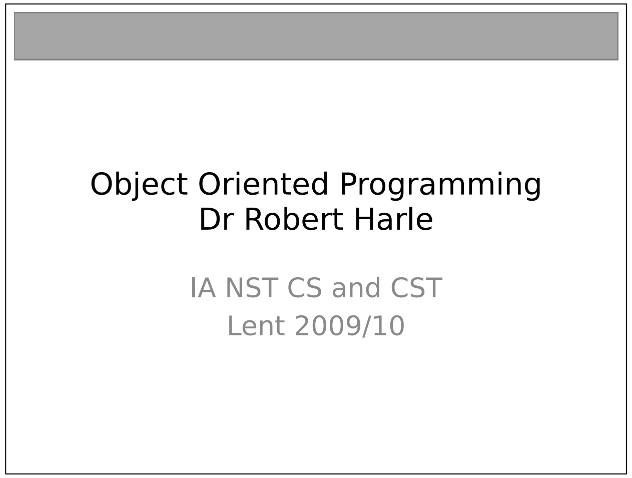
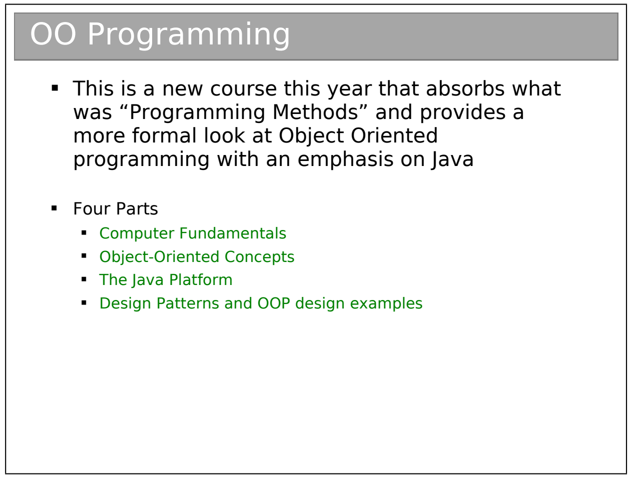


Object Oriented Programming
Dr Robert Harle

IA NST CS and CST
Lent 2009/10



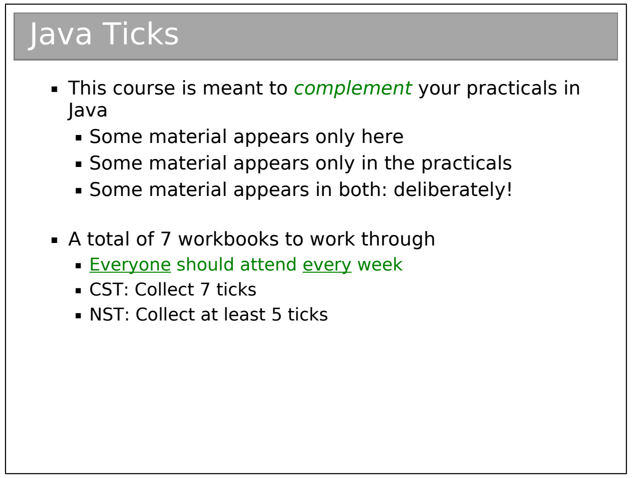
OO Programming

- This is a new course this year that absorbs what was “Programming Methods” and provides a more formal look at Object Oriented programming with an emphasis on Java
- Four Parts
  - Computer Fundamentals
  - Object-Oriented Concepts
  - The Java Platform
  - Design Patterns and OOP design examples

If we teach Java in isolation, there’s a good chance that students don’t manage to mentally separate the object-oriented concepts from

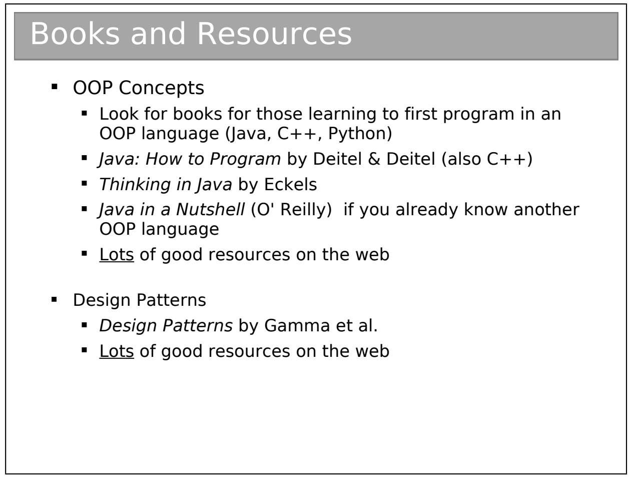
Java’s implementation of them. Understanding the underlying principles of OOP allows you to transition quickly to a new OOP language. Because Java is the chosen teaching language here, the vast majority of what I do will be in Java, but with the occasional other language thrown in to make a point.

Actually learning to program is best done practically. That’s why you have your practicals. I don’t want to concentrate on the minutiae of programming, but rather the larger principles. It might be that some of this stuff will make more sense when you’ve done your practicals; I hope that this material will help set you up to complete your practicals quickly and efficiently. But we’ll see.



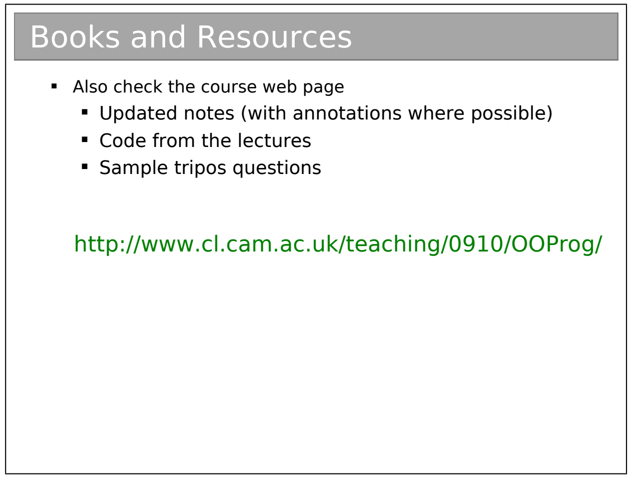
Java Ticks

- This course is meant to *complement* your practicals in Java
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately!
- A total of 7 workbooks to work through
  - Everyone should attend every week
  - CST: Collect 7 ticks
  - NST: Collect at least 5 ticks



Books and Resources

- OOP Concepts
  - Look for books for those learning to first program in an OOP language (Java, C++, Python)
  - *Java: How to Program* by Deitel & Deitel (also C++)
  - *Thinking in Java* by Eckels
  - *Java in a Nutshell* (O' Reilly) if you already know another OOP language
  - Lots of good resources on the web
- Design Patterns
  - *Design Patterns* by Gamma et al.
  - Lots of good resources on the web



Books and Resources

- 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/0910/OOProg/

There really is no shortage of books and websites describing the basics of object oriented programming. The concepts themselves

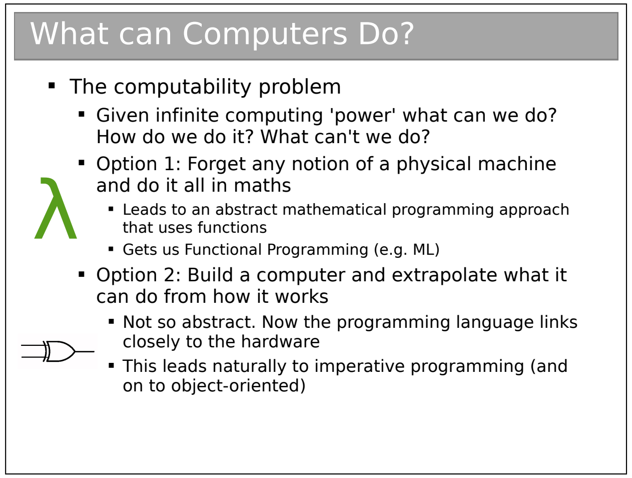
are quite abstract, but most texts will use a specific language to demonstrate them. The books I've given favour Java (because that's the primary language you learn this term). You shouldn't see that as a dis-recommendation for other books. In terms of websites, SUN produce a series of tutorials for Java, which cover OOP:

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

but you'll find lots of other good resources if you search. And don't forget your practical workbooks, which will aim *not* to assume anything from these lectures.

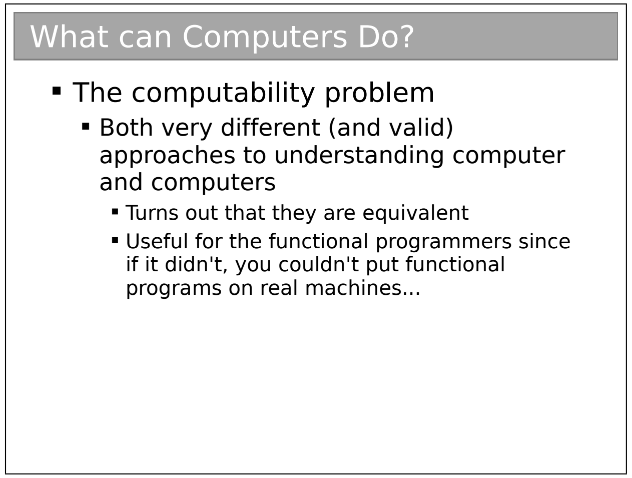
Chapter 1

Computer Fundamentals



What can Computers Do?

- The computability problem
  - Given infinite computing 'power' what can we do? How do we do it? What can't we do?
  - Option 1: Forget any notion of a physical machine and do it all in maths
    - Leads to an abstract mathematical programming approach that uses functions
    - Gets us Functional Programming (e.g. ML)
  - Option 2: Build a computer and extrapolate what it can do from how it works
    - Not so abstract. Now the programming language links closely to the hardware
    - This leads naturally to imperative programming (and on to object-oriented)



What can Computers Do?

- The computability problem
  - Both very different (and valid) approaches to understanding computer and computers
    - Turns out that they are equivalent
    - Useful for the functional programmers since if it didn't, you couldn't put functional programs on real machines...

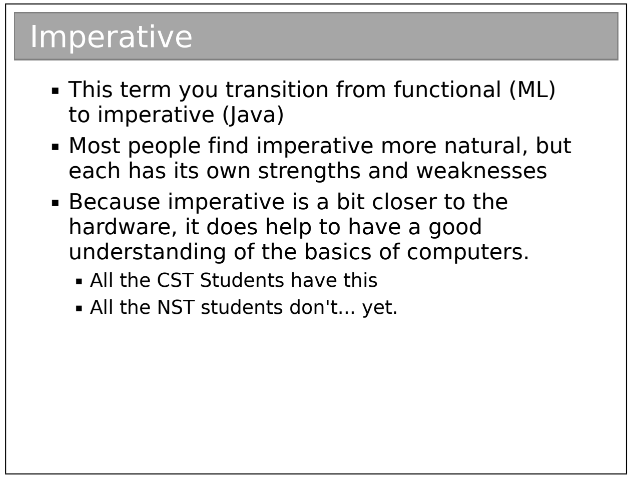
WWII spurred an interest in machinery that could compute. During the war, this interest was stoked by a need to break codes, but also to compute relatively mundane quantities such as the trajectory of an artillery shell. After the war, interest continued in the abstract notion of 'computability'. Brilliant minds (Alan Turing, etc) began to wonder what was and wasn't 'computable'. They defined this in an abstract way: given an infinite computing power (whatever that is), could they solve anything? Are there things that can't be solved by machines?

Roughly two approaches to answering these questions appeared:

Maths, maths, maths. Ignore the mechanics of any real machine and imagine a hypothetical machine that had infinite computation power, then write some sort of 'programming language' for it. This language turned out to be a form of mathematics known as 'Lambda calculus'. It was based around the notion of *functions* in a mathematical sense.

Build it and they will come. Understand how to build and use a real computing machine. This resulted in the von Neumann architecture and the notion of a Turing machine (basically what we would call a computer).

It turned out that both approaches were useful for answering the fundamental questions. In fact, they can be proven to be the same thing now!



Imperative

- This term you transition from functional (ML) to imperative (Java)
- Most people find imperative more natural, but each has its own strengths and weaknesses
- Because imperative is a bit closer to the hardware, it does help to have a good understanding of the basics of computers.
  - All the CST Students have this
  - All the NST students don't... yet.

All this is very nice, but why do you care? Well, there's a legacy from these approaches, in the form of two *programming paradigms*:

Functional Languages. These are very mathematically oriented and have the notion of functions as the building blocks of computation.

Imperative or Procedural Languages. These have variables (state) and procedures as the main building blocks[1]. Such languages are well known—e.g. C—and include what we call *object-oriented* languages such as C++ and Java.

Note that I have pluralised "Language" in the above sentences. Terms like "Object-oriented" are really a set of ideas and concepts that various languages implement to varying degrees.

Last term you toured around Computer Science (in FoCS) and used a particular language to do it (ML). Although predominantly a functional programming language, ML has acquired a few imperative 'features' so you shouldn't consider it as a pure functional language. (For example, the ML reference types you looked at are not functional!).

This term you will shift attention to an *object-oriented* language in the form of Java. What we will be doing in this course is looking at the paradigm of object-oriented programming itself so you can better understand the underlying ideas and separate the Java from the paradigm.

1.1 Hardware Fundamentals

As I said before, the imperative style of programming maps quite easily to the underlying computer hardware. A good understanding of how computers work can greatly improve your programming capabilities with an imperative language. Thing is, only around a half of you have been taught the fundamentals of a computer (the

[1]A *procedure* is a function-like chunk of code that takes inputs and gives outputs. However, unlike a formal function it can have side effects i.e. can make non-local changes to something other than its inputs and declared outputs.

CST half). Those people will undoubtedly be rather bored by what follows in the next few pages, but it's necessary so that everyone is at the same point when we start to delve deeper into object-oriented programming.

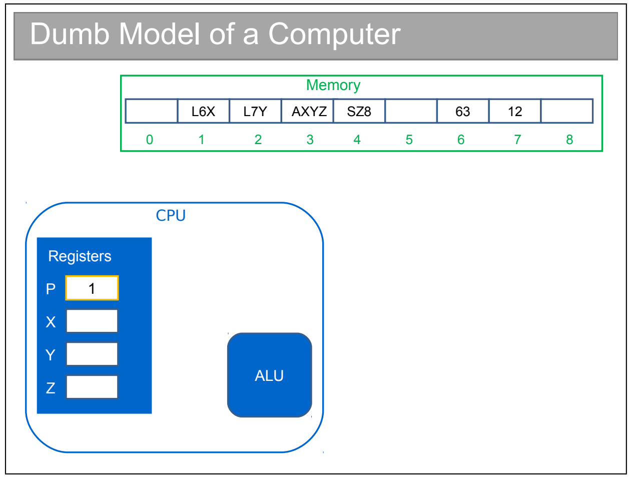
Computers do lots of very simple things very fast. Over the years we have found optimisation after optimisation to make the simple processes that little bit quicker, but really the fundamentals involve some memory to store information and a CPU to perform simple actions on small chunks of it.

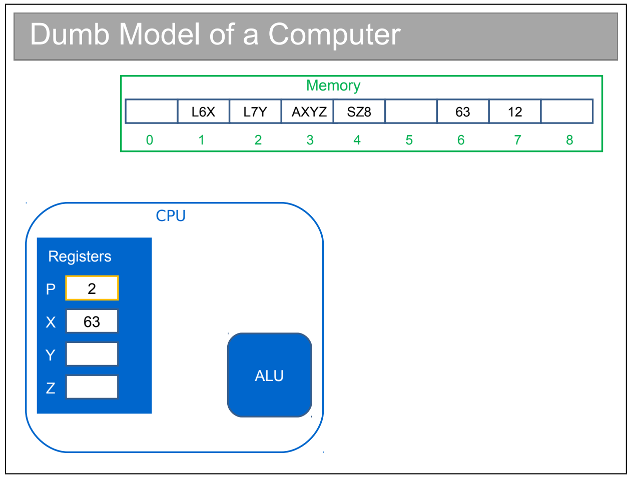
We use lots of different types of memory, but conceptually only two are of interest here. System memory is a very large pool of memory (the 2GB or so you get when you buy a machine). Then there are some really fast, but small chunks of memory called registers. These are built into the CPU itself.

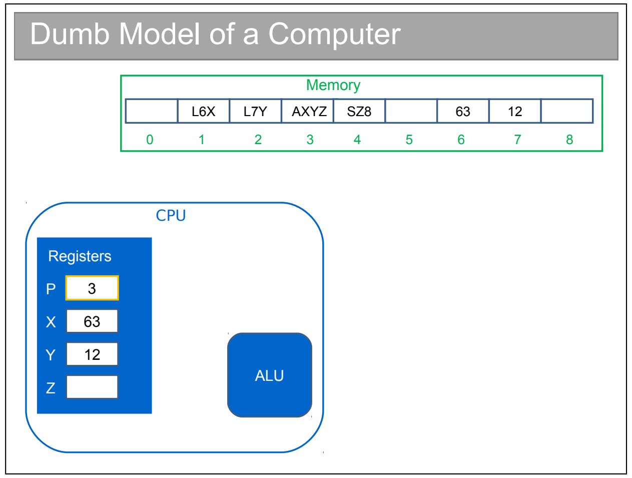
The CPU acts only on the chunks in the registers so the computer is constantly *loading* chunks of data from system memory into registers and operating on the register values.

The example that follows loads in two numbers and adds them. Those doing the Operating Systems course will realise this is just the *fetch-execute* cycle. Basically, there is a special register called the program counter (marked P) that tells the computer where to look to get its next instruction. Here I've made up some operations:

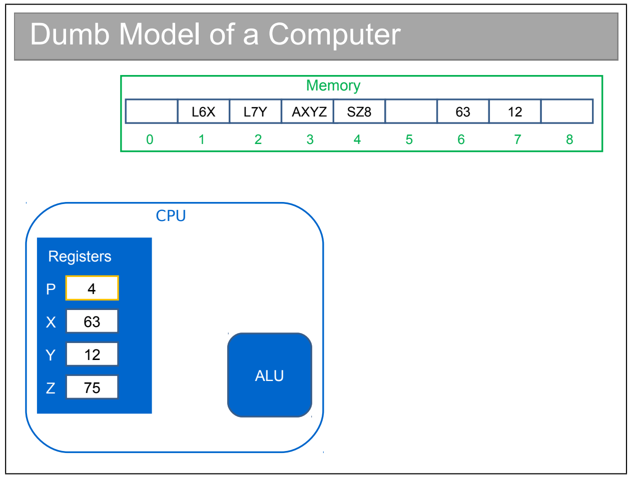
LAM. LOAD the value in memory slot A into register M
AMNO. ADD the values in registers M and N and put the result in register O.
SMA. STORE the value in register M into memory slot A





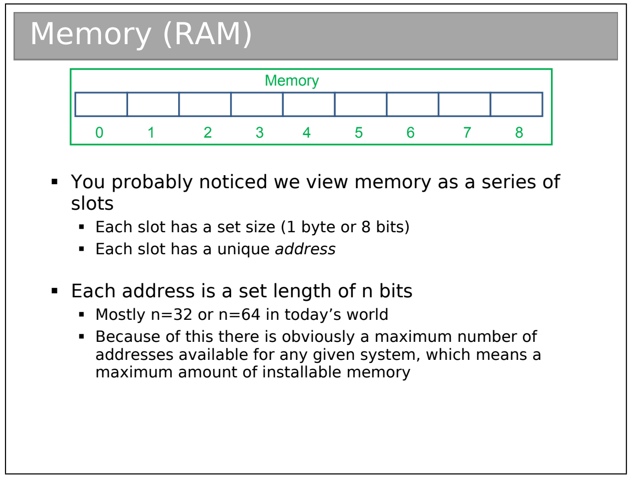


Dumb Model of a Computer



Dumb Model of a Computer

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



Memory (RAM)

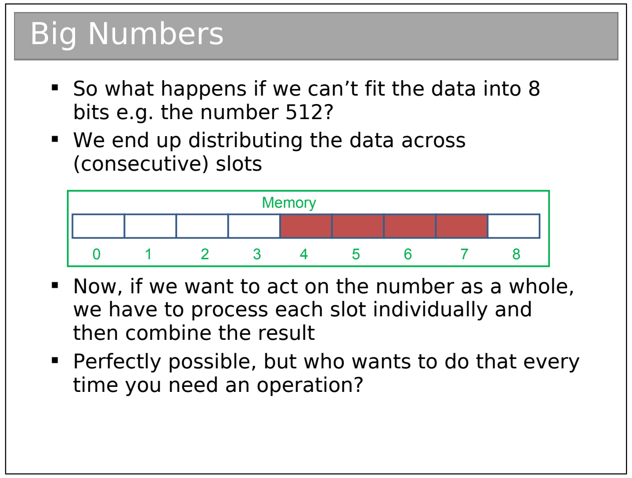
Memory
0 1 2 3 4 5 6 7 8

- You probably noticed we view memory as a series of slots
  - Each slot has a set size (1 byte or 8 bits)
  - Each slot has a unique *address*
- Each address is a set length of n bits
  - Mostly n=32 or n=64 in today's world
  - Because of this there is obviously a maximum number of addresses available for any given system, which means a maximum amount of installable memory

We slice system memory up into 8-bit (1 byte) chunks and give each one an *address* (i.e. a slot number).

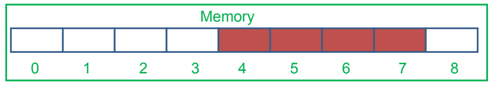
Each of the registers is a small, fixed size. On today's machines, they are usually just 32 or 64 bits. Since we have to be able to squeeze a memory address into a single register, the size of the registers dictates how much of the system memory we can use.

Q1. How much system memory can we use if we have 8, 32 or 64 bit registers?



Big Numbers

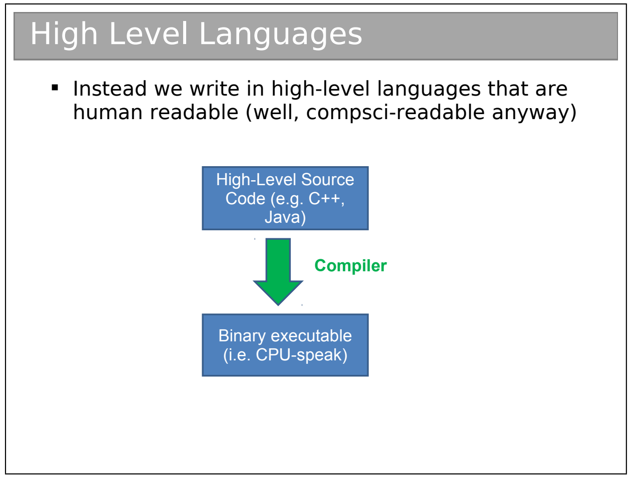
- So what happens if we can't fit the data into 8 bits e.g. the number 512?
- We end up distributing the data across (consecutive) slots



- Now, if we want to act on the number as a whole, we have to process each slot individually and then combine the result
- Perfectly possible, but who wants to do that every time you need an operation?

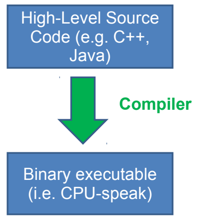
If I have two lots of 128-bits of data, how do I add them together on a 32 bit machine? I can't squeeze 128 bits into 32 bits. I have to chop the 128 bits up into 32 bit chunks, add corresponding chunks together and then put those new chunks back together again to get a 128-bit answer.

Can you imagine having to do this every time you needed to do an addition? Nasty.



High Level Languages

- Instead we write in high-level languages that are human readable (well, compsci-readable anyway)


High-Level Source Code (e.g. C++, Java)
Compiler
Binary executable (i.e. CPU-speak)

You already know the solution from FoCS and ML. We build up layers of abstractions and write tools to drill down from one layer to the next.

We program in high level code and leave it to the *compiler* to figure out that when we say

```
c=a+b;
```

we actually mean "chop up a and b, load them into registers, add them, look out for overflow, then put all the pieces back together in memory somewhere called c".

1.2 Functional and Imperative Revisited

We said imperative programming developed from consideration of the hardware. Given the hardware we've just designed, the logical thing to do is to represent data using *state*. i.e. explicit chunks of memory used to store data values. A program can then be viewed as running a series of commands that alter ('*mutate*') that state (hopefully in a known way) to give the final result.

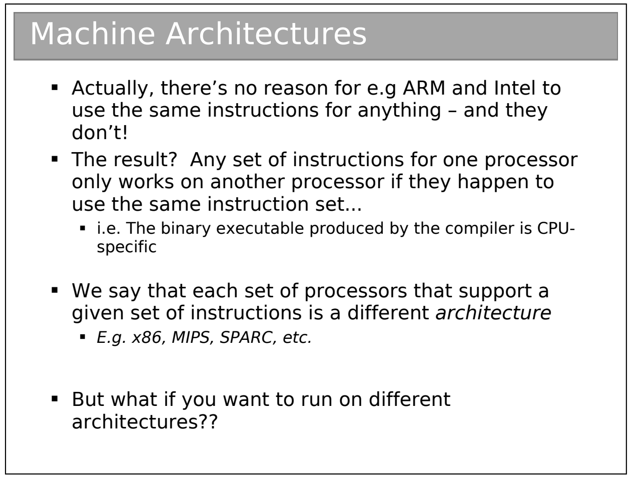
So `c=a+b` is referring to three different chunks of memory that represent numbers. Unlike in ML, it is perfectly valid to say `a=a+b` in an imperative language, which computes the sum and then overwrites the state stored in `a`.

Functional programs, on the other hand, can be viewed as independent of the machine. Strictly speaking, this means they can't have state in the same way since there's simply nowhere to put it (no notion of memory). Instead of starting with some data that gets *mutated* to give a result, we only have an input and an output for each function.

Of course, ML runs on real machines and therefore makes use of real hardware components such as memory. But that's because someone has mapped the ideal mathematical processes in functional code to machine operations (this must be possible because I said both the mathematical view and the system view are equivalent). The key point is that, although functional code runs on real hardware, it is independent of it since it is really a mathematical construct.

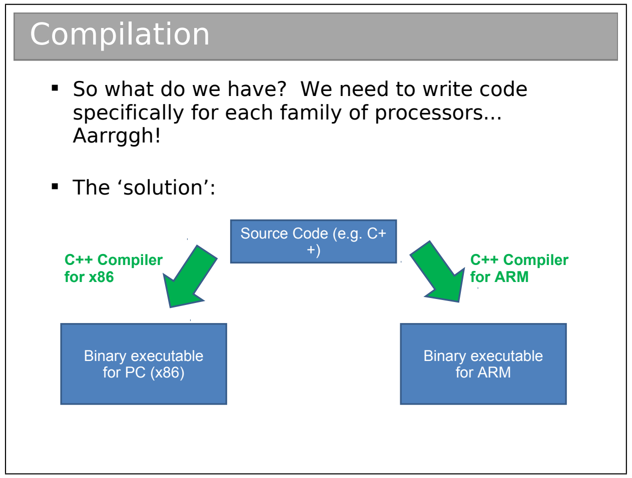
Most people find the imperative approach easier to understand and, in part, that is why it has gained such a stronghold within modern programming languages.

1.3 Machine Architectures



Machine Architectures

- Actually, there's no reason for e.g ARM and Intel to use the same instructions for anything – and they don't!
- The result? Any set of instructions for one processor only works on another processor if they happen to use the same instruction set...
  - i.e. The binary executable produced by the compiler is CPU-specific
- We say that each set of processors that support a given set of instructions is a different *architecture*
  - *E.g. x86, MIPS, SPARC, etc.*
- But what if you want to run on different architectures??



Compilation

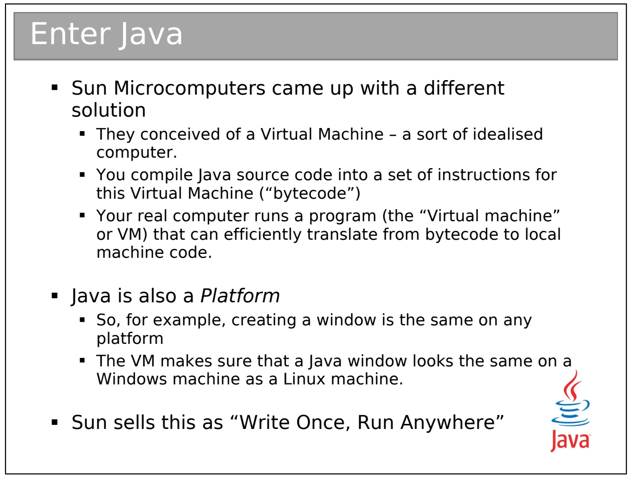
- So what do we have? We need to write code specifically for each family of processors... Aarrggh!
- The 'solution':

Remember those weird commands I made up when we did the fetch-execute cycle? Well, not all CPUs are equivalent: newer ones have cool new instructions to do things better. All that means is that a set of instructions for CPU X won't mean anything to CPU Y unless the manufacturers have agreed they will.

Q2. Why do you think CPU manufacturers haven't all agreed on a single set of instructions?

There is one very popular set of machine instructions known as x86. Basically, these were what IBM came up with for the original PC. Now the term 'PC' has come to mean a machine that supports at least the set of x86 instructions. Manufacturers like Apple traditionally chose not to do so, meaning that Mac applications couldn't run on PC hardware and vice-versa.

Q3. Apple recently started using Intel processors that support x86 instructions. This means Apple machines can now run Microsoft Windows. However, off-the-shelf PC software (which is compiled for x86) does not run on a Mac that is using the Apple operating system compiled for the Intel processor. Why not?



Enter Java

- Sun Microcomputers came up with a different solution
  - They conceived of a Virtual Machine – a sort of idealised computer.
  - You compile Java source code into a set of instructions for this Virtual Machine ("bytecode")
  - Your real computer runs a program (the "Virtual machine" or VM) that can efficiently translate from bytecode to local machine code.
- Java is also a *Platform*
  - So, for example, creating a window is the same on any platform
  - The VM makes sure that a Java window looks the same on a Windows machine as a Linux machine.
- Sun sells this as "Write Once, Run Anywhere"

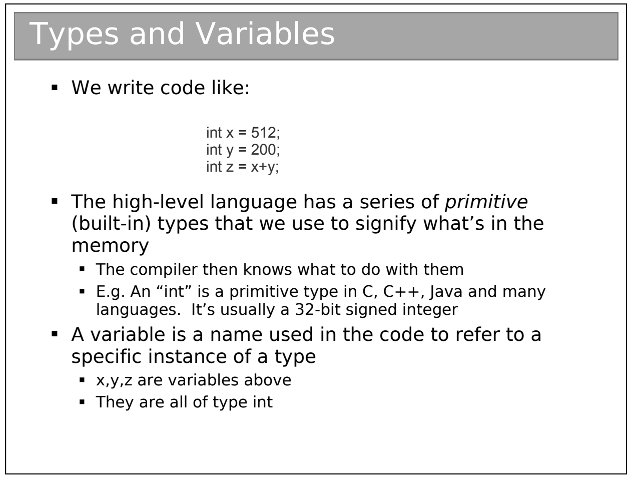
Java compiles high-level source code not into CPU-specific instructions but into a generic CPU instruction set called **bytecode**. Initially there were no machines capable of directly using bytecode, but recently they have started to appear in niche areas. For the most part, however, you use a **virtual machine**. This is a piece of software that has been compiled specifically for your architecture (like most off-the-shelf software) and which acts an an **interpreter**, converting bytecode instructions into meaningful instructions for your local CPU.

All the CPU-specific code goes into the virtual machine, which you should just think of as a piece of intermediary software that translates commands so the local hardware can understand.

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

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

1.4 Types

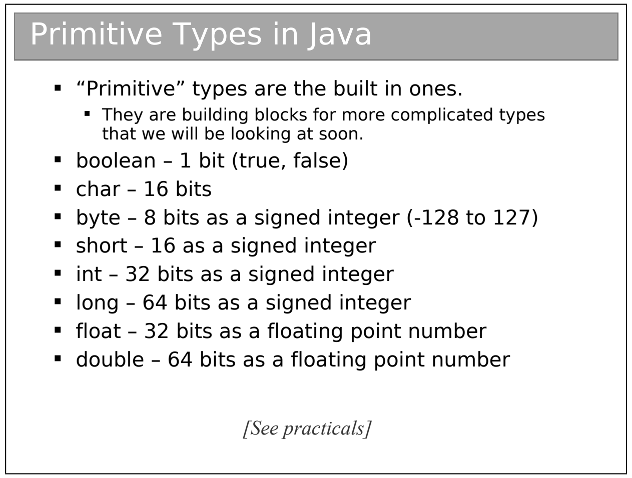


Types and Variables

- We write code like:

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

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



Primitive Types in Java

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

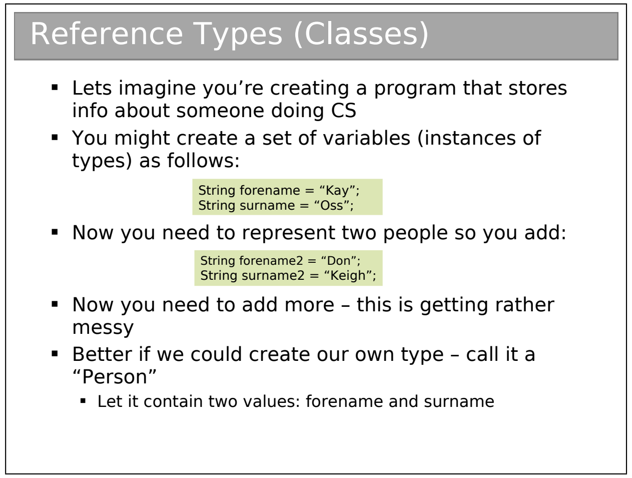
[See practicals]

You met the notion of types in FoCS and within ML. They're important because they allow the compiler to keep track of what the data in its memory actually means and stop us from doing dumb things like interpreting a floating point number as an integer.

In ML, you know you can specify the types of variables. But you almost never do: you usually let the compiler infer the type unless there's some ambiguity (e.g. between int and double). In an imperative language such as Java you *must* explicitly assign the types to every variable.

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

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



Reference Types (Classes)

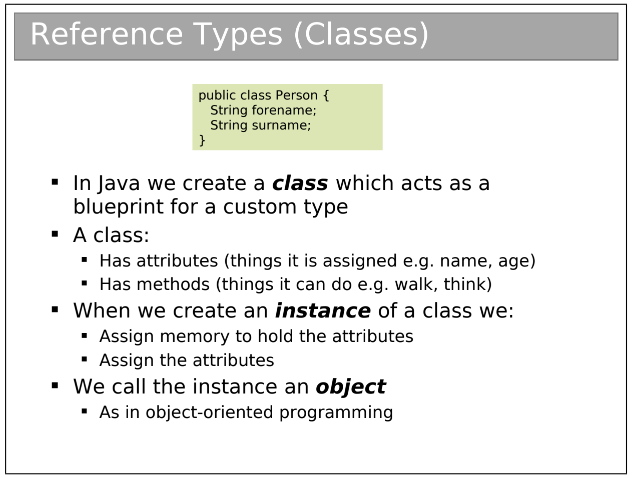
- Lets imagine you're creating a program that stores info about someone doing CS
- You might create a set of variables (instances of types) as follows:

```
String forename = "Kay";
String surname = "Oss";
```

- Now you need to represent two people so you add:

```
String forename2 = "Don";
String surname2 = "Keigh";
```

- Now you need to add more – this is getting rather messy
- Better if we could create our own type – call it a "Person"
  - Let it contain two values: forename and surname



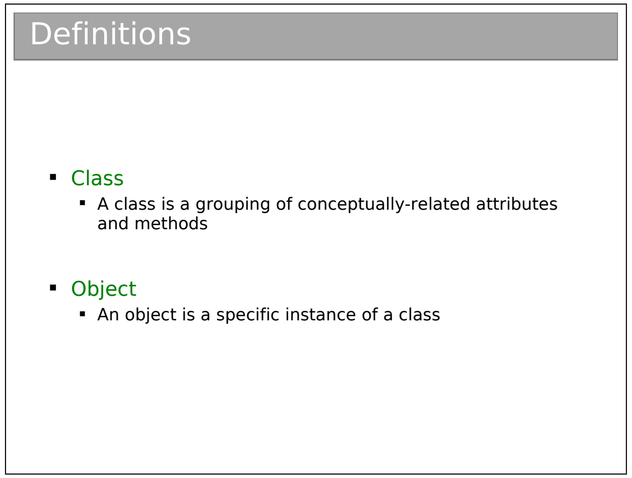
Reference Types (Classes)

```
public class Person {
   String forename;
   String surname;
}
```

- In Java we create a ***class*** which acts as a blueprint for a custom type
- A class:
  - Has attributes (things it is assigned e.g. name, age)
  - Has methods (things it can do e.g. walk, think)
- When we create an ***instance*** of a class we:
  - Assign memory to hold the attributes
  - Assign the attributes
- We call the instance an ***object***
  - As in object-oriented programming

You've met things called 'reference types' in ML so hopefully the idea of a reference isn't completely new to you. But if you're wondering

why classes are called reference types in Java, we'll come to that soon.



Definitions

- Class
  - A class is a grouping of conceptually-related attributes and methods
- Object
  - An object is a specific instance of a class

Once we have compiled our Java source code, we end up with a set of .class files. We can then distribute these files without their source code (.java) counterparts.

In addition to javac you will also find a javap program which allows you to poke inside a class file. For example, you can disassemble a class file to see the raw bytecode using javap -c classfile:

Input:

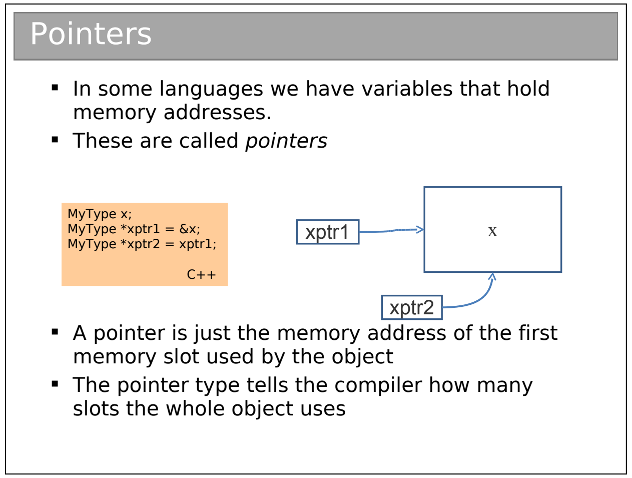
```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

javap output:

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

public static void main(java.lang.String[]);
  Code:
   0: getstatic #2; //Field java/lang/System.out:
                    //Ljava/io/PrintStream;
   3: ldc #3; //String Hello World
   5: invokevirtual #4; //Method java/io/PrintStream.println:
                        //(Ljava/lang/String;)V
   8: return

}
```

This probably won't make a lot of sense to you right now: that's OK. Just be aware that we can view the bytecode and that sometimes this can be a useful way to figure out *exactly* what the JVM will do with a bit of code. You aren't expected to know bytecode.

1.5 Pointers and References



Pointers

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

```
MyType x;
MyType *xptr1 = &x;
MyType *xptr2 = xptr1;

                    C++
```

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

A pointer is just the name given to memory addresses when we handle them in code. They are very powerful concepts to work with, but you need to take a lot of care.

Java doesn't expose pointers to the programmer, so we'll use some basic C examples to make some points in the lectures. There are just a few bits of C you may need to know to follow:

- `int x` declares a variable `x` that is a 32-bit signed integer
- `int *p` declares a pointer `p` that can point to an `int` in memory. It must be manually initialised to point somewhere useful.
- `p=&x` Gets the memory address for the start of variable `x` and stick it in `p` so `p` is a pointer to `x`.
- `int a=*p` declares a new `int` variable called `a` and sets its value to that pointed to by pointer `p`. We say that we are *dereferencing*

p.

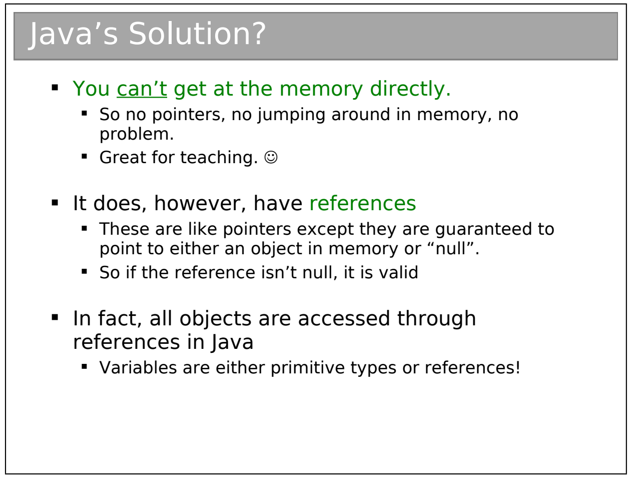
If it helps, we can map these operations to ML something like[2]:

C	ML	Comments
int x	let val xm=0 in...	xm is immutable
int *p	let val pm=ref 0 in...	pm is a reference to a variable initially set to zero
p=&x	val pm = ref xm	
int a=*p	val am = !pm	

The key point to take away is that a pointer is just a number that maps to a particular byte in memory. You can set that number to anything you like, which means there's no guarantee that it points to anything sensible..! And the computer just obeys — if you say that a pointer points to an int, then it doesn't argue.

Working with pointers helps us to avoid needless copies of data and usually speeds up our algorithms (see next term's Algorithms I course). But a significant proportion of program crashes are caused by trying to read or write memory that hasn't been allocated or initialised...

[2]Now, if all that reference type stuff in ML confused you last term, don't worry about making these mappings just now. Just be aware that what will come naturally in imperative programming has been retrofitted to ML, where it seems a little less obvious! For now, just concentrate on understanding pointers and references from scratch.



Java's Solution?

- You can't get at the memory directly.
  - So no pointers, no jumping around in memory, no problem.
  - Great for teaching. ☺
- It does, however, have references
  - These are like pointers except they are guaranteed to point to either an object in memory or "null".
  - So if the reference isn't null, it is valid
- In fact, all objects are accessed through references in Java
  - Variables are either primitive types or references!

Remember that classes are called *Reference Types* in Java. This is because they are *only* handled by reference. Code like:

```
MyClass m = new MyClass();
```

creates an instance of MyClass somewhere in main memory (that's what goes on on the right of the equals). And then it creates a *reference* to that instance and stores the result in a variable m.

A reference is really just a special kind of pointer. Remember that a pointer is just a number (representing a memory slot). We can set a pointer to any number we like and so it is possible that dereferencing it gets us garbage.

A reference, however, *always* has a valid address *or* possibly a special value to signify it is invalid. If you dereference a reference without the special value in it, the memory you read *will* contain something sensible.

In Java, the 'special value' is signified using the keyword `null`:

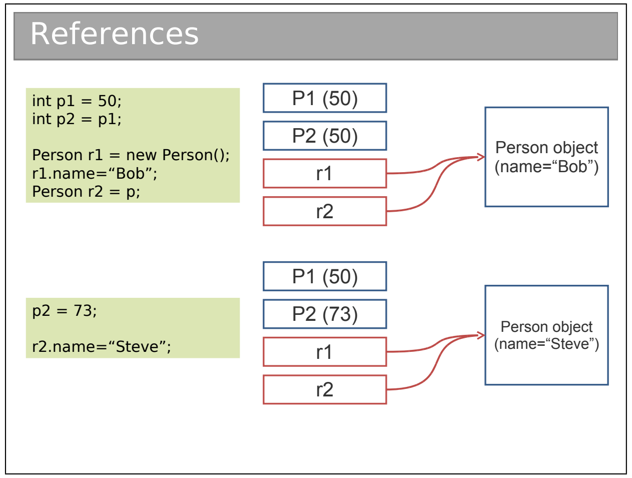
```
MyClass m = null;   // Not initialised
...
m = new MyClass();
```

If you try to do something with a reference set to `null`, Java will bail out on you. To test whether a reference is `null`:[3]

```
if (m==null) {
   ...
}
```

Q4. Pointers are problematic because they might not point to anything useful. A `null` reference doesn't point to anything useful. So what is the advantage of using references over pointers?

[3]Note that the Java/C syntax uses = for assignment and == for comparison. We'll discuss this in more detail later.



References

```
int p1 = 50;
int p2 = p1;

Person r1 = new Person();
r1.name="Bob";
Person r2 = p;
```

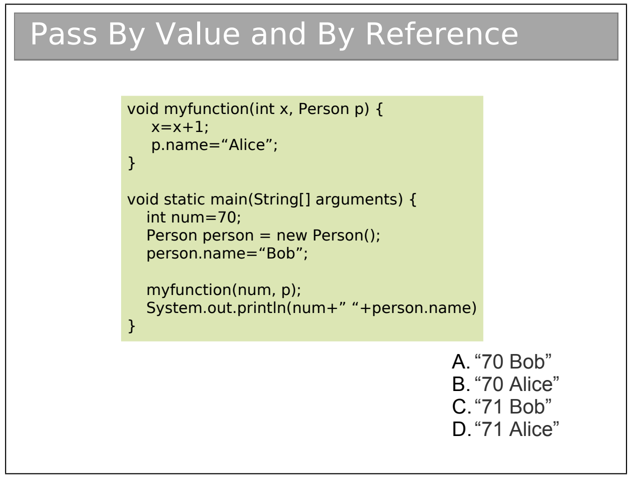
```
p2 = 73;

r2.name="Steve";
```

So we have *primitive* types and *reference* types. A variable for a primitive type can be thought of as being an instance of a type, so each you assign something to it, the same bit of memory gets changed as with P1 and P2 in the above slide.

A variable for a reference type holds the memory address. If you assign something to it, what it used to be assigned to still exists in memory (but your variable doesn't link to it any more). Multiple variables can reference the same object in memory (like r1 and r2 above).

Q5. Draw box diagrams like the one above to illustrate what happens in each step of the following Java code in memory:

```
Person p = null;
Person p2 = new Person();
p = p2;
p2 = new Person();
p=null;
```



Pass By Value and By Reference

```
void myfunction(int x, Person p) {
    x=x+1;
    p.name="Alice";
}

void static main(String[] arguments) {
   int num=70;
   Person person = new Person();
   person.name="Bob";

   myfunction(num, p);
   System.out.println(num+" "+person.name)
}
```

A. "70 Bob"
B. "70 Alice"
C. "71 Bob"
D. "71 Alice"

Now things are getting a bit more involved. We have added a function *prototype* into the mix: `void myfunction(int x, Person p)`. This means it returns nothing (codenamed `void`) but takes an `int` and a `Person` object as input.

When the function is run, the computer takes a copy of the inputs and creates local variables `x` and `p` to hold the copies.

Here's the thing: if you copy a primitive type, the value gets copied across (so if we were watching memory we would see two integers in memory, the second with the same value as the first).

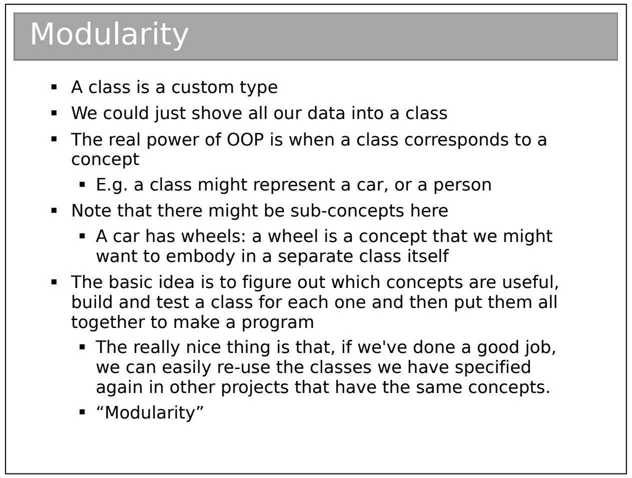
But for the reference type, it is the *reference* that gets copied and not the object it points to (i.e. we just create a new reference `p` and point it at the same object).

When we copy a primitive value we say that it is *pass by value*. When we send a reference to the object rather than the object, it is *pass by reference*.

In my opinion, this all gets a bit confusing in Java because you have no choice but to use references (unlike in, say C++). So I prefer to think of all variables as representing either primitives or references (rather than primitives and actual objects), and when we pass any variable around its value gets copied. The 'value' of a primitive is its actual value, whilst the 'value' of a reference is a memory address. You may or may not like this way of thinking: that's fine. Just make *certain* you understand the different treatment of primitives and reference types.

Chapter 2

Object-Oriented Concepts



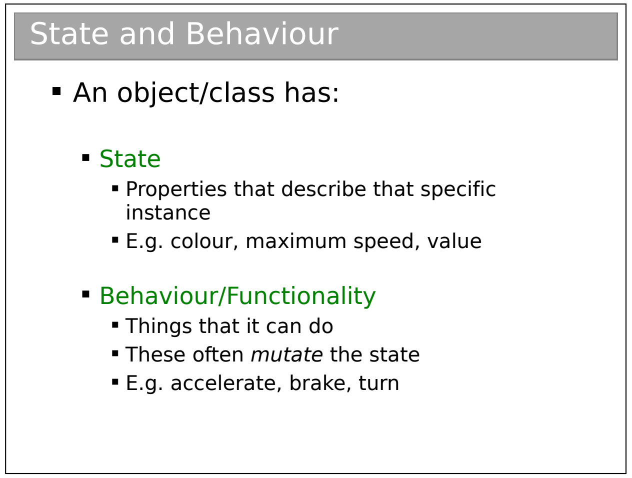
Modularity

- A class is a custom type
- We could just shove all our data into a class
- The real power of OOP is when a class corresponds to a concept
  - E.g. a class might represent a car, or a person
- Note that there might be sub-concepts here
  - A car has wheels: a wheel is a concept that we might want to embody in a separate class itself
- The basic idea is to figure out which concepts are useful, build and test a class for each one and then put them all together to make a program
  - The really nice thing is that, if we've done a good job, we can easily re-use the classes we have specified again in other projects that have the same concepts.
  - “Modularity”

Modularity is extremely important in OOP. It's the usual CS trick: break big problems down into chunks and solve each chunk. In this case, we have large programs, meaning scope for lots of coding bugs. We split the program into modules. The goal is:

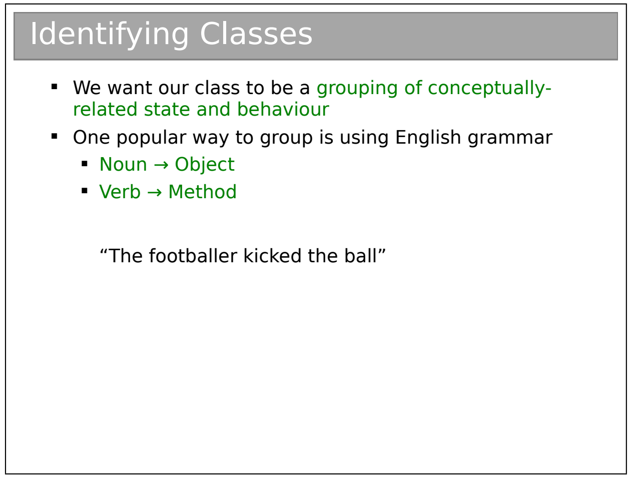
- Modules are conceptually easier to handle
- Different people can work on different modules simultaneously
- Modules may be re-used in other software projects
- Modules can be individually tested as we go (unit testing)

The module 'unit' in OOP is a class.



State and Behaviour

- An object/class has:
  - State
    - Properties that describe that specific instance
    - E.g. colour, maximum speed, value
  - Behaviour/Functionality
    - Things that it can do
    - These often *mutate* the state
    - E.g. accelerate, brake, turn

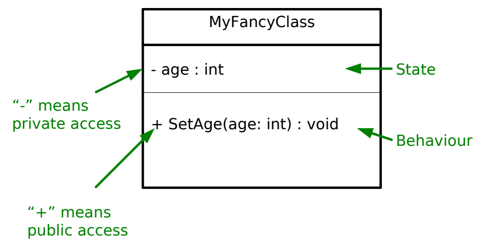


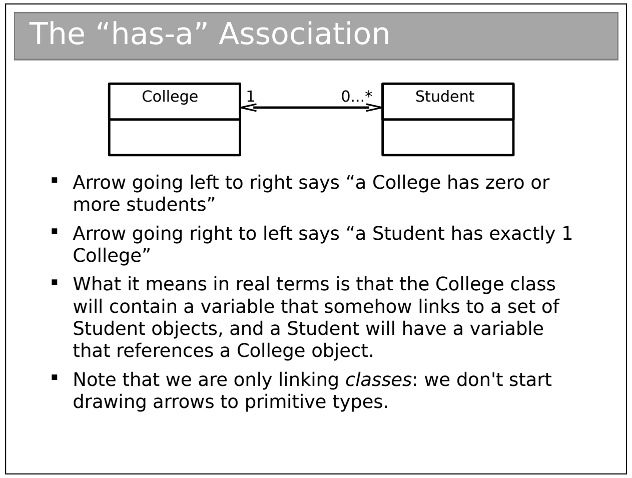
Identifying Classes

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

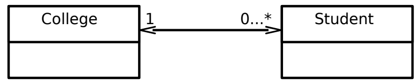
"The footballer kicked the ball"

Representing a Class Graphically (UML)





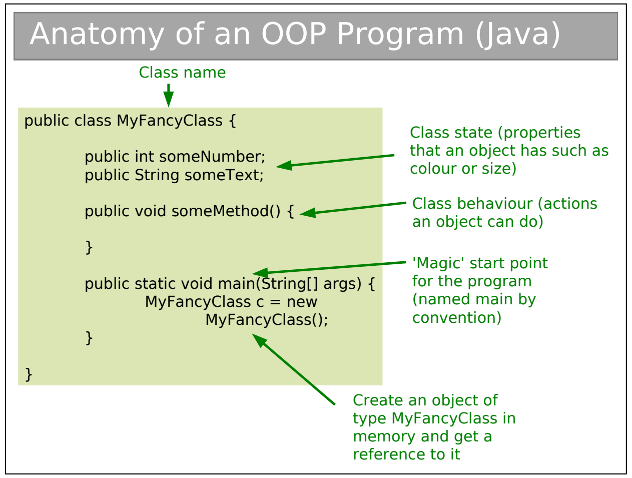
The “has-a” Association



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

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

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



Anatomy of an OOP Program (Java)

Class name

```
public class MyFancyClass {

          public int someNumber;
          public String someText;

          public void someMethod() {

          }

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

}
```

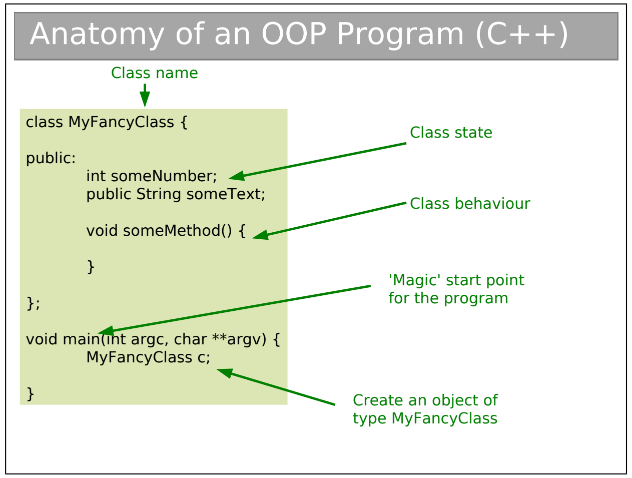
Class state (properties that an object has such as colour or size)

Class behaviour (actions an object can do)

'Magic' start point for the program (named main by convention)

Create an object of type MyFancyClass in memory and get a reference to it

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

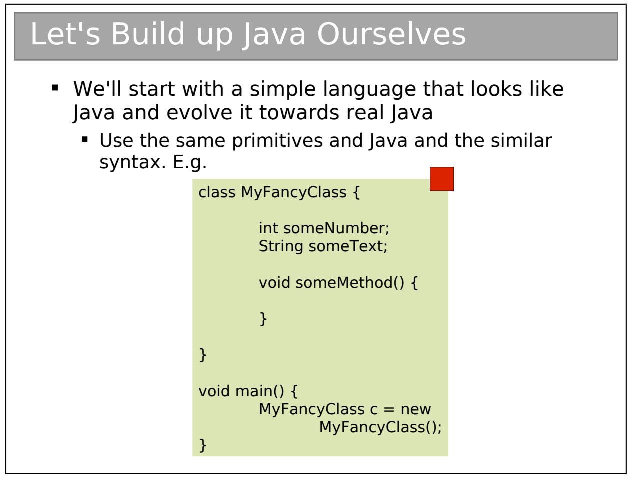


Anatomy of an OOP Program (C++)

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

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

}
```

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

This is here just so you can compare. The Java syntax is based on C/C++ so it's no surprise that there are a lot of similarities. This certainly eases the transition from Java to C++ (or vice-versa), but there are a lot of pitfalls to bear in mind (mostly related to memory management).



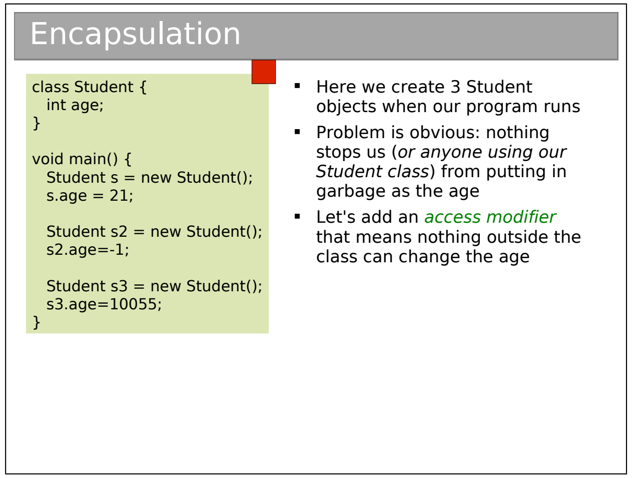
Let's Build up Java Ourselves

- We'll start with a simple language that looks like Java and evolve it towards real Java
  - Use the same primitives and Java and the similar syntax. E.g.

```
class MyFancyClass {  🟥

        int someNumber;
        String someText;

        void someMethod() {

        }

}

void main() {
        MyFancyClass c = new
                MyFancyClass();
}
```

A red box on the code box means it is not valid Java.

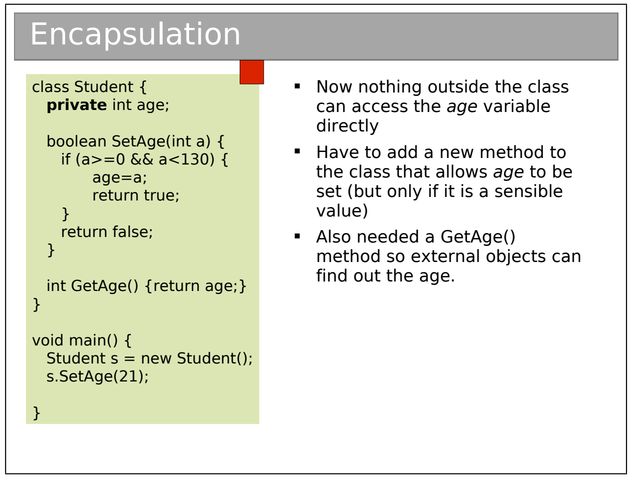
2.1 Encapsulation



Encapsulation

```
class Student {
   int age;
}

void main() {
   Student s = new Student();
   s.age = 21;

   Student s2 = new Student();
   s2.age=-1;

   Student s3 = new Student();
   s3.age=10055;
}
```

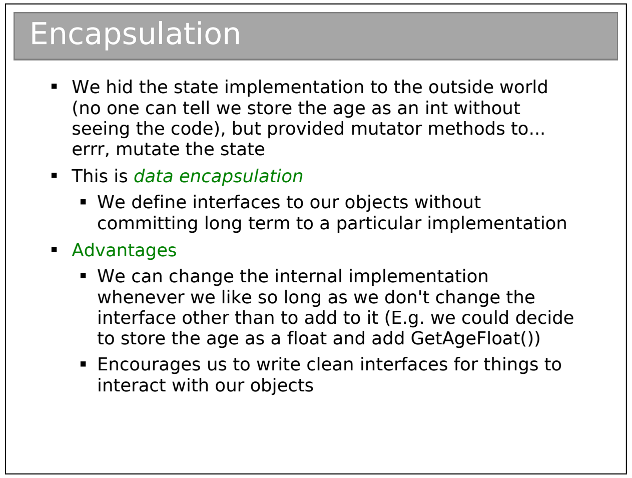
- Here we create 3 Student objects when our program runs
- Problem is obvious: nothing stops us (*or anyone using our Student class*) from putting in garbage as the age
- Let's add an *access modifier* that means nothing outside the class can change the age



Encapsulation

```
class Student {
   private int age;

   boolean SetAge(int a) {
      if (a>=0 && a<130) {
            age=a;
            return true;
      }
      return false;
   }

   int GetAge() {return age;}
}

void main() {
   Student s = new Student();
   s.SetAge(21);

}
```

- Now nothing outside the class can access the *age* variable directly
- Have to add a new method to the class that allows *age* to be set (but only if it is a sensible value)
- Also needed a GetAge() method so external objects can find out the age.



Encapsulation

- We hid the state implementation to the outside world (no one can tell we store the age as an int without seeing the code), but provided mutator methods to... errr, mutate the state
- This is *data encapsulation*
  - We define interfaces to our objects without committing long term to a particular implementation
- Advantages
  - We can change the internal implementation whenever we like so long as we don't change the interface other than to add to it (E.g. we could decide to store the age as a float and add GetAgeFloat())
  - Encourages us to write clean interfaces for things to interact with our objects

Another name for encapsulation is *information hiding*. The basic idea is that a class should expose a clean interface that allows interaction, but nothing about its internal state. So the general rule is that all state should start out as private and only have that access relaxed if there is a very, very good reason.

Encapsulation helps to minimise *coupling* between classes. High coupling between two class, A and B, implies that a change in A is likely to ripple through to B. In a large software project, you really don't want a change in one class to mean you have to go and fix up the other 200 classes! So we strive for low coupling.

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