[1] that helps here:

> Functional programming is like describing your problem to a mathematician. Imperative programming is like giving instructions to an idiot.

[1] attributed to arcus, #scheme on Freenode

Those of you who have done the Databases course will have encountered SQL. Even those that haven't can probably decipher the language to a point—here's a trivial example:

```
select * from person_table where name="Bob";
```

This gets all entries from the database person_table where the name column contains "Bob". This language is highly functional: I have specified what I want to achieve but given no indication as to *how* it should be done. The point is that the SQL language simply doesn't have any way to specify how to look something up—that functionality is built into the underlying database and we(as users) shouldn't concern ourselves.

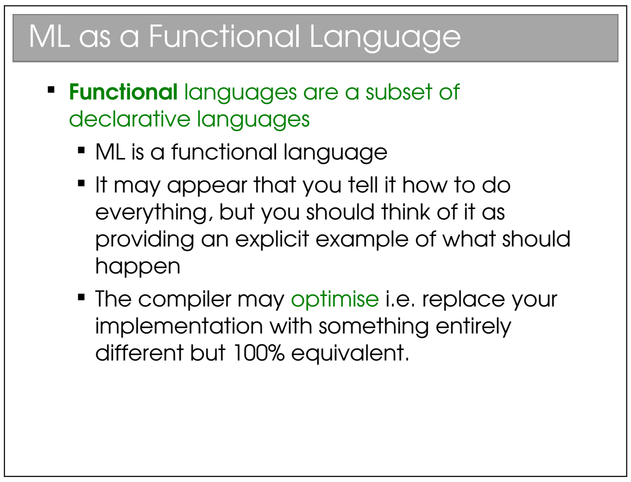
On the other hand, the machine and assembly code you saw in the pre-arrival course are arguably the ultimate imperative languages. However, as you can imagine, programming directly in them isn't very nice. Other imperative languages have evolved from these, each attempting to make the coding process more human-friendly.

A key innovation was the use of procedures (equate these to functions for now) to form *procedural* programming. The idea is to group statements together into procedures/functions that can be called to manipulate the state. Code ends up as a mass of functions plus some logic that calls them in an appropriate way to achieve the desired result. That's exactly how you used Java in the pre-arrival course.

OOP is an extension to procedural programming (so still imperative) where we recognise that these procedures/functions can themselves be usefully grouped (e.g. all procedures that update a PackedLong vs all procedures that draw to screen) *and* furthermore that it often makes sense to group the state they affect with them (e.g. group the PackedLong methods with the underlying long).

Wait...

You might be struggling with the fact you specified functions in ML that appear to fit the imperative mould: there were statements expressing *how* to do something. Think of these as specifying the desired result by giving an *example* of how it might be obtained. Exactly what the compiler does may or may not be the same as the example—so long as it gives the same outputs for all inputs, it doesn't matter.



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[2]. So the boundaries are blurred, but ML is predominantly functional and Java predominantly imperative.

1.1.1 Java as a Procedural Language

We used Java to introduce you to general procedural programming in the pre-arrival course. This is not ideal since Java is designed as an object oriented language first. Trying to force it to act entirely procedurally required a number of ugly hacks:

- all the functions had to be static (we'll explain that shortly);
- we had to create placeholder classes (public class TinyLife...); and.
- we had a lot of annoying 'boilerplate' code that seemed rather unnecessary (e.g. public static void main(...

Over the next few lectures the reason why these irritations are necessary should become apparent. Ideally, we would have used a purely procedural language that didn't need such hacks, but we didn't want to force you to setup and learn another language in one year.

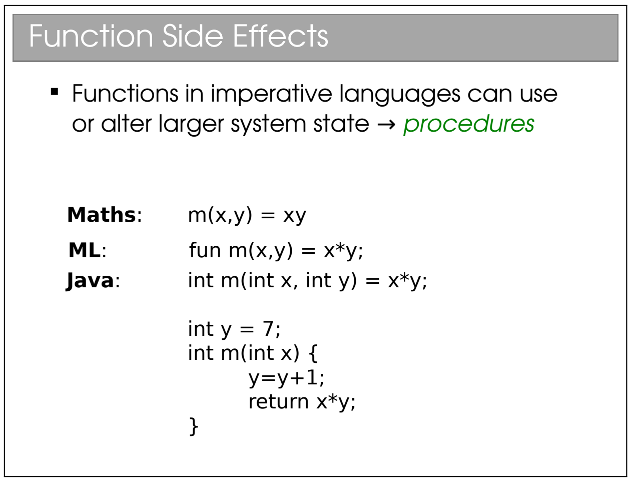
1.2 Procedures, Functions, Methods etc

Up to now, we've treated procedures as the same as functions. Herein it's useful for us to start making some distinctions. One of the key properties of functional languages is that they use *proper functions*. By this I mean functions that have the same properties as those you find in maths:

- ll functions return a (non-void) result;
- the result is *only* dependent on the inputs (arguments); and
- no state outside of the function can be modified (i.e. no "side effects").

Restricting ourselves to proper functions makes it easier for the compiler to assert declarative-like optimisations: a given function is completely standalone (i.e. dependent on no state). You can pluck out an arbitrary function without reference to anything else in the program and optimise it as you see fit.

[2]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..!



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

Procedures have similarities to (proper) functions but permit side effects and hence break the three rules given above. Here's another example:

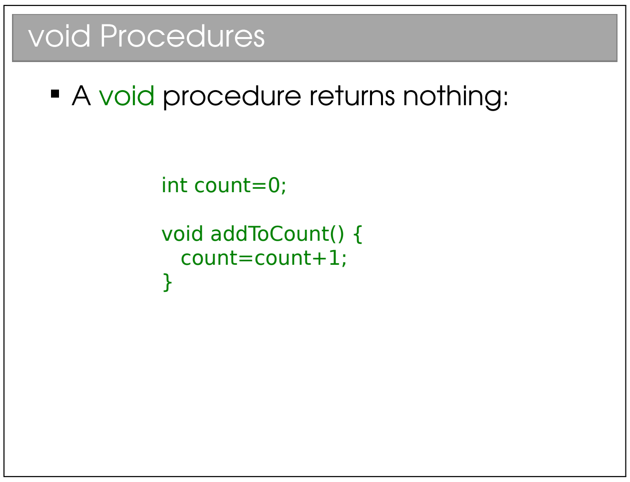
```
fun add(x,y)=x+y;
add(1,2);
```

This ML (proper) function will always return 3 regardless of the statements before or after it. In Java, we could write:

```
static int z=0;  // this is some global state
static 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
```

Eeek! Three calls with the same arguments gives three different answers. You certainly don't get that in maths! The problem is that the output is dependent on some other state in the system (z), which it changes (a *side effect* of calling it). 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.

To really hammer this home, you can now have useful functions with no arguments that return nothing (void)—both rather useless in maths:



void Procedures

- A void procedure returns nothing:

```
int count=0;

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

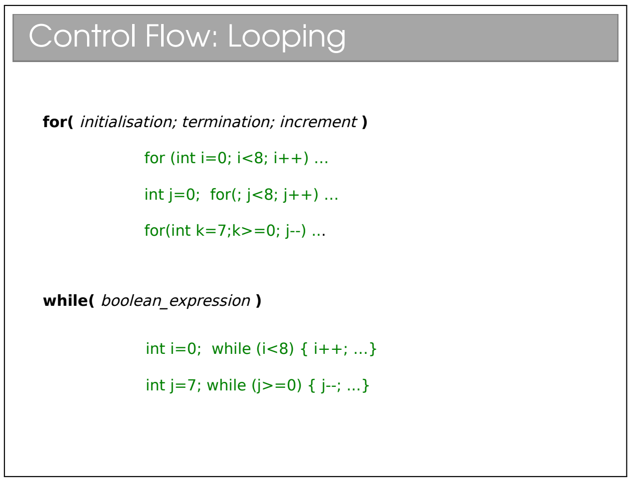
Health warning: Many imperative programmers use the word 'function' as a synonym for 'procedure'. Even in these lectures I will use 'function' loosely. You will have to use your intelligence when you hear the words.

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

1.3 Recap: Control Flow

You've covered control flow in the pre-arrival course but it seems wrong not to at least mention it here (albeit the coverage in lectures will be very brief). The associated statements are classified into *decision-making*, *looping* and *branching*.

Decision-making is quite simple: `if (...) {...} else {...}` is the main thing we care about. Looping doesn't require recursion (yay!) since we have `for` and `while`:



Control Flow: Looping

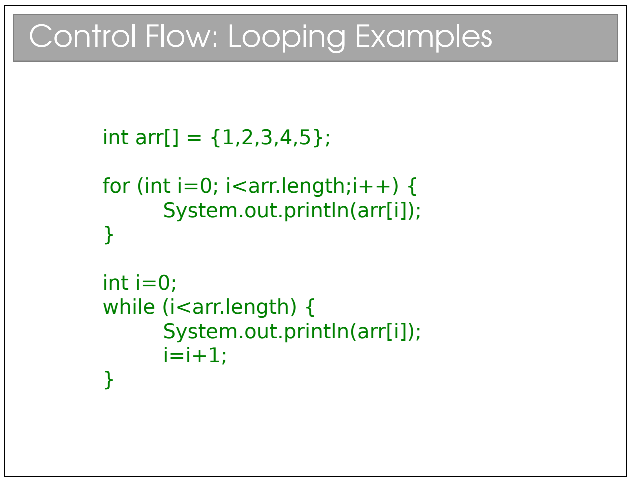
for(*initialisation; termination; increment* **)**

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

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

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

while(*boolean_expression* **)**

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

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

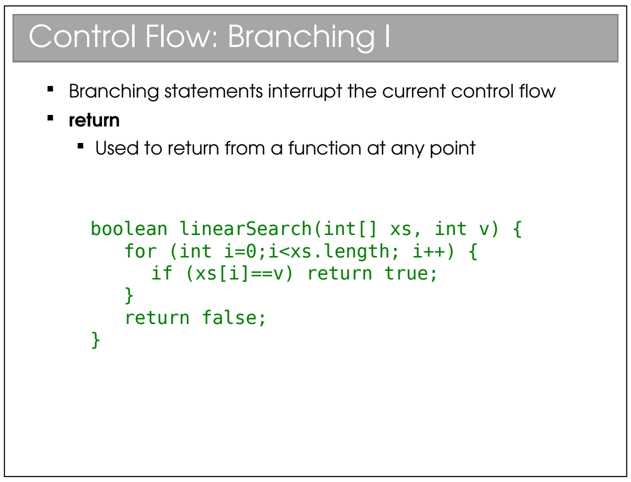
These examples all loop eight times. The following code loops over the entirety of an array (the `for` approach is more usual for this task!):



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;
}
```

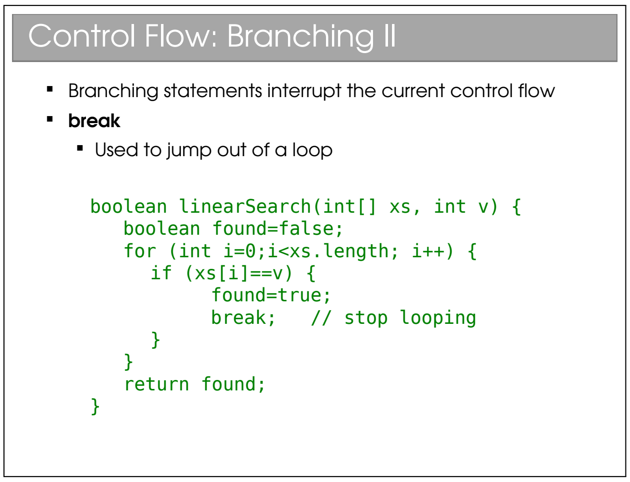
For branching, we mainly care about return, break and continue:



Control Flow: Branching I

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

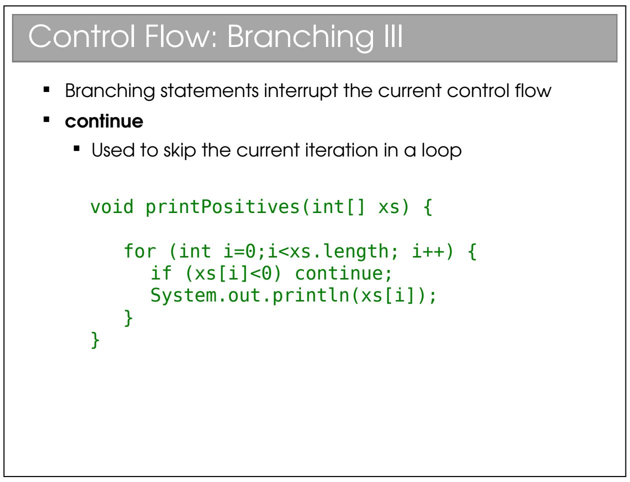
```
boolean linearSearch(int[] xs, int v) {
   for (int i=0;i<xs.length; i++) {
     if (xs[i]==v) return true;
   }
   return false;
}
```



Control Flow: Branching II

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

```
boolean 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;
}
```



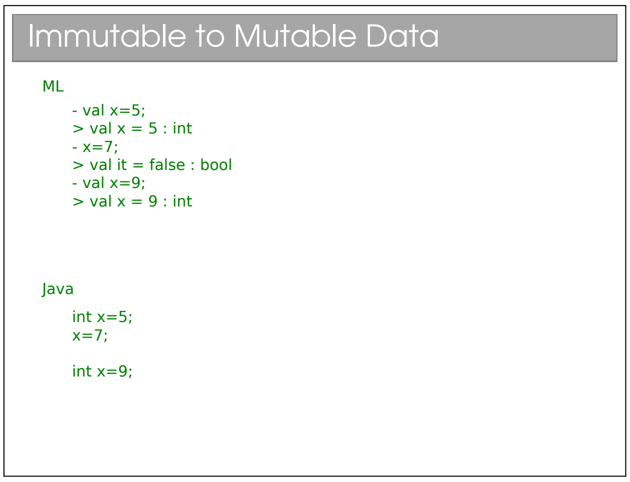
Control Flow: Branching III

- 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]);
   }
}
```

1.4 Values, Variables and Types

1.4.1 State Mutability



Immutable to Mutable Data

ML

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

Java

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

int x=9;
```

In ML you had *values* and in Java you have *variables*.

A simple way to ensure that functions in ML do not depend on external state is to make all state constant, or *immutable*. In ML you could write:

```
val x = 5;
x=7;
val x=9;
```

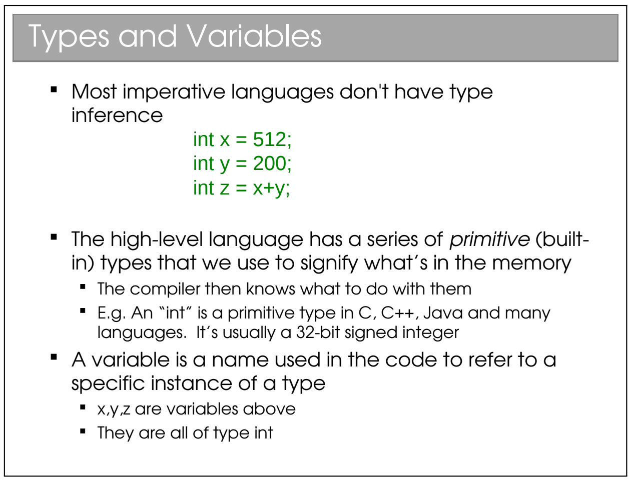
But these almost certainly didn't do what you expected the first time you tried them. The first line (val x = 7) creates a chunk of memory, sets the contents to 7 and associates it with a label x. The second (x = 5) you probably wanted to reassign the value, but—since the values are constant—it actually performed a comparison of x and 5, giving false! The third line val x=5 actually creates another value in memory, sets the value to 5 and reassigns the *label* x to point to it. The original value of 7 remains untouched in memory: you just can't update it. So now you should be able to understand the behaviour of:

```
val x = 7;
fun f(a)=x*a;
f(3);
val x=5;
f(3);
```

Java doesn't have the shackles of proper functions so we can have variables that can be updated (i.e. *mutable* state)–this is actually the essence of imperative programming:

```
int x = 7;  // create x and init to 7
x=5;        // change the value of x
x==5;       // compare the value of x
int x = 9;  // error: x already created
```

1.4.2 Explicit Types vs Type Inference



Types and Variables

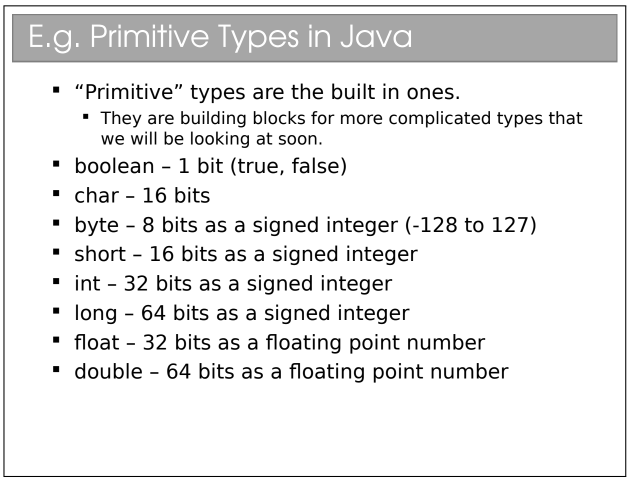
- Most imperative languages don't have type inference
  - int x = 512;
  - int y = 200;
  - int z = x+y;
- The high-level language has a series of *primitive* (built-in) types that we use to signify what's in the memory
  - The compiler then knows what to do with them
  - E.g. An "int" is a primitive 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 types (`real`, `int`, etc). You were (correctly) taught to avoid explicitly assigning types wherever possible: ML had the capability to infer types. This was particularly useful because it allowed polymorphism to avoid writing separate functions for integers, reals, etc. Every now and then you had to help it out and manually specify a type, but ML's *type inference* is essentially a really nice feature to have "baked in". (I acknowledge that ML's error messages about type errors could be a little less... cryptic).

Type inference isn't unique to ML or functional languages, although it's quite rare in imperative languages (Python and Javascript spring to mind). Java doesn't have it and 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.[3]

You have already met the primitive (built-in) types in the pre-arrival course, but here's a recap[4]



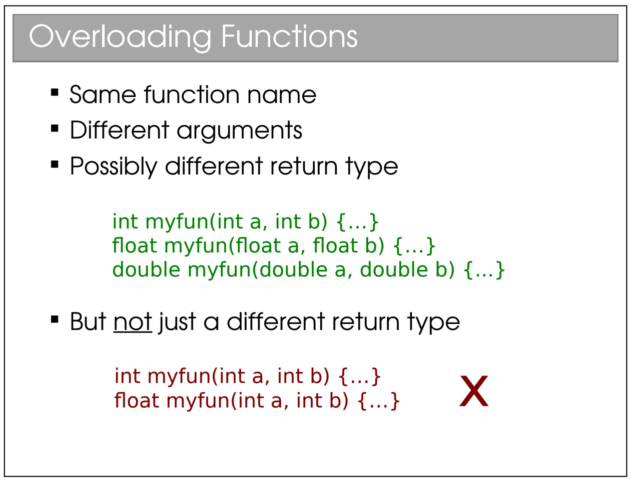
E.g. Primitive Types in Java

- "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

[3]Later 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.

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

1.4.3 Polymorphism vs Overloading



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) {…}
```

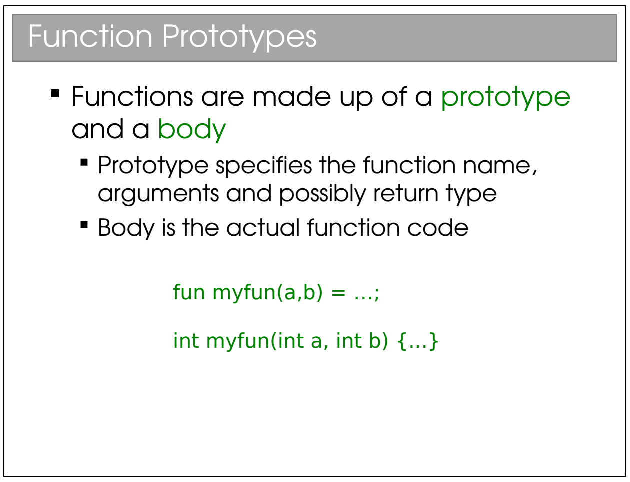
- But not just a different return type

```
int myfun(int a, int b) {…}
float myfun(int a, int b) {…}        X
```

Since Java demands that all types are explicit (disregarding Generics, which we'll come to soon), we rather lose the ability to write one function that can be applied to multiple types—the cool polymorphism you saw in ML. Instead we can make use of procedure *overloading*. This allows you to write multiple functions with the same name but different argument types:

```
int myfun(int a,int b,int c) {
   // blah blah blah
}

int myfun(double a, double b, double c) {
   // blah blah blah
}
```

When you call `myfun` the compiler looks at the argument types and picks the function that best matches. This is nowhere near as elegant as what we had in ML (I have to write out a separate function for every argument set) but at least when it's being used there is naming consistency. In passing, we note that we talk about function *prototypes* or *signatures* to mean the combination of return type, function name and argument set—i.e. the first line such as `int myfun(double a, double b, double c)`.



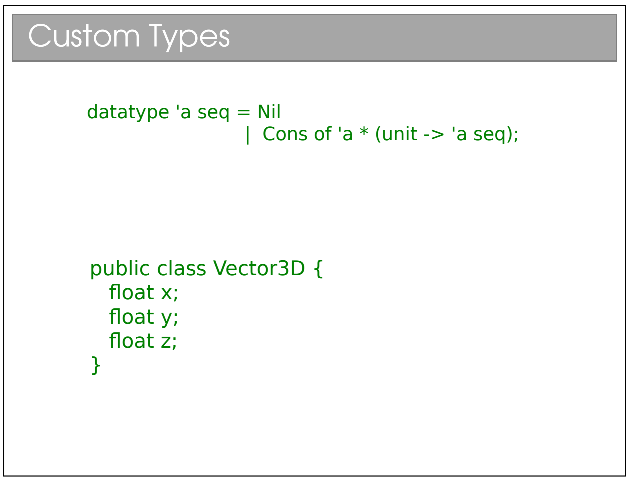
Function Prototypes

- 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) {…}
```

1.5 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;
}
```

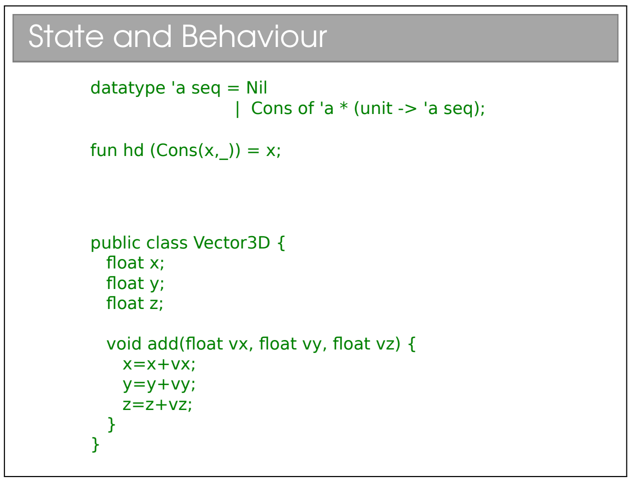
Let's take a simple example: representing 3D vectors (x,y,z). We could keep independent variables in our code. e.g.

```
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. 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*. But it goes further than just being a grouping of variables...

1.5.1 State *and* Behaviour

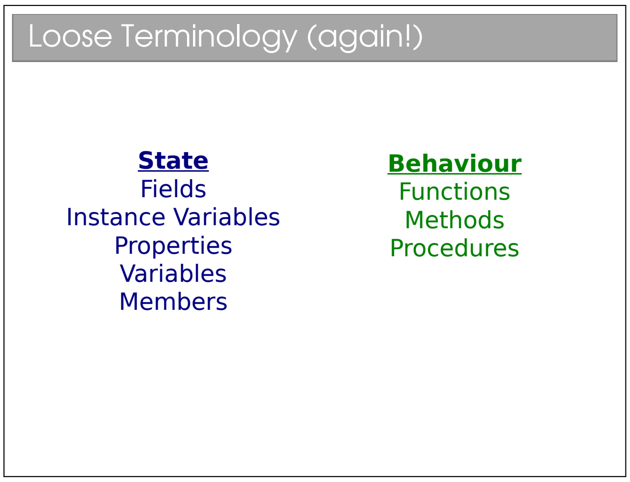


State and Behaviour

```
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. 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.

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).



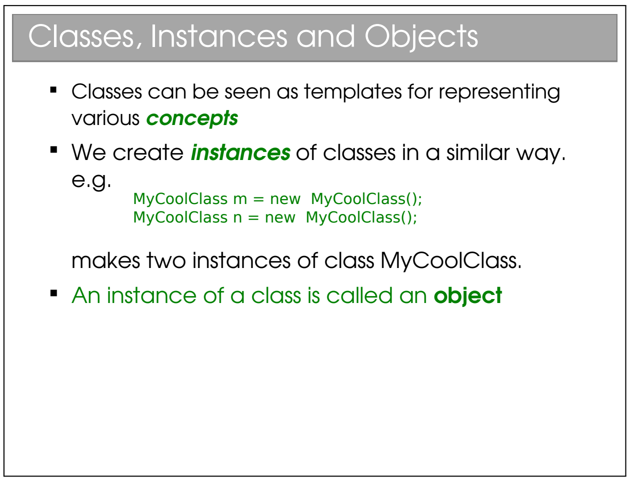
Loose Terminology (again!)

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

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

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

1.5.2 Instantiating classes: 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**

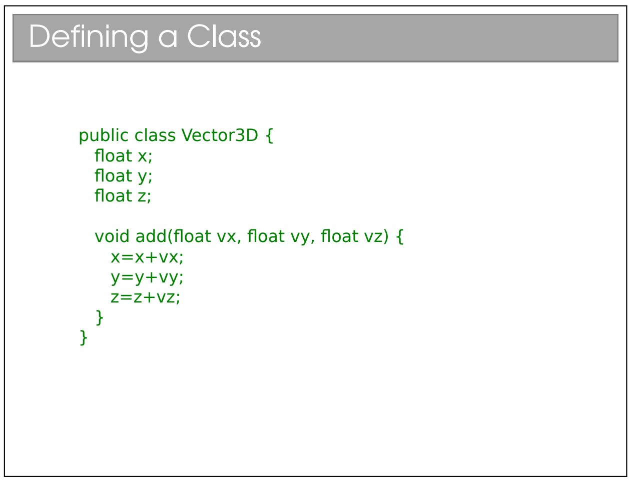
So a class is a grouping of state (data/variables) and behaviour (methods). Whenever we create an *instance* of a class, we call it an *object*. The difference between a class and an object is thus very simple, but you'd be surprised how much confusion it can cause for novice programmers. *Classes* define what properties and procedures every object of the type should have (a template if you like), while 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").

We've already seen how to create (define) objects in the last lesson of the pre-arrival course. There we had:

```
// Define p as a new Vector3 object
Vector3 p = new Vector3();
// Reassign p to a new Vector3 object
p = new Vector3()
```

The things to note are that we needed a special new keyword to instantiate an object; and that we pass it what looks to be a method call (Vector3()). Indeed, it *is* a method, but a special one: it's called a *constructor* for the class.

1.5.3 Defining Classes



Defining a Class

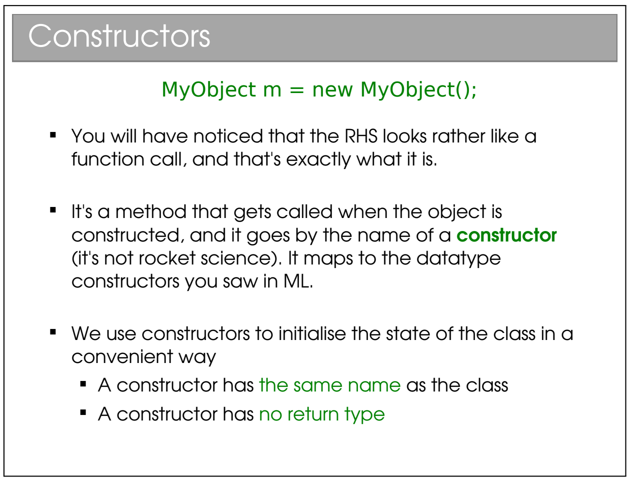
```
public class Vector3D {
  float x;
  float y;
  float z;

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

To define a class we need to declare the state and define the methods it is to contain. Here's a very simple Java class containing one integer as its state and a single method:

```
class MyShinyClass {
   int x;

   void setX(int xinput) {
      x=xinput;
   }
}
```

You were defining classes like this in your Pre-arrival course code. Except there it was peppered with the words public and static—we'll look at both shortly.

1.5.4 Constructors



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

In Java, C++ and most other OOP languages, constructors have two properties:

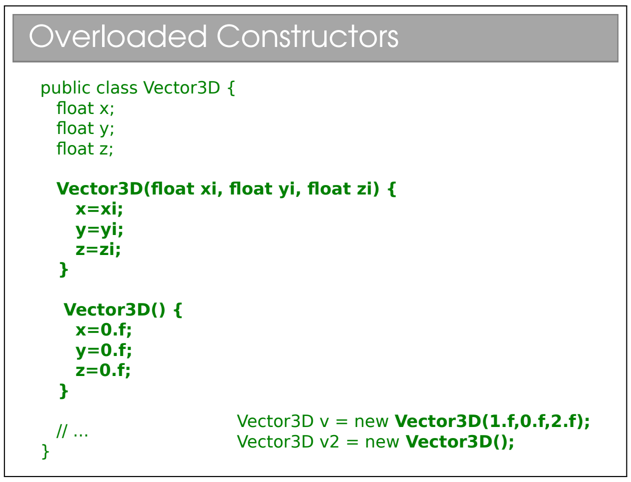
1. they have the same name as the class; and
2. they have *no* return type.

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:

```
class MyShinyClass {
   int x;

   MyShinyClass(int x_init) {
      setX(x_init);
   }

   void setX(int xinput) {
      x=xinput;
   }
}
```

Here, the constructor is used to initialise the member variable x to a value passed in. We would specify this when using the new keyword to create an object. e.g.

```
   MyShinyClass m = new MyShinyClass(42);
```



Overloaded Constructors

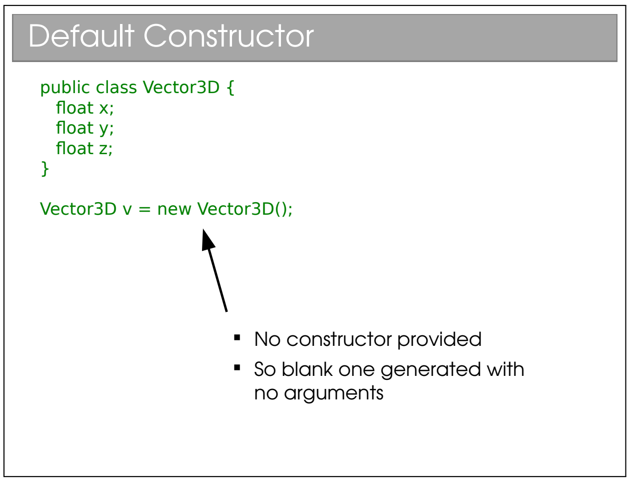
```
public class Vector3D {
  float x;
  float y;
  float z;

  Vector3D(float xi, float yi, float zi) {
     x=xi;
     y=yi;
     z=zi;
  }

  Vector3D() {
     x=0.f;
     y=0.f;
     z=0.f;
  }

  // ...
}
```

```
Vector3D v = new Vector3D(1.f,0.f,2.f);
Vector3D v2 = new Vector3D();
```

You can have multiple constructors by overloading the method:

```
class MyShinyClass {
   int x;

   MyShinyClass() {
      set
   }

   MyShinyClass(int x_init) {
      setX(x_init);
   }

   void setX(int xinput) {
      x=xInput;
   }
}

//...

// An object with x set to 42
MyShinyClass m = new MyShinyClass(42);

// And object with x set to 0
MyShinyClass m2 = new MyShinyClass();
```

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



Default Constructor

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

Vector3D v = new Vector3D();
```

- No constructor provided
- So blank one generated with no arguments

If you don't specify any constructor at all, Java fills in a *default constructor* for you. This takes no arguments and does nothing, other than allowing you to make objects. i.e.

```
class MyShinyClass {

}
```

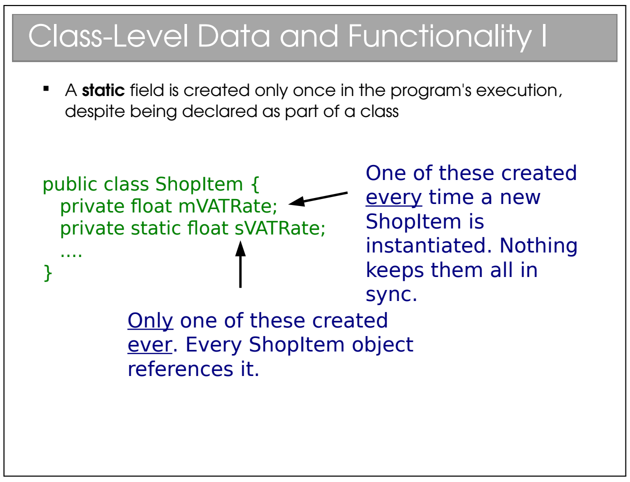
is converted to

```
class MyShinyClass {
   MyShinyClass() { }
}
```

so you can write MyShinyClass m = new MyShinyClass();.

1.5.5 Static

Sometimes there is state that is more logically associated with a class than with an object. An example will help here:



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

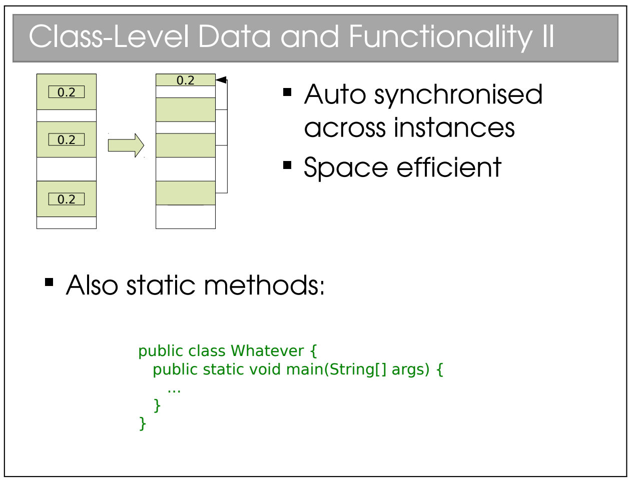
```
public class ShopItem {
  private float mVATRate;
  private static float sVATRate;
  ....
}
```

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.

So a static variable is only instantiated once per *class* not per *object*. Therefore you don't even need to create an object to access a static variable. 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. The same goes for methods: you write Math.sqrt(...) rather than having to first instantiate a Math object.



Class-Level Data and Functionality II

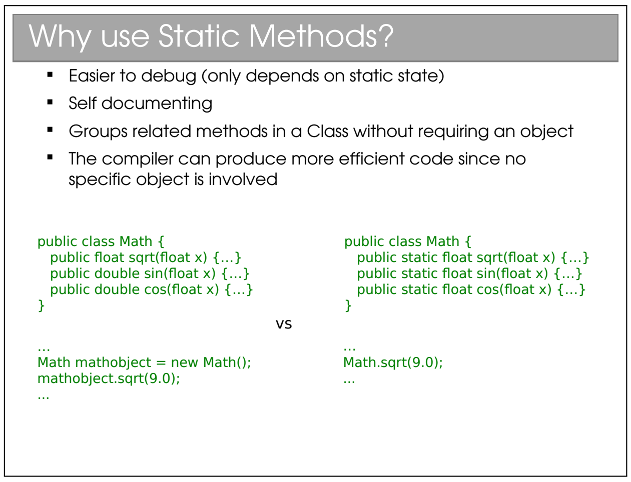
- Auto synchronised across instances
- Space efficient

- Also static methods:

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

Methods can also be static. In this case they must not use 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).

Looking back at the pre-arrival course, we really wanted something that had no class notion at all. The closest we could get in Java was to make everything static so there weren't any objects floating around.



Why use Static Methods?

- Easier to debug (only depends on static state)
- Self documenting
- Groups related methods in a Class without requiring an object
- The compiler can produce more efficient code since no specific object is involved

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

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

vs

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

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

Lecture 2

Designing Classes

2.1 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

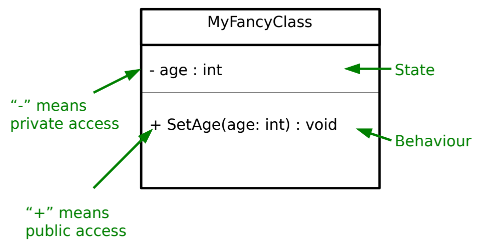
- We want our class to be a grouping of conceptually-related state and behaviour
- One popular way to group is using grammar
  - Noun   Object
  - Verb   Method

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

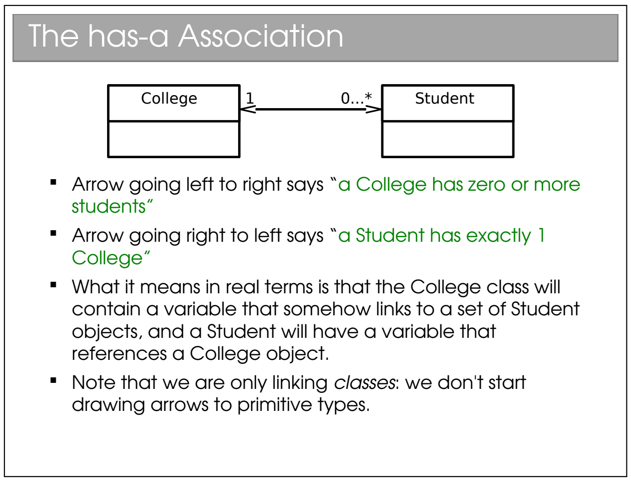
Very often classes follow naturally from the problem domain. So, if you are making a snooker game, you might have an object to represent the table; to represent each ball; to represent the cue; etc. Identifying the best possible set of classes for your program is more of an art than a science and depends 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



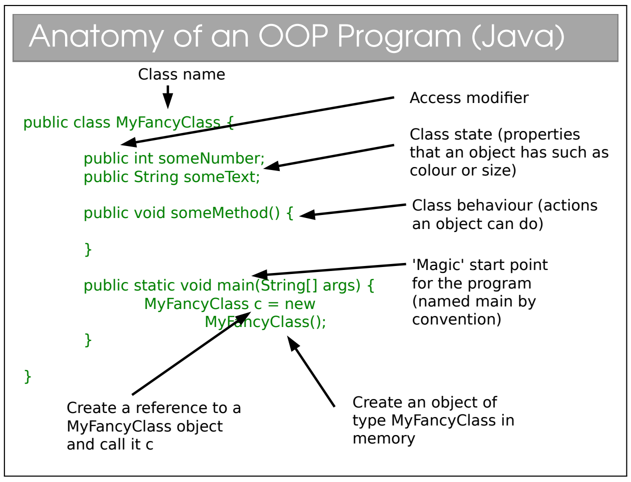
The graphical notation used here is part of UML (Unified Modeling Language). UML is a standardised set of diagrams that can be used to describe software independently of any programming language used to implement it. UML contains many different diagrams. In this course we will only use the *UML class diagram* such as the one in the slide.


The has-a Association

College
1
0...*
Student

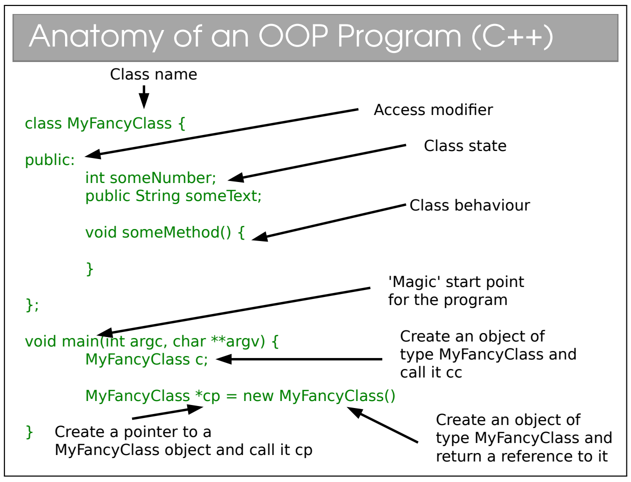
- Arrow going left to right says "a College has zero or more students"
- Arrow going right to left says "a Student has exactly 1 College"
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable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

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


Anatomy of an OOP Program (Java)

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

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

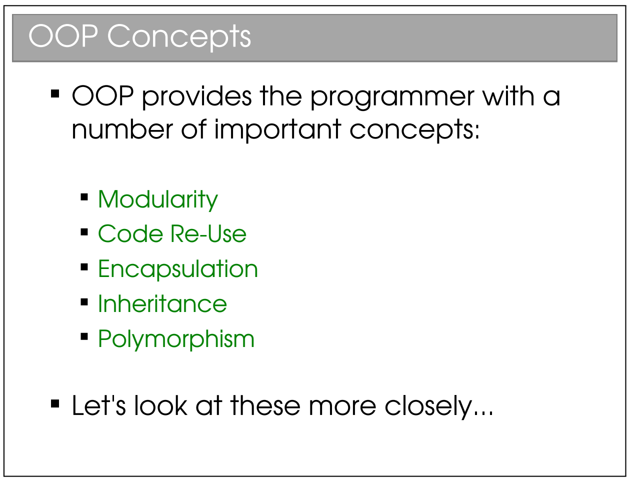
Class name
Access modifier
Class state (properties that an object has such as colour or size)
Class behaviour (actions an object can do)
'Magic' start point for the program (named main by convention)
Create a reference to a MyFancyClass object and call it c
Create an object of type MyFancyClass in memory


Anatomy of an OOP Program (C++)

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

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

        MyFancyClass *cp = new MyFancyClass()

}
```

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

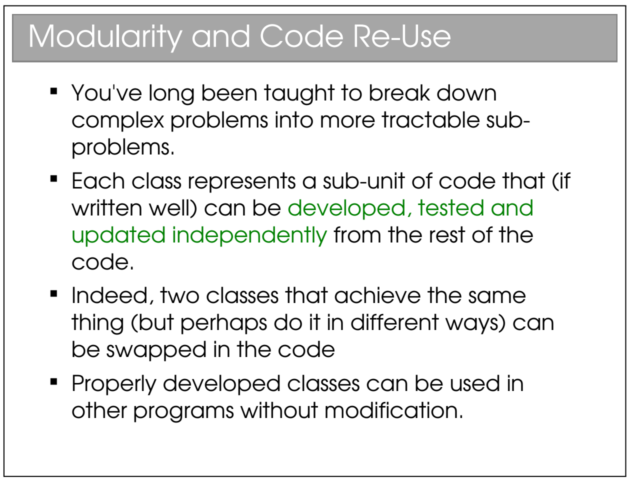
2.2 OOP Concepts


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.

2.2.1 Modularity and Code Re-Use


Modularity and Code Re-Use

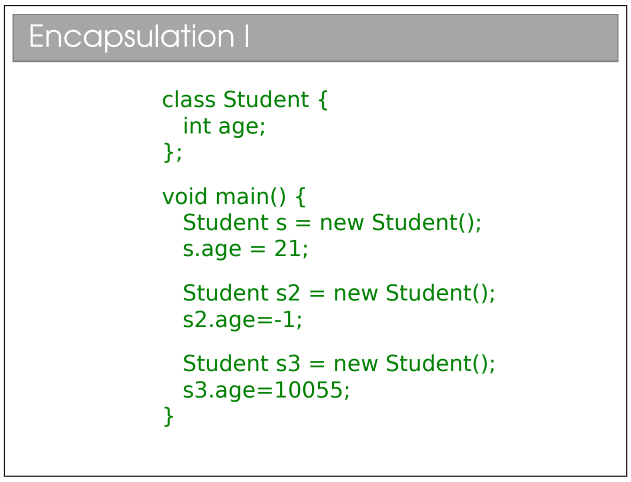
- You've long been taught to break down complex problems into more tractable sub-problems.
- Each class represents a sub-unit of code that (if written well) can be developed, tested and updated independently from the rest of the code.
- Indeed, two classes that achieve the same thing (but perhaps do 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 hem 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.

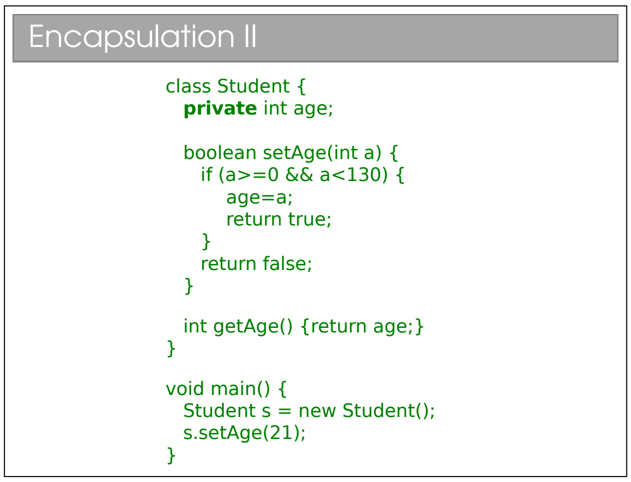
2.2.2 Encapsulation and Information Hiding



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.



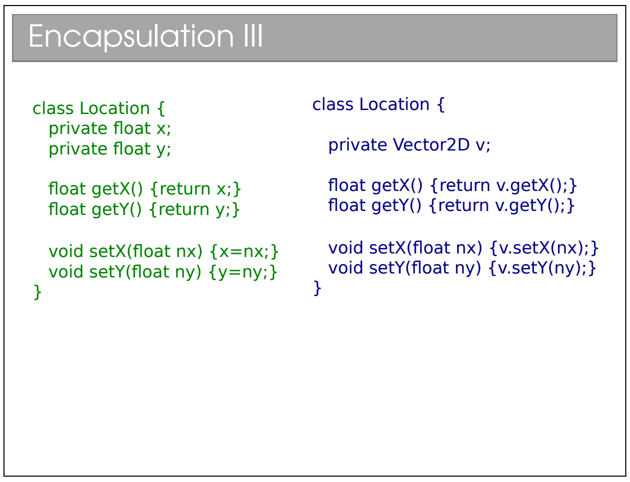
Encapsulation II

```
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.

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.



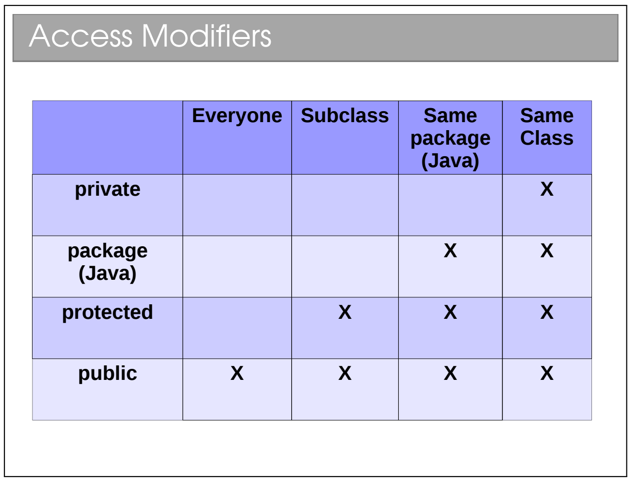
Encapsulation III

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

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

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

```
class Location {

  private Vector2D v;

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

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

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

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



Access Modifiers

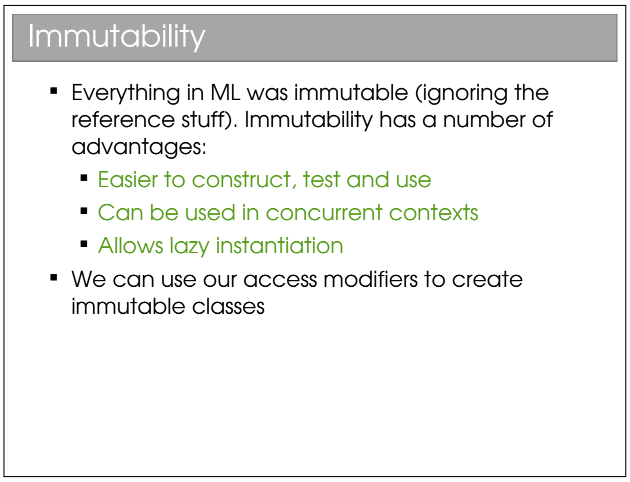
	Everyone	Subclass	Same package (Java)	Same Class
private				X
package (Java)			X	X
protected		X	X	X
public	X	X	X	X

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

2.3 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:

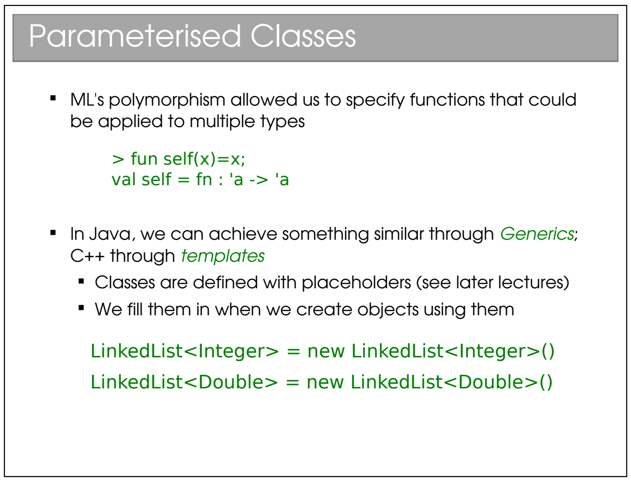
- Make sure all state is private.
- Consider making state final (this just tells the compiler that the value never changes once constructed).
- Make sure no method tries to change any internal state.

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."

2.4 Parameterised Types

We commented earlier that Java lacked the nice polymorphism that type inference gave us. Languages evolve, however, and it has been retrofitted to the language via something called *Generics*. It's not quite the same (it would have been too big a change to put in full type inference), but it does give us similar flexibility.



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>()
```

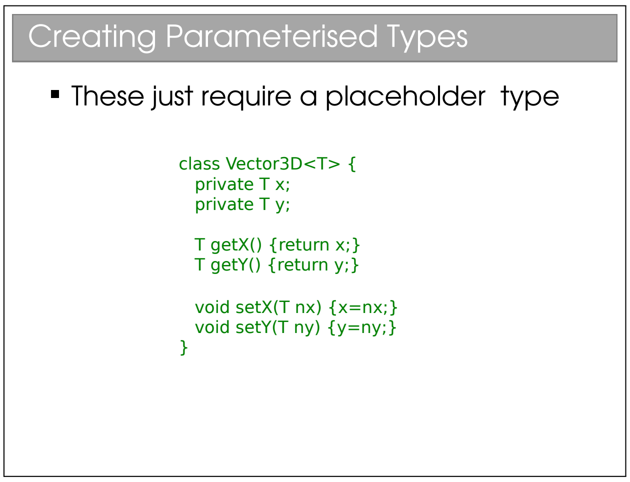
Initially, you will most likely encounter Generics when using Java's built-in data structures such as LinkedList, ArrayList, Map, etc. For example, say you wanted a linked list of integers or Vector3D objects. You would declare:

```
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—a source of many bugs!).

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 simply 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.

2.4.1 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`:

```
Vector3D<Integer> vi = new Vector3D<Integer>(); // Ve
Vector3D<Float> vi = new Vector3D<Float>(); // Vector
Vector3D<Double> vi = new Vector3D<Double>(); // Vect
```

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;
   ...
}
```

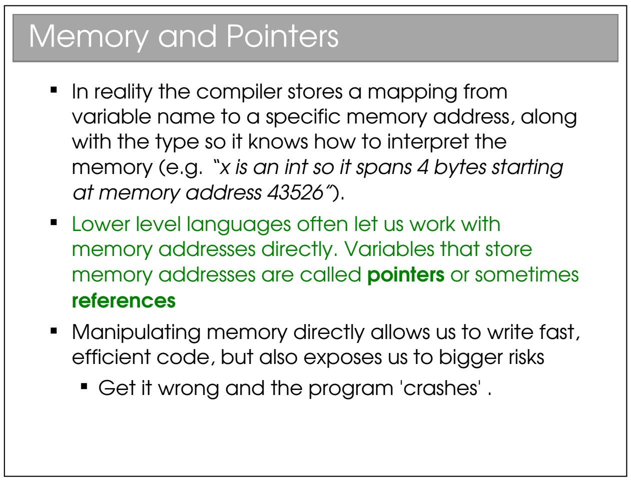
You see this most commonly with `Maps` in Java, which represent dictionaries, mapping keys of some type to values of (potentially) some other type. e.g. a `TreeMap<String,Integer>` could be used to map names to ages).

Lecture 3

Pointers, References and Memory

Imperative languages manipulate state held in system memory. They more naturally extend from assembly and before we go any further we need a mental model of how the compiler uses all this memory.

3.1 Pointers and References

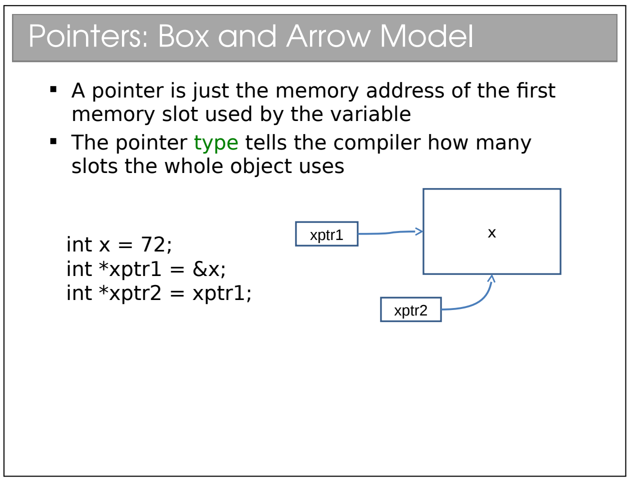


Memory and Pointers

- 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 memory, 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 `int`s are 32 bits) and change the bytes appropriately.

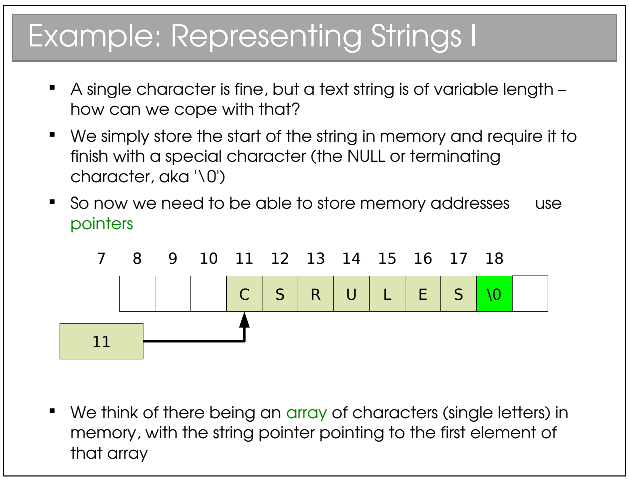
Some languages allow us, as programmers, to move beyond the abstraction of memory provided by explicit variable creation. They allow us to have variables that contain the actual memory addresses and even to manipulate them. We call such variables *pointers* and the traditional way to understand them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

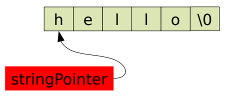
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses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

7	8	9	10	11	12	13	14	15	16	17	18	
				C	S	R	U	L	E	S	\0	

- 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);
```



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

- A reference is an **alias** for another thing (object/array/etc)
- When you use it, you are 'redirected' somehow to the underlying thing
- Properties:
  - Either assigned or unassigned
  - If assigned, it is valid
  - You can easily check if assigned

References are *aliases* for other objects—i.e. they redirect to the 'real' object. You have of course met them in ML.

Implementing References

- A sane reference implementation in an imperative language is going to use pointers
- So each reference is the same as a pointer <u>except</u> that the compiler restricts operations that would violate the properties of references
- For this course, thinking of a reference as a restricted pointer is fine

Any sane implementation of a reference is likely to use pointers (and in FoCS they were directly equated). However, the concept of a reference forbids you from doing all of the things you can do with a pointer:

Distinguishing References and Pointers

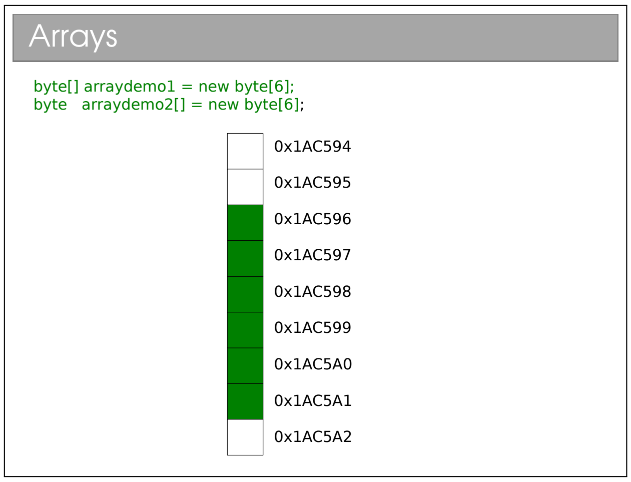
	Pointers	**References**
Can be unassigned (null)	Yes	Yes
Can be assigned to established object	Yes	Yes
Can be assigned to an arbitrary chunk of memory	Yes	**No**
Can be tested for validity	**No**	Yes
Can perform arithmetic	Yes	**No**

The ability to test for validity 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? So references are going to be safer than pointers.

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 lead to you being able to reference arbitrary memory.

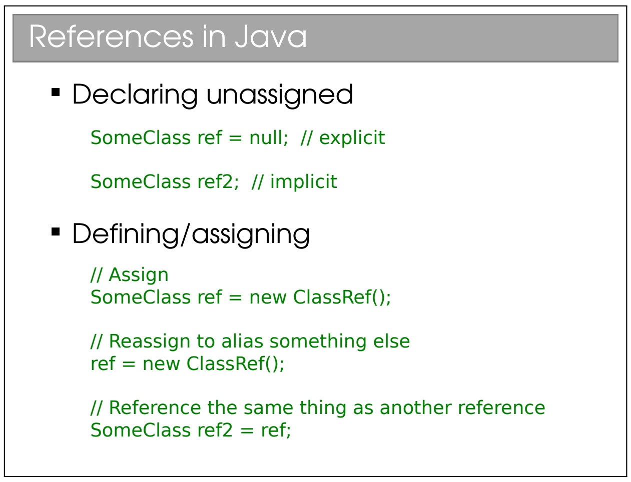
Languages and References

- Pointers are useful but dangerous
- C, C++: pointers *and* references
- Java: references <u>*only*</u>
- ML: references <u>*only*</u>



Arrays

```
byte[] arraydemo1 = new byte[6];
byte   arraydemo2[] = new byte[6];
```



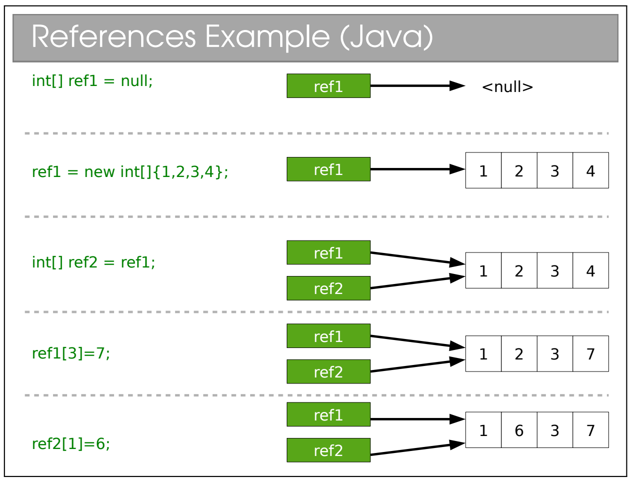
References in Java

- Declaring unassigned

```
SomeClass ref = null;  // explicit

SomeClass ref2;  // implicit
```

- Defining/assigning

```
// Assign
SomeClass ref = new ClassRef();

// Reassign to alias something else
ref = new ClassRef();

// Reference the same thing as another reference
SomeClass ref2 = ref;
```

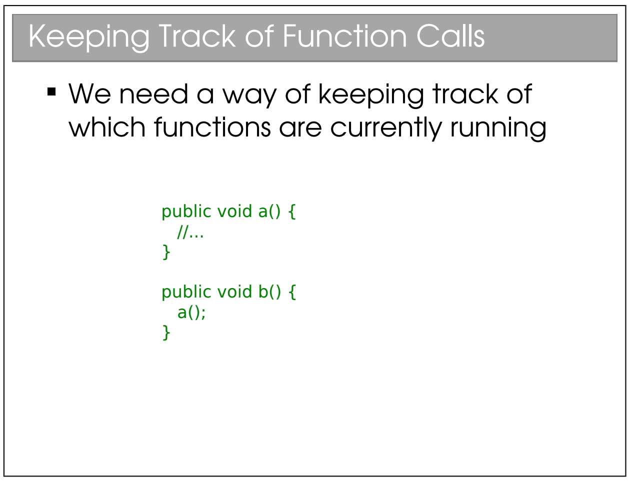


References Example (Java)

```
int[] ref1 = null;

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

int[] ref2 = ref1;

ref1[3]=7;

ref2[1]=6;
```

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. Everything else is a reference type, including arrays and objects.

3.2 Keeping Track of Function Calls: The Call Stack



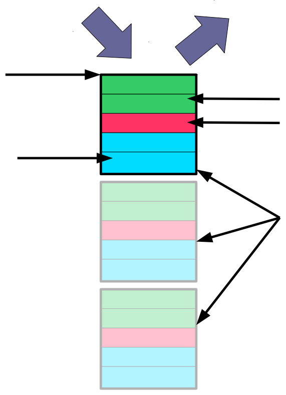
Keeping Track of Function Calls

- We need a way of keeping track of which functions are currently running

```
public void a() {
   //...
}

public void b() {
   a();
}
```

When we call b(), the system must run a() while remembering that we return to b() afterwards. When a function is called from another, this is called *nesting*[1] (just as loops-within-loops are considered *nested*). The nesting can go arbitrarily deep (well, OK, until we run out of memory to keep track). The data structure widely used to keep track is the *call stack*

The Call Stack



Remember the way the fetch-execute cycle handles procedure calls[2]: 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.

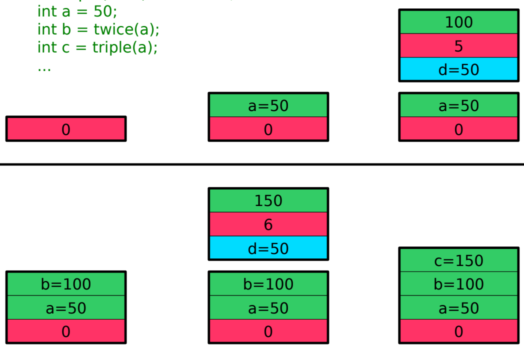
We do this using a *call stack*. A stack is a simple data structure that is the digital analogue of a stack of plates: you add and take from the top of the pile

[1]If the function is calling itself then it is of course *recursive*
[2]Review the pre-arrival course if not

only[3]. 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.

The Call Stack: Example

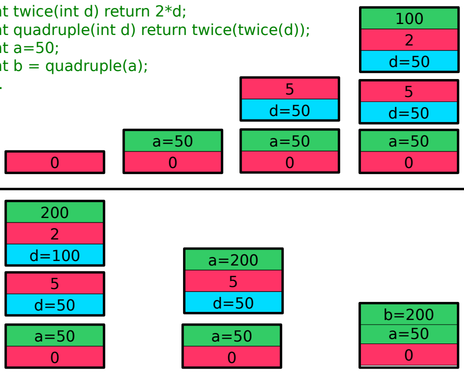
```
1    int twice(int d) return 2*d;
2    int triple(int d) return 3*d;
3    int a = 50;
4    int b = twice(a);
5    int c = triple(a);
6    ...
```



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

Nested Functions

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

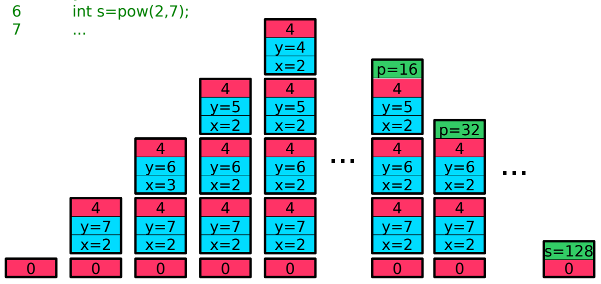


And even more interesting if we add nesting/recursion into the mix:

[3]See Algorithms next term for a full analysis

Recursive Functions

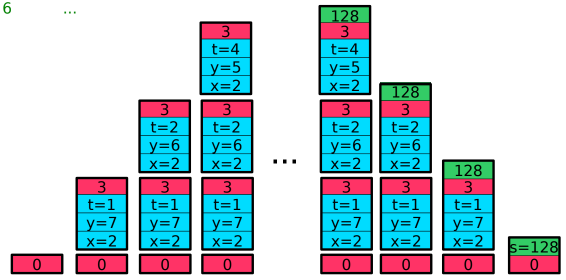
```
1    int pow (int x, int y) {
2        if (y==0) return 1;
3        int p = pow(x,y-1);
4        return x*p;
5    }
6    int s=pow(2,7);
7    ...
```



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:

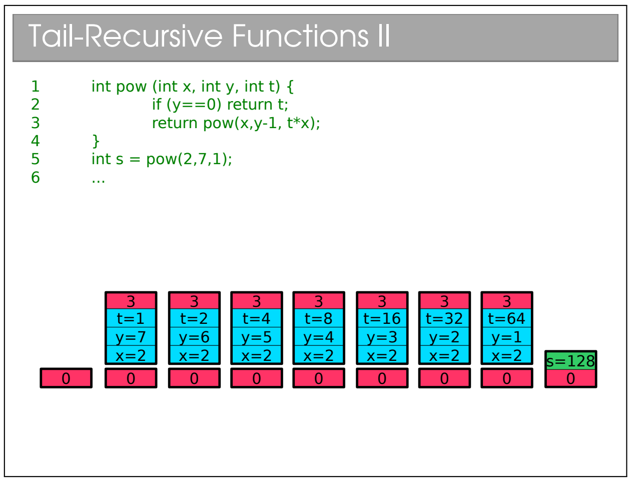
Tail-Recursive Functions I

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



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...[4]

[4]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.



Tail-Recursive Functions II

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

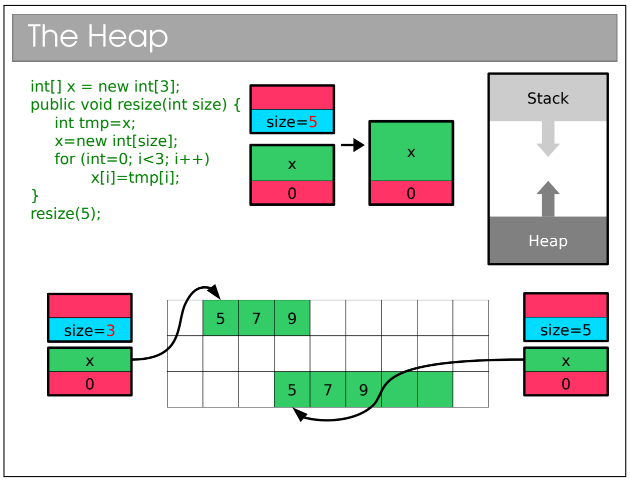
3.3 The Heap

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

We avoid this by using a *heap*[5]. Quite simply we allocate the memory we need from some large pool of free memory, and store a pointer in the stack. Pointers are of known size so won't ever increase. If we want to resize our array, we create a new, bigger array, copy the contents across and update the pointer within the stack.

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

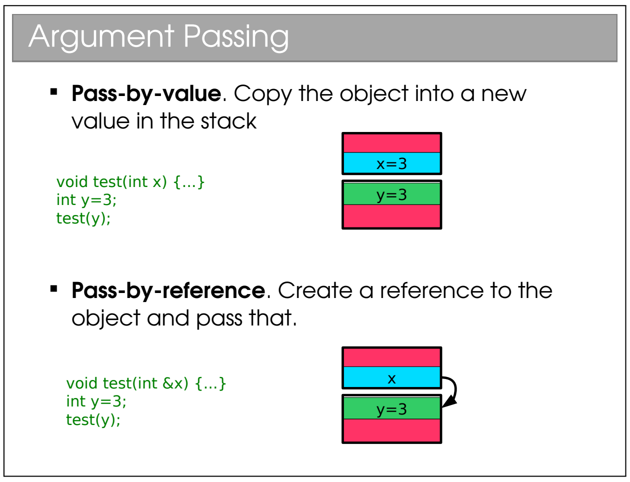


The Heap

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

For those who do the Paper 2 O/S course, you will find 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.

3.4 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.

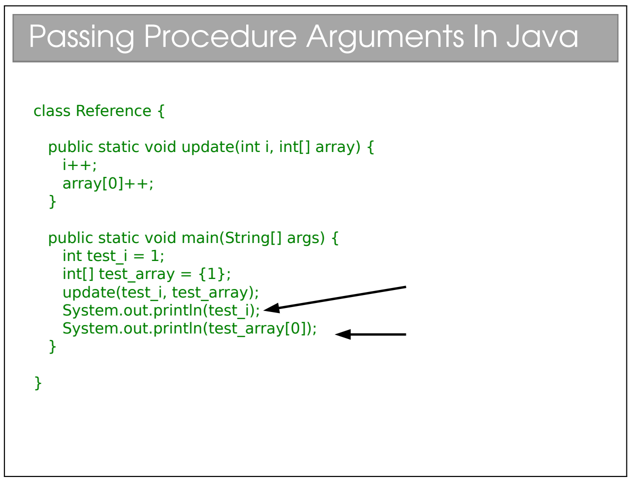
```
void test(int &x) {...}
int y=3;
test(y);
```

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, 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.

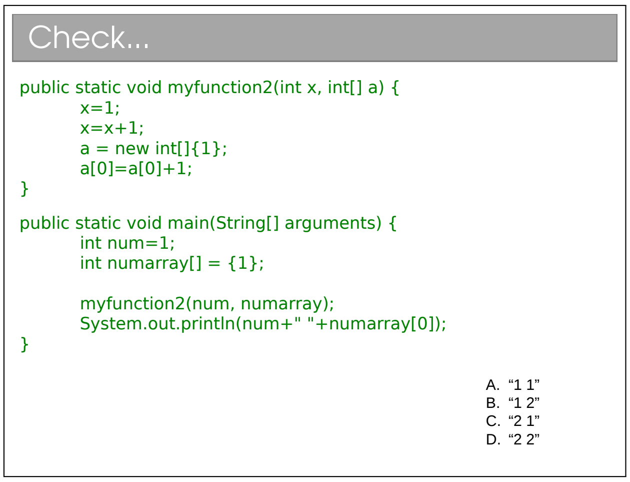
Let's create a model for what happens when we pass a primitive in Java, say an int like test_i. A new stack frame is created and the value of test_i is *copied* into the stack frame. You can do whatever you like to this copy: at the end of the function it is deleted along with the stack frame. The original 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[6].

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.

[6]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"

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

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

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

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

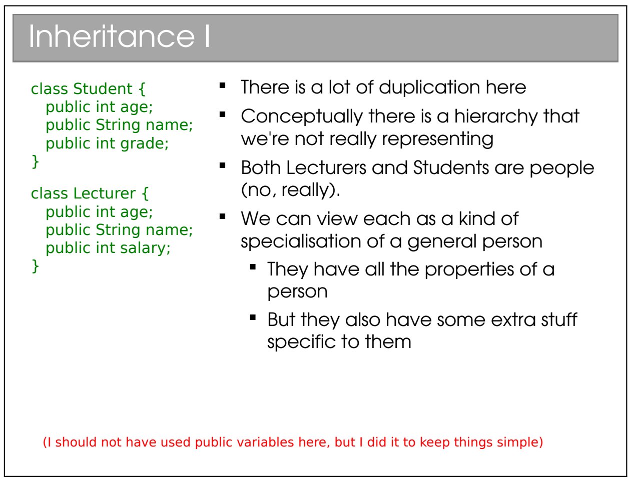
Here I have made a fictional type HugeInt which is meant to represent something that takes a lot of space in memory. Calling either of these functions will give the same answer, but what happens at a low level is quite different. In the first, the variable is copied (lots of memory copying required—bad)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

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

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

Lecture 4

Inheritance

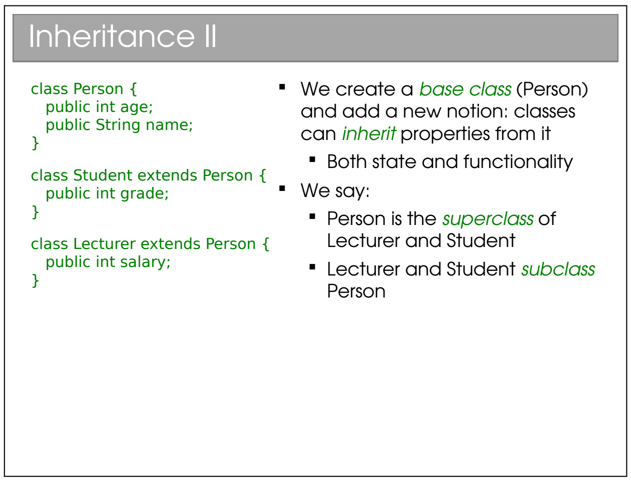


Inheritance I

```
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

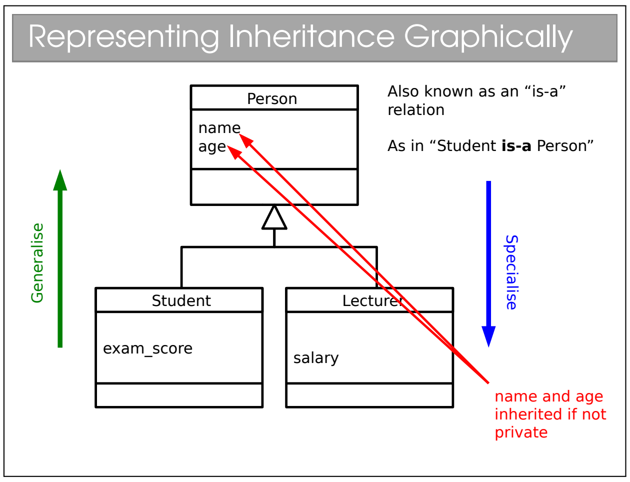
```
class Person {
   public int age;
   public String name;
}
class Student extends Person {
   public int grade;
}
class Lecturer extends Person {
   public int salary;
}
```

- 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


Person
name
age
Student
exam_score
Lecturer
salary
Generalise
Specialise
Also known as an "is-a" relation
As in "Student **is-a** Person"
name and age inherited if not private

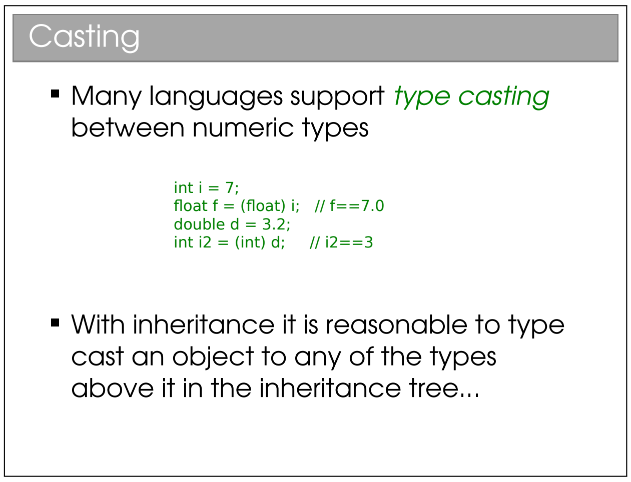
Inheritance is an extremely powerful concept that is used extensively in good OOP. We discussed the "has-a" relation amongst classes; inheritance adds an "is-a" concept. E.g. A car *is a* vehicle that *has a* steering wheel.

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'.

4.1 Casting

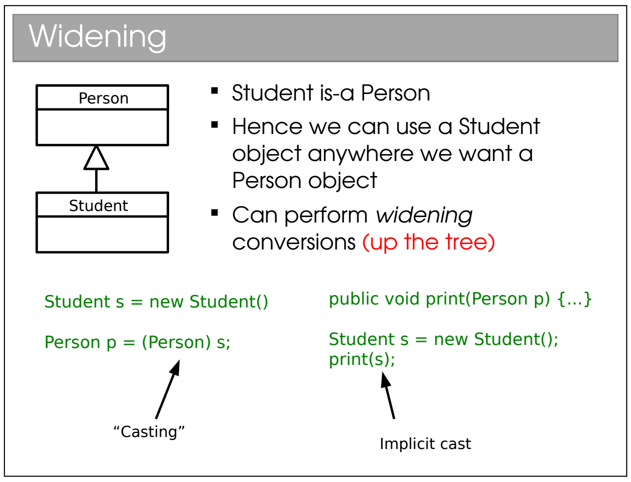


Casting

- 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

Person
Student

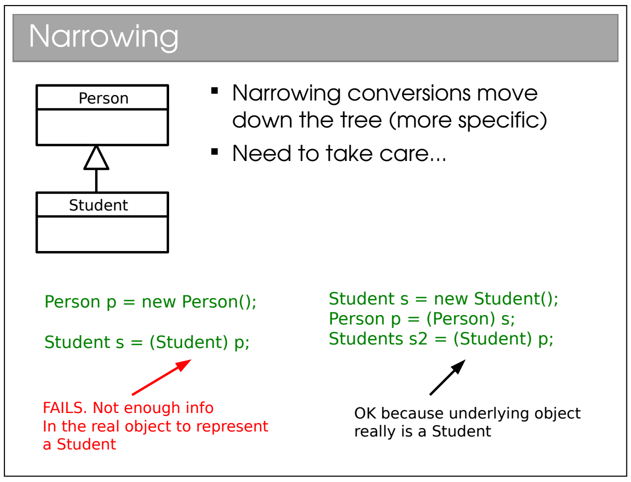
- Student is-a Person
- Hence we can use a Student object anywhere we want a Person object
- Can perform *widening* conversions (up the tree)

```
Student s = new Student()

Person p = (Person) s;
```

"Casting"

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

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

Implicit cast



Narrowing

Person
Student

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

```
Person p = new Person();

Student s = (Student) p;
```

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

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

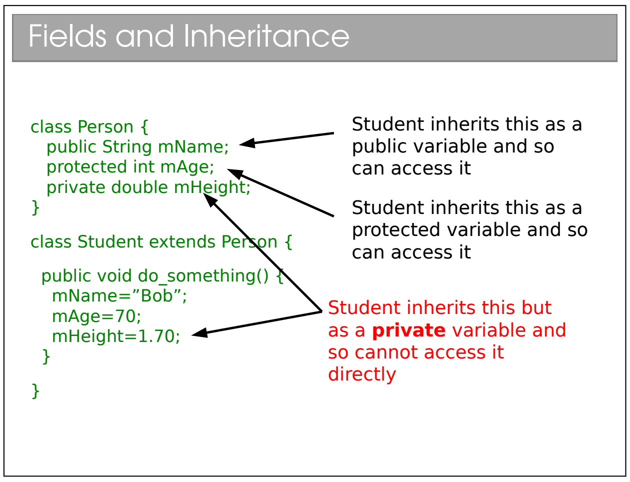
OK because underlying object
really is a Student

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;
  }

}
```

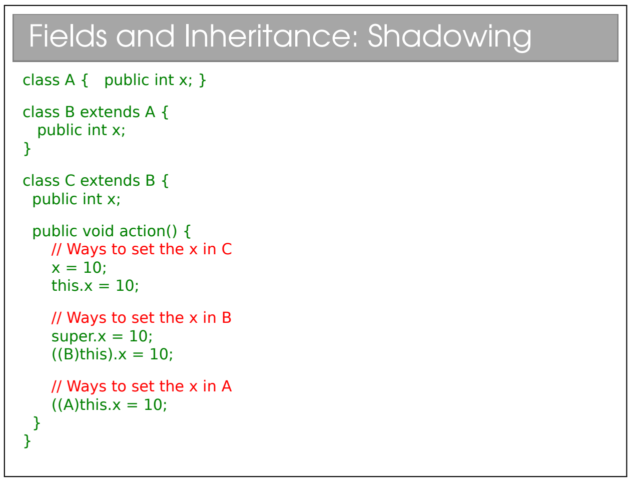
Student inherits this as a public variable and so can access it

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[1].

[1] 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.



Fields and Inheritance: Shadowing

```
class A {   public int x; }

class B extends A {
   public int x;
}

class C extends B {
  public int x;

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

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

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

What happens here?? There is an inheritance tree (A is the parent of B is the parent of C). Each of these declares an integer field with the name x. In memory, you will find three allocated integers for every object of type C. We say that variables in parent classes with the same name as those in child classes are *shadowed*.

Note that the variables are genuinely being shadowed and nothing is being replaced. This is in contrast to the behaviour with methods...

NB: A common novice error is to assume that we have to redeclare a field in its subclasses for it to be inherited: not so. *Every* 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[2] and provides us with a reference to the current object. In fact, the this keyword is what you need to access anything within a class, but because we'd end up writing this all over the place, it is taken as implicit. So, for example:

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

becomes:

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

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

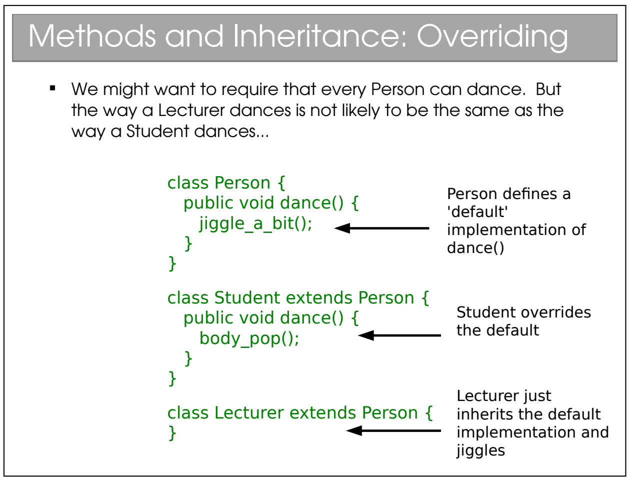
The super keyword gives us access to the direct parent (one step up in the tree). You've met both keywords in your Java practicals.

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.

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:



Methods and Inheritance: Overriding

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

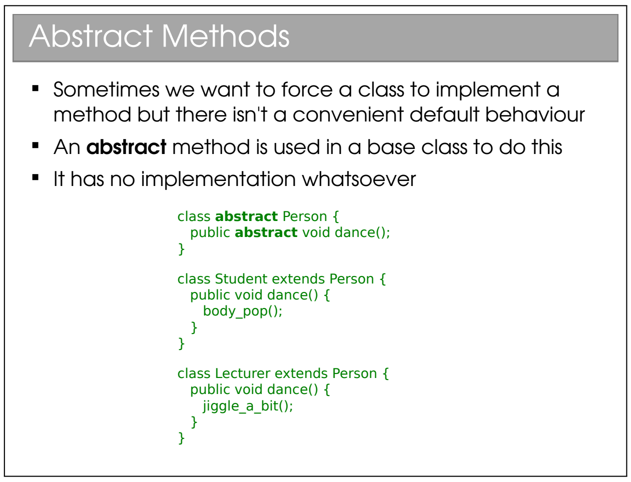
```
class Person {
  public void dance() {
    jiggle_a_bit();
  }
}

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

class Lecturer extends Person {
}
```

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

Every object that has Person for a parent must have a dance() method since it is defined in the Person class and is inherited. 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

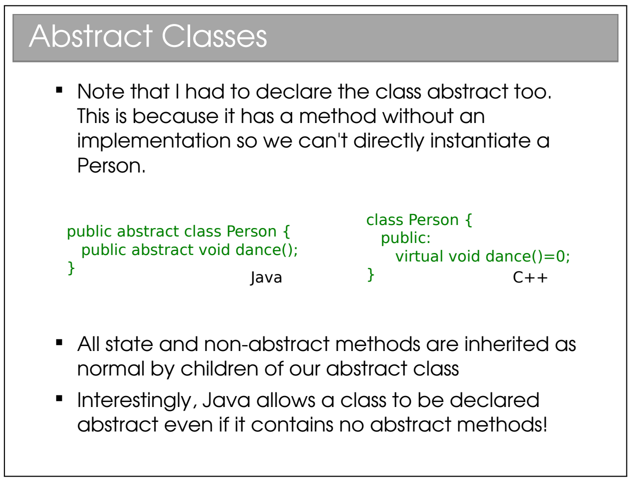


Abstract Methods

- 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.

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

Java

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

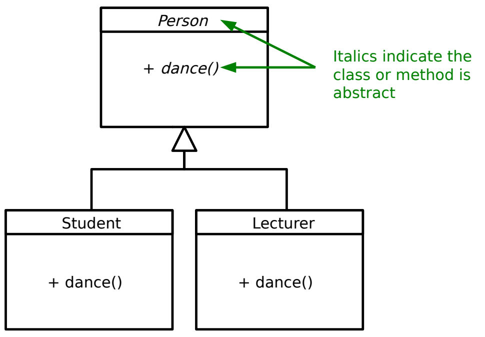
C++

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

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

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

Representing Abstract Classes

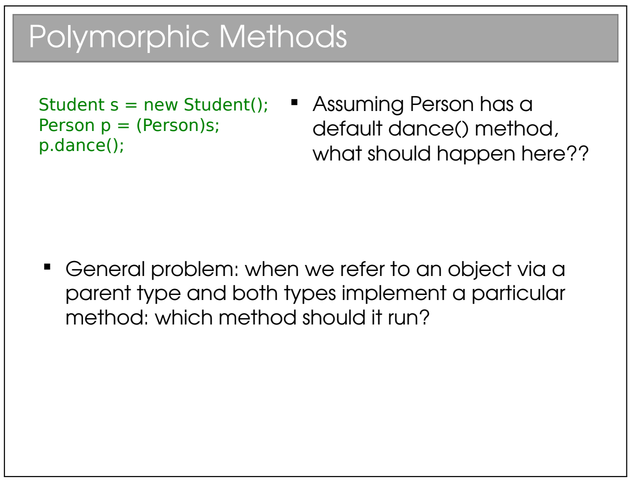


You have to look at UML diagrams carefully since the italics that represent abstract methods or classes aren't always obvious on a quick glance.

Lecture 5

Polymorphism

You should be comfortable with the polymorphism[1] 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:

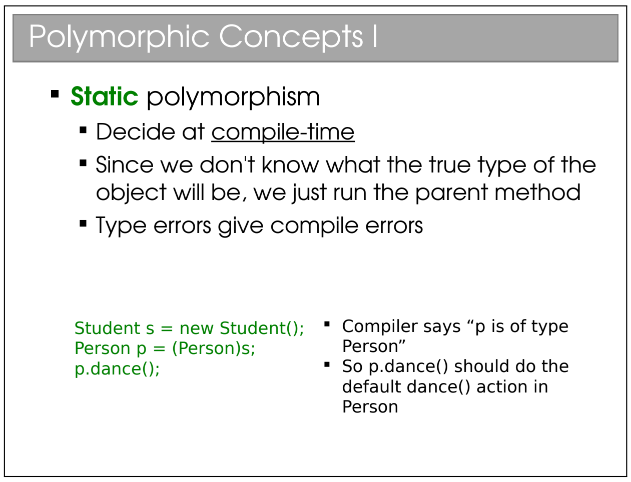


Polymorphic Methods

```
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?



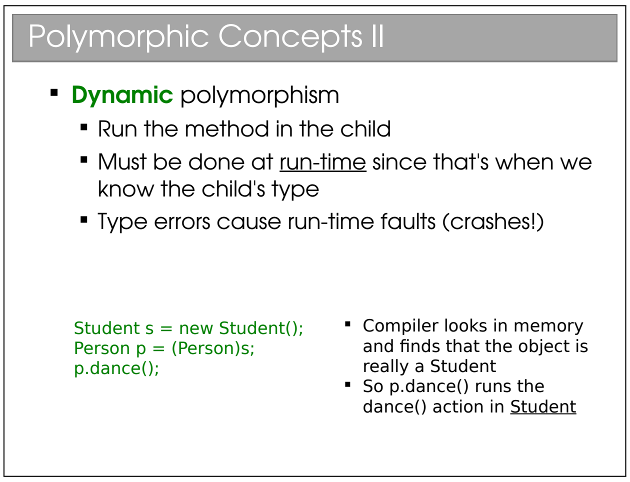
Polymorphic Concepts I

- **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 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

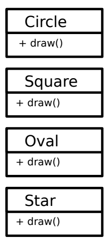
Here we get the same method implementation regardless of what we cast the object to. In order to be sure that it gets this right, we can't figure out which method to run when we are compiling. Instead, the system has to run the program and, when a decision needs to be made about which method to run, 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.

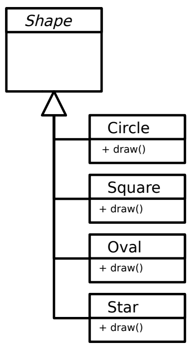
[1]The etymology of the word polymorphism is from the ancient Greek: *poly* (many)–*morph* (form)–ism

The Canonical Example I



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

The Canonical Example II

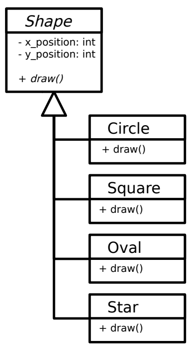


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

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

  - What if we want to add a new shape?

The Canonical Example III



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

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

  - What if we want to add a new shape?

Implementations

- Java
  - All methods are 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 <u>sure</u> you understand...

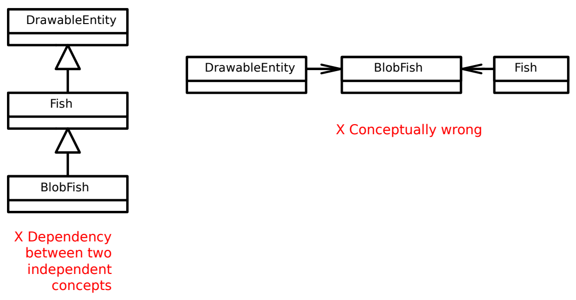
C++ allows you to choose whether methods are inherited statically (default) or dynamically (explicitly labelled with the keyword `virtual`). This can be good for performance (you only incur the dynamic overhead when you need to) but gets complicated, especially if the base method isn't dynamic but a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

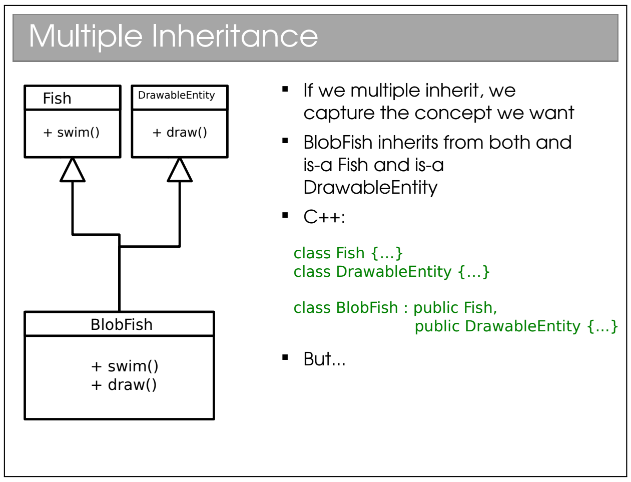
The Java designers avoided the problem by enforcing dynamic polymorphism. You may find reference to `final` methods being Java's static polymorphism since this gives a compile error if you try to override it in subclasses. To me, this isn't quite the same: it's not making a choice between multiple implementations but rather enforcing that there can only be one implementation!

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?





Fish
+ swim()
DrawableEntity
+ draw()
BlobFish
+ swim()
+ draw()

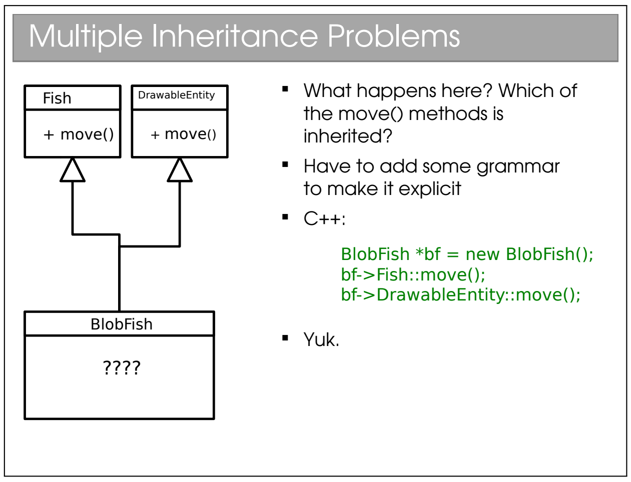
Multiple Inheritance

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

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

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

- But...

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



Fish
+ move()
DrawableEntity
+ move()
BlobFish
????

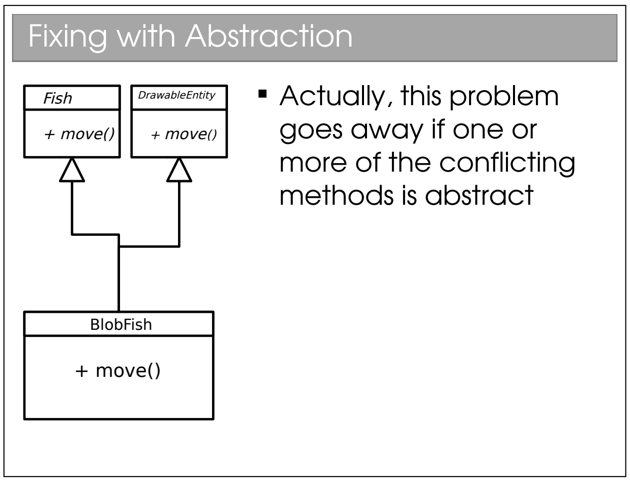
Multiple Inheritance Problems

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

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

- Yuk.

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

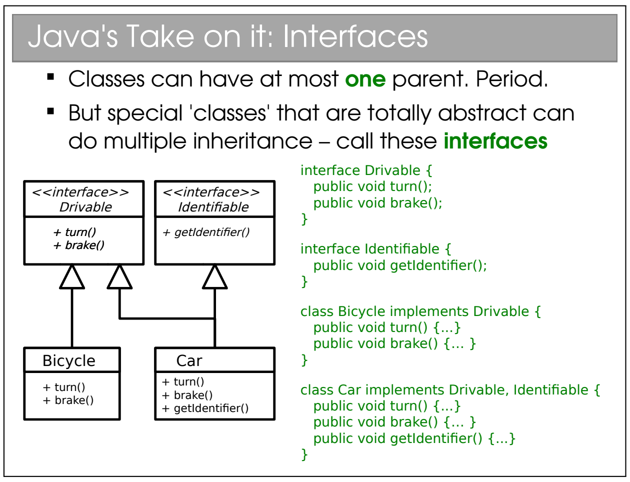


Fish
+ move()
DrawableEntity
+ move()
BlobFish
+ move()

Fixing with Abstraction

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

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



<<interface>>
Drivable
+ turn()
+ brake()
<<interface>>
Identifiable
+ getIdentifier()
Bicycle
+ turn()
+ brake()
Car
+ turn()
+ brake()
+ getIdentifier()

Java's Take on it: Interfaces

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

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

interface Identifiable {
  public void getIdentifier();
}

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

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

So Java allows you to inherit from one class *only* (which may itself inherit from one other, which may itself...). Many programmers coming from C++ find this limiting, but it just means you have to think of another way to represent your classes (often a better way, although not always!).

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

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

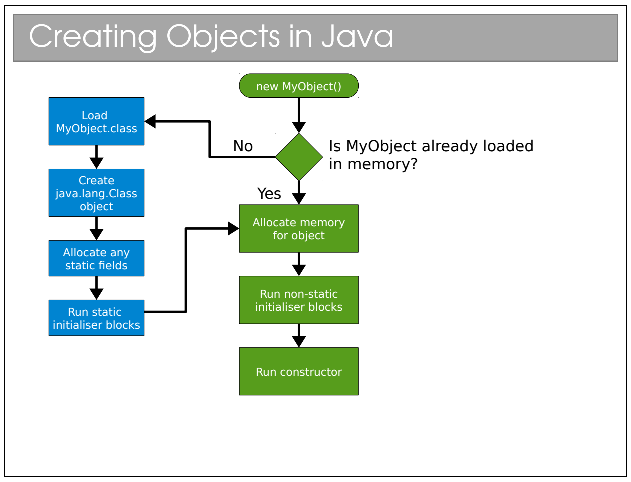
This is a greatly simplified concept that allows for multiple inheritance without any chance of conflict. Interfaces are represented in our UML class diagram with a preceding <<interface>> label and inheritance occurs via the implements keyword rather than through extends.

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

Lecture 6

Lifecycle of an Object

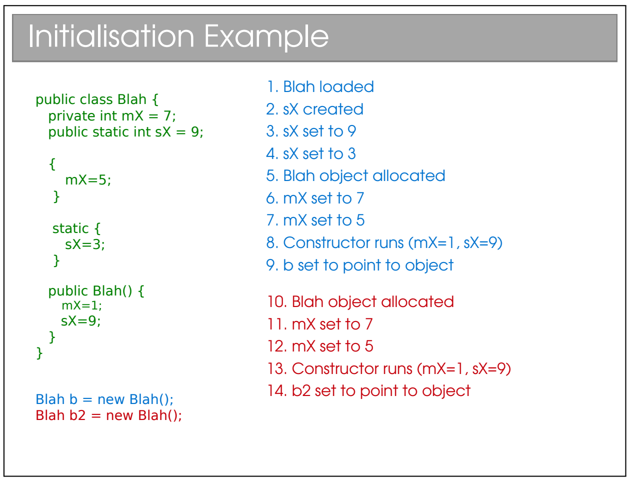
We met constructors earlier in the course as methods that initialise objects. We can now add a bit more detail. When you request a new object, Java will do quite a lot of work:



Creating Objects in Java

new MyObject()
Is MyObject already loaded in memory?
No
Yes
Load MyObject.class
Create java.lang.Class object
Allocate any static fields
Run static initialiser blocks
Allocate memory for object
Run non-static initialiser blocks
Run constructor

Note that Java maintains a java.lang.Class object for every class it loads into memory from a .class file. This object actually allows you query things about the class, such as its name or to list all the methods it has. The ability to do inspect (and possibly modify!) a program's structure is a feature called *reflection*. It's quite a powerful feature that exists in some (but certainly not all) languages. It's out of scope here but worth exploring if you're interested.



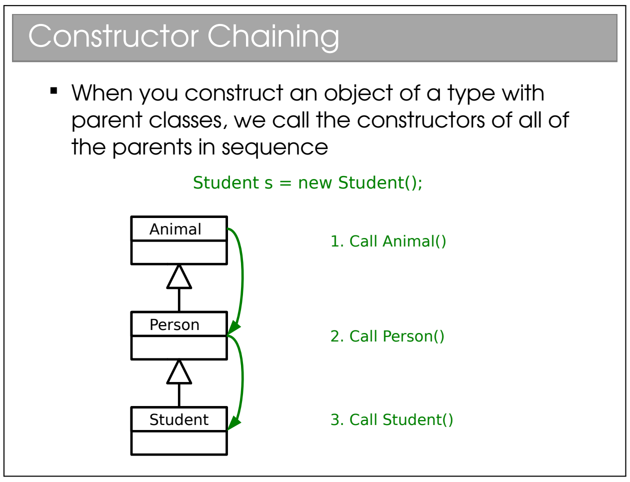
Initialisation Example

```
public class Blah {
  private int mX = 7;
  public static int sX = 9;

  {
     mX=5;
   }

   static {
     sX=3;
   }

  public Blah() {
    mX=1;
    sX=9;
  }
}

Blah b = new Blah();
Blah b2 = new Blah();
```

1. Blah loaded
2. sX created
3. sX set to 9
4. sX set to 3
5. Blah object allocated
6. mX set to 7
7. mX set to 5
8. Constructor runs (mX=1, sX=9)
9. b set to point to object

10. Blah object allocated
11. mX set to 7
12. mX set to 5
13. Constructor runs (mX=1, sX=9)
14. b2 set to point to object

Things get even more complex when we throw in some inheritance:



Constructor Chaining

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

`Student s = new Student();`

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

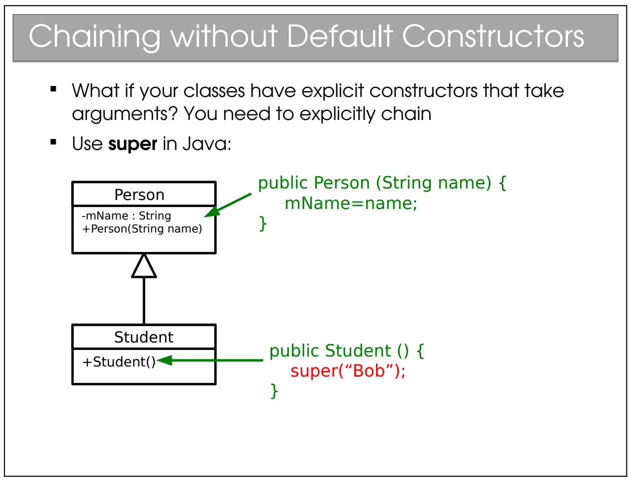
```
public class Person {
  public Person() {
    }
  }
```

becomes:

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

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

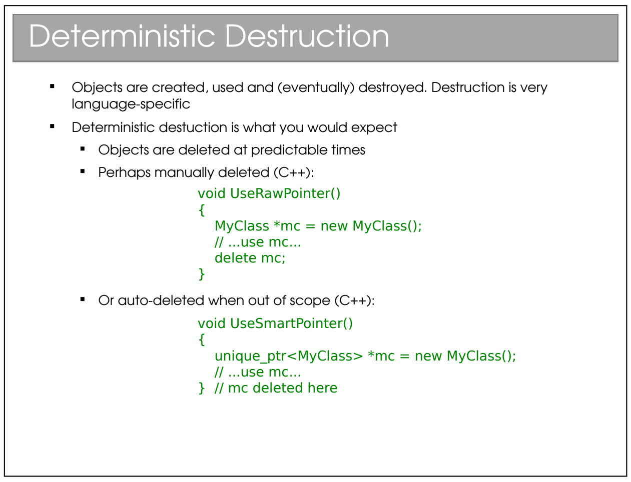
```
  class Child : public Parent1, Parent2 {
    public:
      Child() : Parent1("Alice"), Parent2("Bob") {...}
  }
```



Chaining without Default Constructors

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

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

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



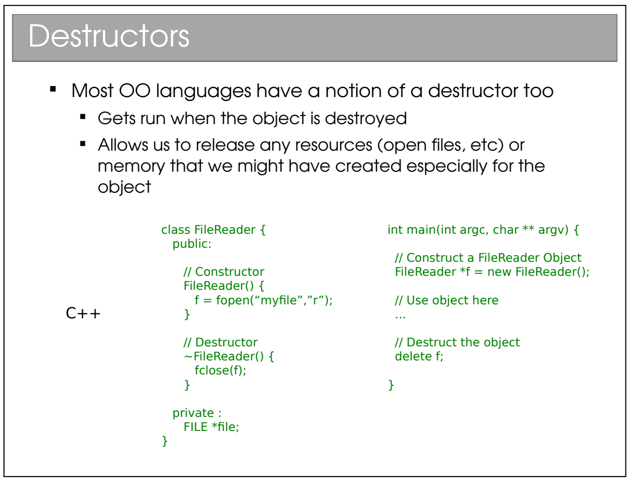
Deterministic Destruction

- Objects are created, used and (eventually) destroyed. Destruction is very language-specific
- Deterministic destuction is what you would expect
  - Objects are deleted at predictable times
  - Perhaps manually deleted (C++):

```
void UseRawPointer()
{
   MyClass *mc = new MyClass();
   // ...use mc...
   delete mc;
}
```

  - Or auto-deleted when out of scope (C++):

```
void UseSmartPointer()
{
   unique_ptr<MyClass> *mc = new MyClass();
   // ...use mc...
}  // mc deleted here
```



Destructors

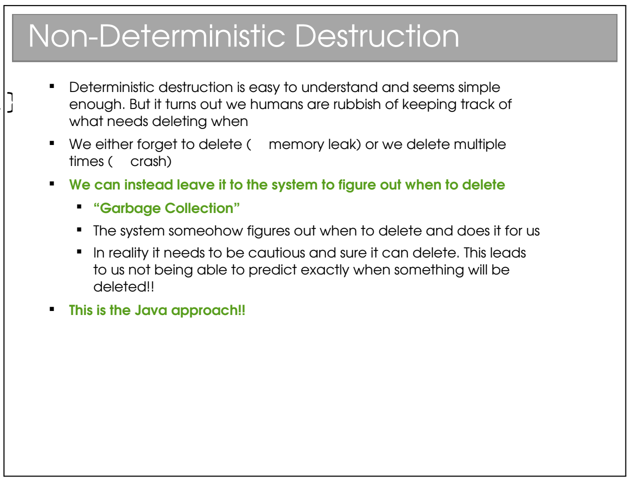
- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

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

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

  private :
    FILE *file;
}
```

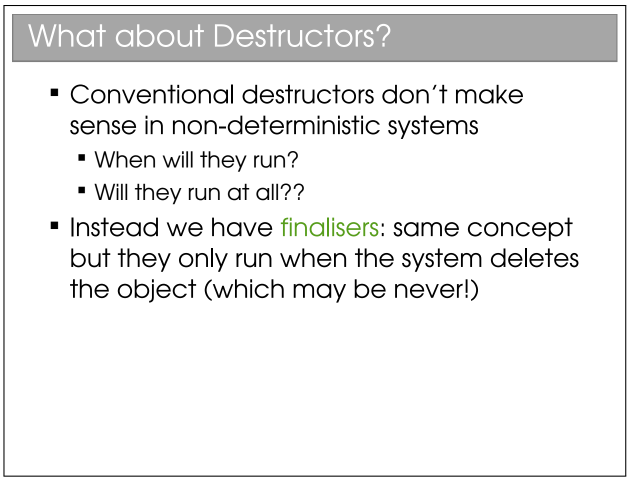
```
int main(int argc, char ** argv) {

  // 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.



Non-Deterministic Destruction

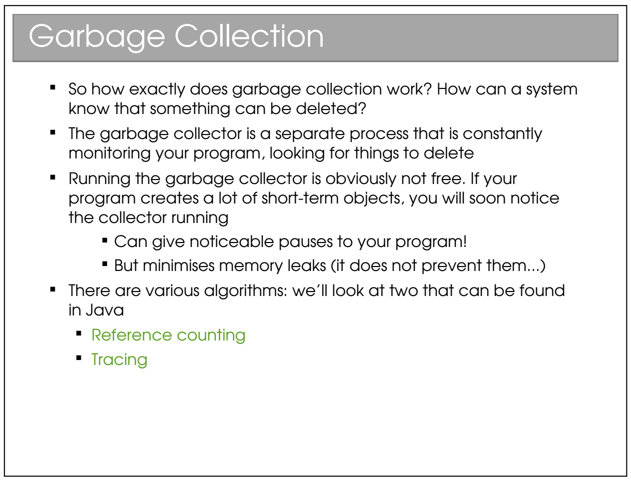
- Deterministic destruction is easy to understand and seems simple enough. But it turns out we humans are rubbish of keeping track of what needs deleting when
- We either forget to delete (memory leak) or we delete multiple times (crash)
- **We can instead leave it to the system to figure out when to delete**
  - **"Garbage Collection"**
  - The system someohow figures out when to delete and does it for us
  - In reality it needs to be cautious and sure it can delete. This leads to us not being able to predict exactly when something will be deleted!!
- **This is the Java approach!!**



What about Destructors?

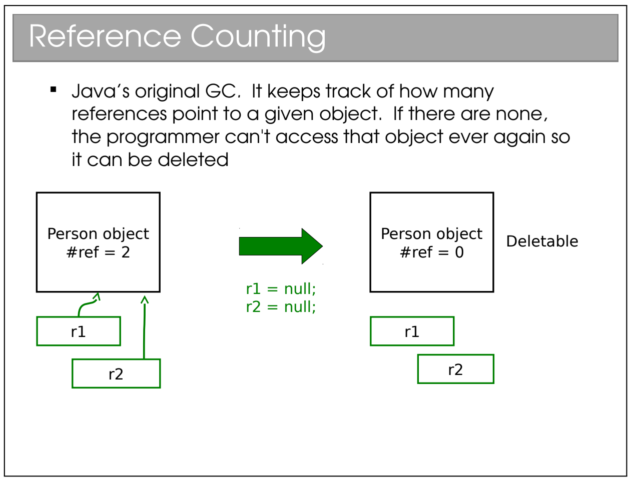
- Conventional destructors don't make sense in non-deterministic systems
  - When will they run?
  - Will they run at all??
- Instead we have finalisers: same concept but they only run when the system deletes the object (which may be never!)

OK, so a finaliser is just a rebadged destructor, but the rebadging is important. It reminds us as programmers that it won't run deterministically. 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.



Garbage Collection

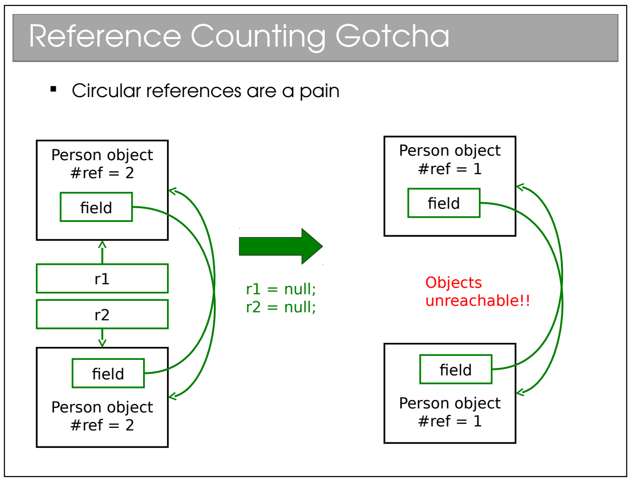
- So how exactly does garbage collection work? How can a system know that something can be deleted?
- The garbage collector is a separate process that is constantly monitoring your program, looking for things to delete
- Running the garbage collector is obviously not free. If your program creates a lot of short-term objects, you will soon notice the collector running
  - Can give noticeable pauses to your program!
  - But minimises memory leaks (it does not prevent them…)
- There are various algorithms: we'll look at two that can be found in Java
  - Reference counting
  - Tracing



Reference Counting

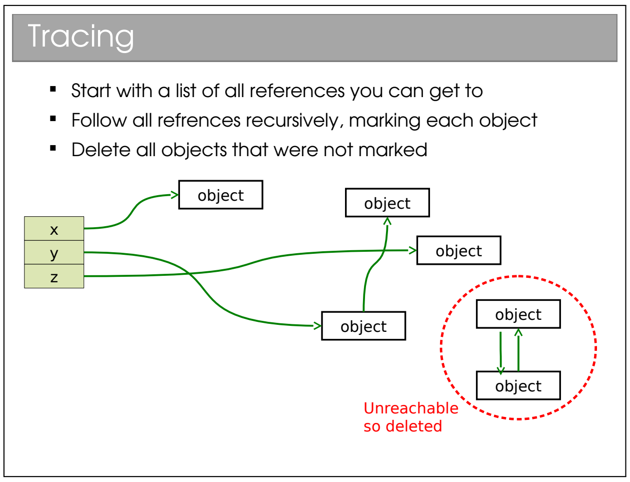
- Java's original GC. It keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted

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.



Reference Counting Gotcha

- Circular references are a pain



Tracing

- Start with a list of all references you can get to
- Follow all refrences recursively, marking each object
- Delete all objects that were not marked