Object Oriented Programming
Dr Robert Harle

IA NST CS and CST

Lent 2009/10

Handout 1

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

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
  - <u>Everyone</u> should attend <u>every</u> 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
  - <u>Lots</u> of good resources on the web

- Design Patterns
  - *Design Patterns* by Gamma et al.
  - <u>Lots</u> 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/

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...

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.

Bits, Bytes, Compilers and Languages

Dumb Model of a Computer
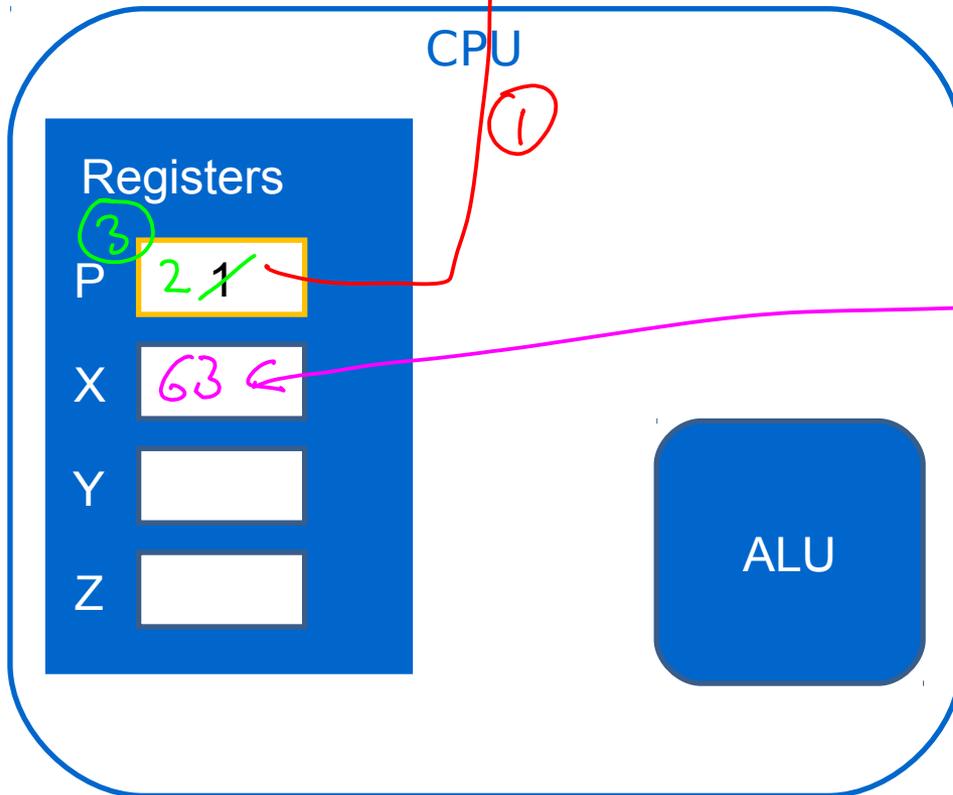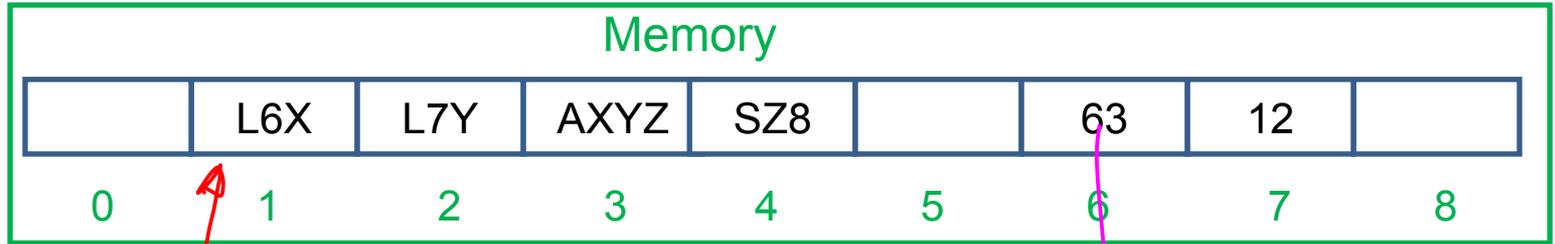
Dumb Model of a Computer
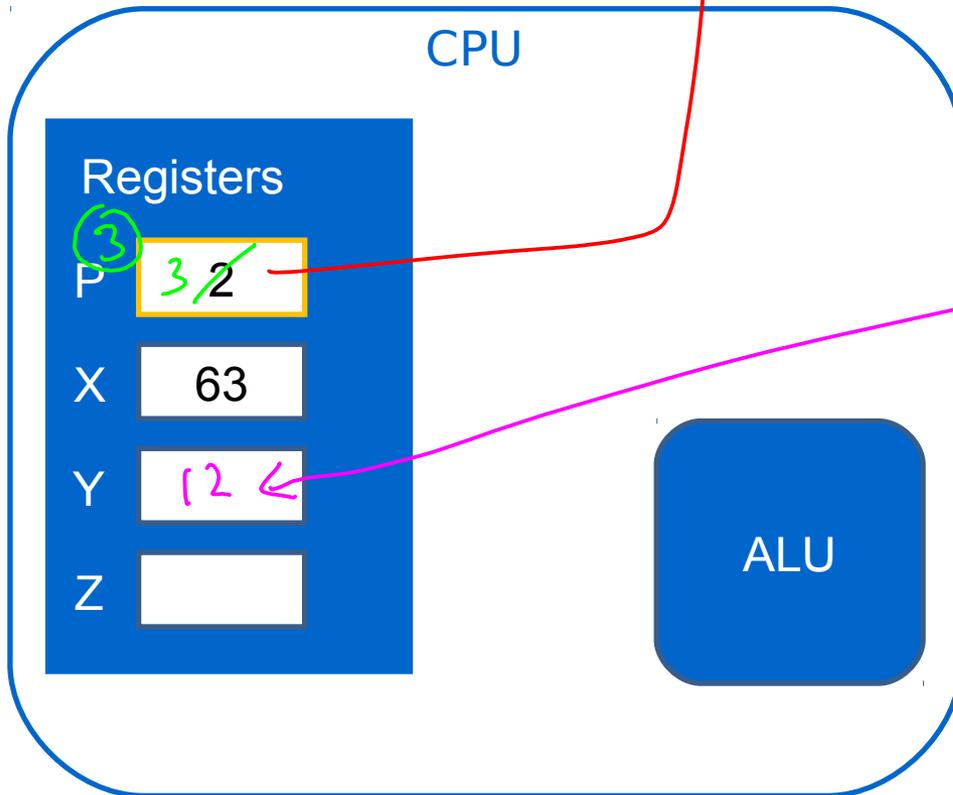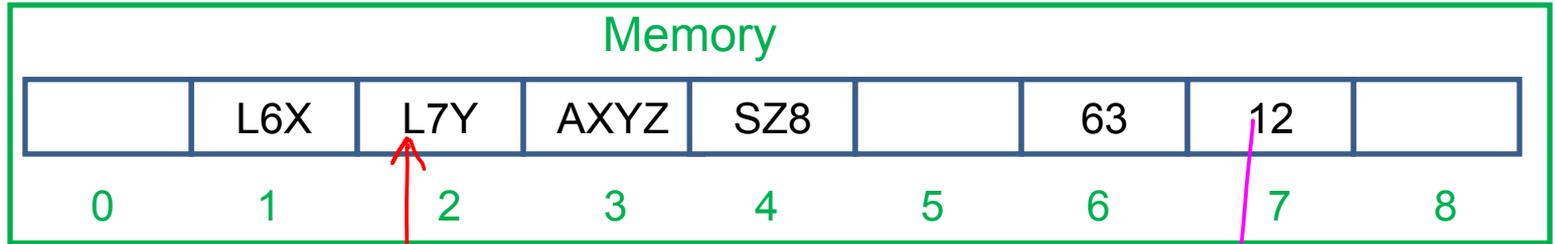
Dumb Model of a Computer
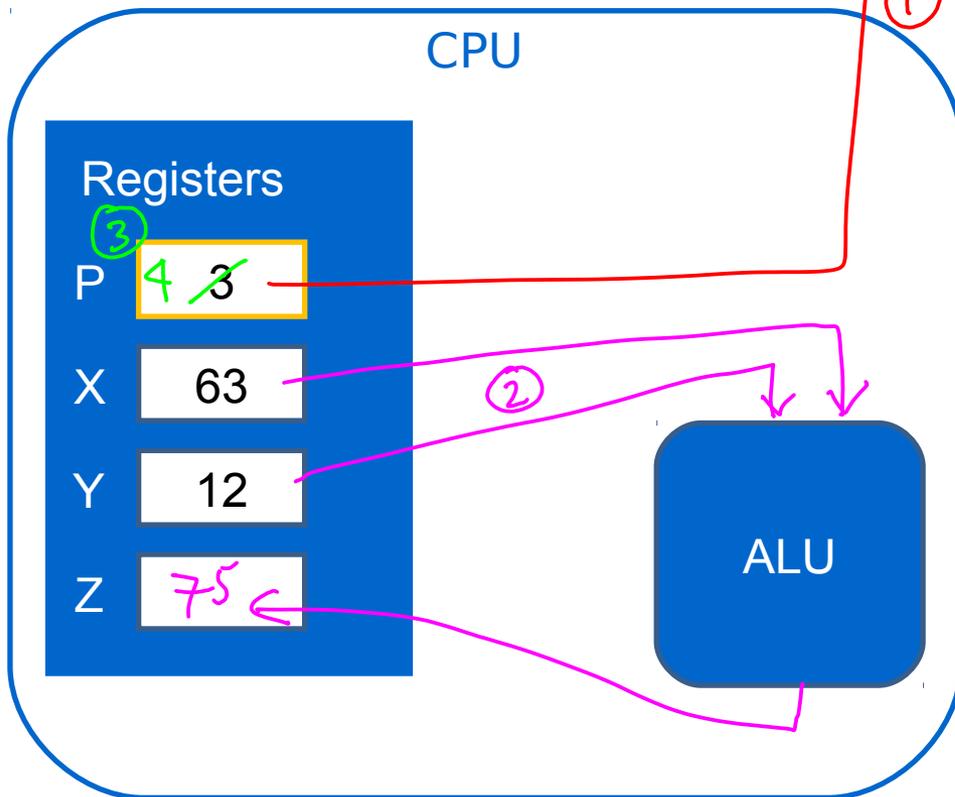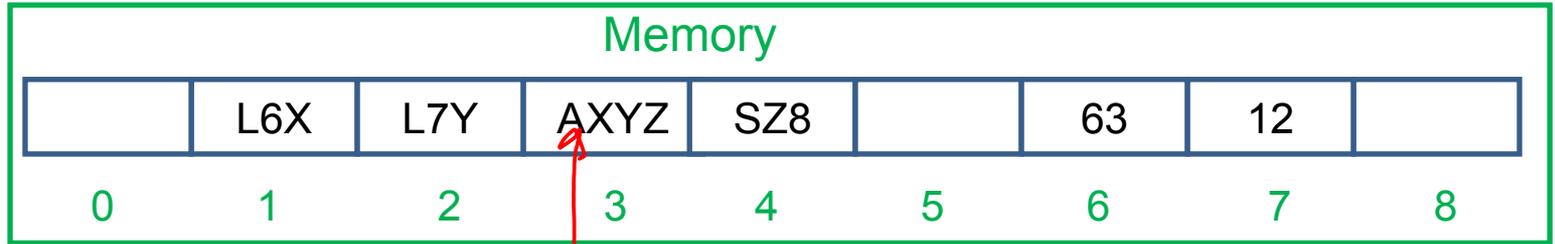
Dumb Model of a Computer
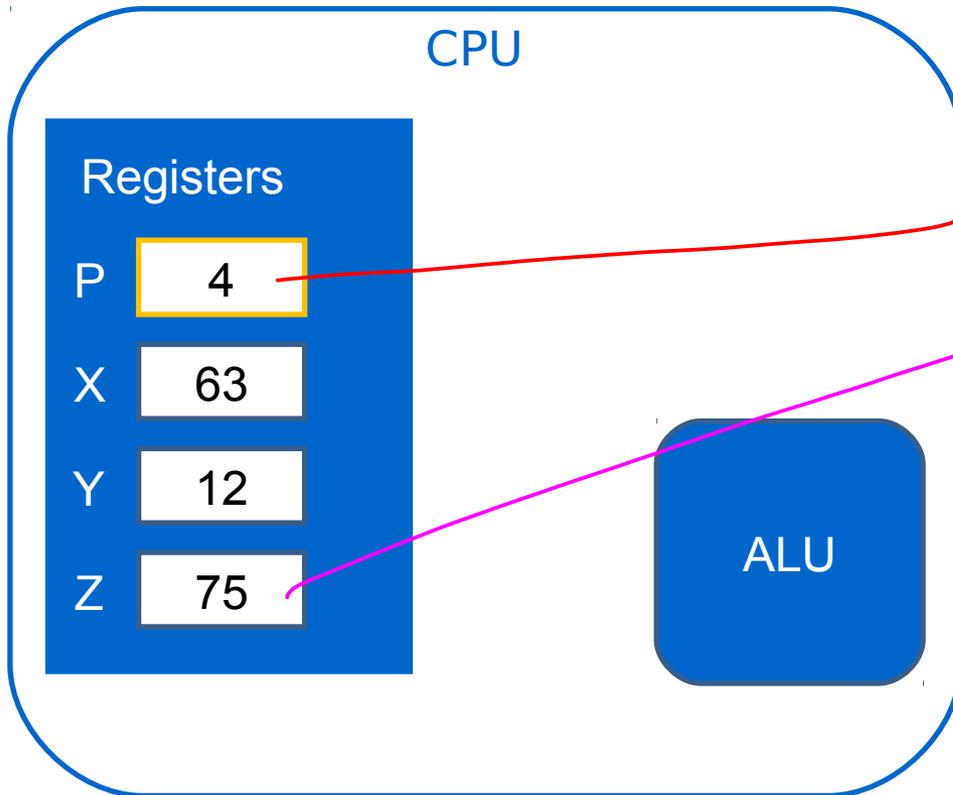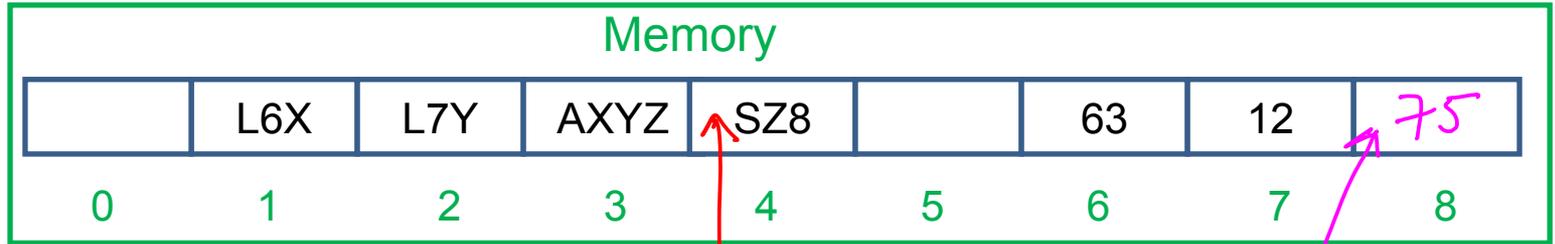
Memory (RAM)
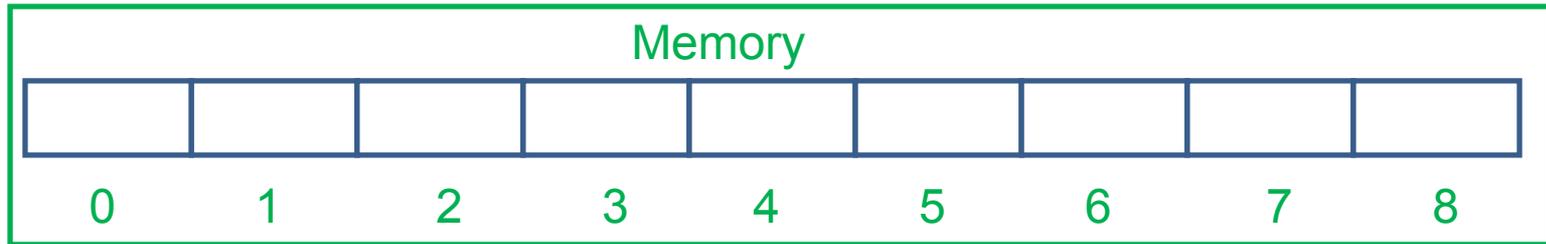
- You probably noticed we view memory as a series of slots
  - Each slot has a set size (1 byte or 8 bits)
  - Each slot has a unique *address*

- 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

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

Memory

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
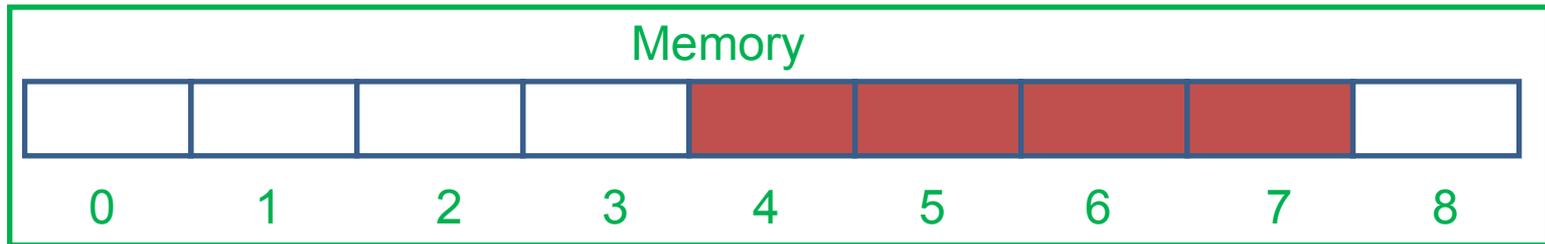
- Now, if we want to act on the number as a whole, we have to process each slot individually and then combine the result

- Perfectly possible, but who wants to do that every time you need an operation?

High Level Languages

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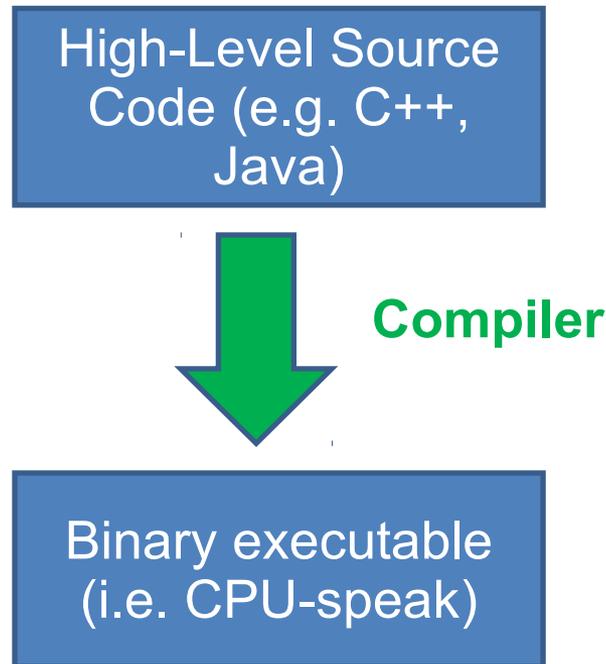
Machine Architectures

- 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':
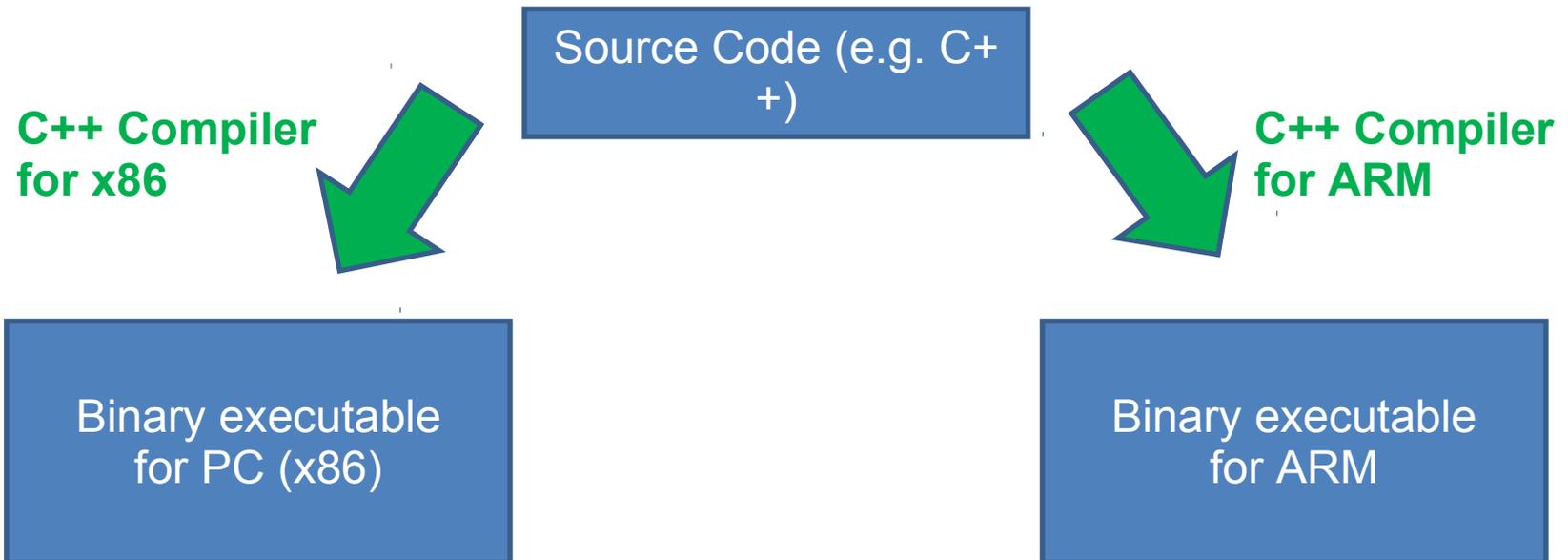
Enter Java

- 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"

compile

$C++$

Machine code

run

Virtual Machine translates
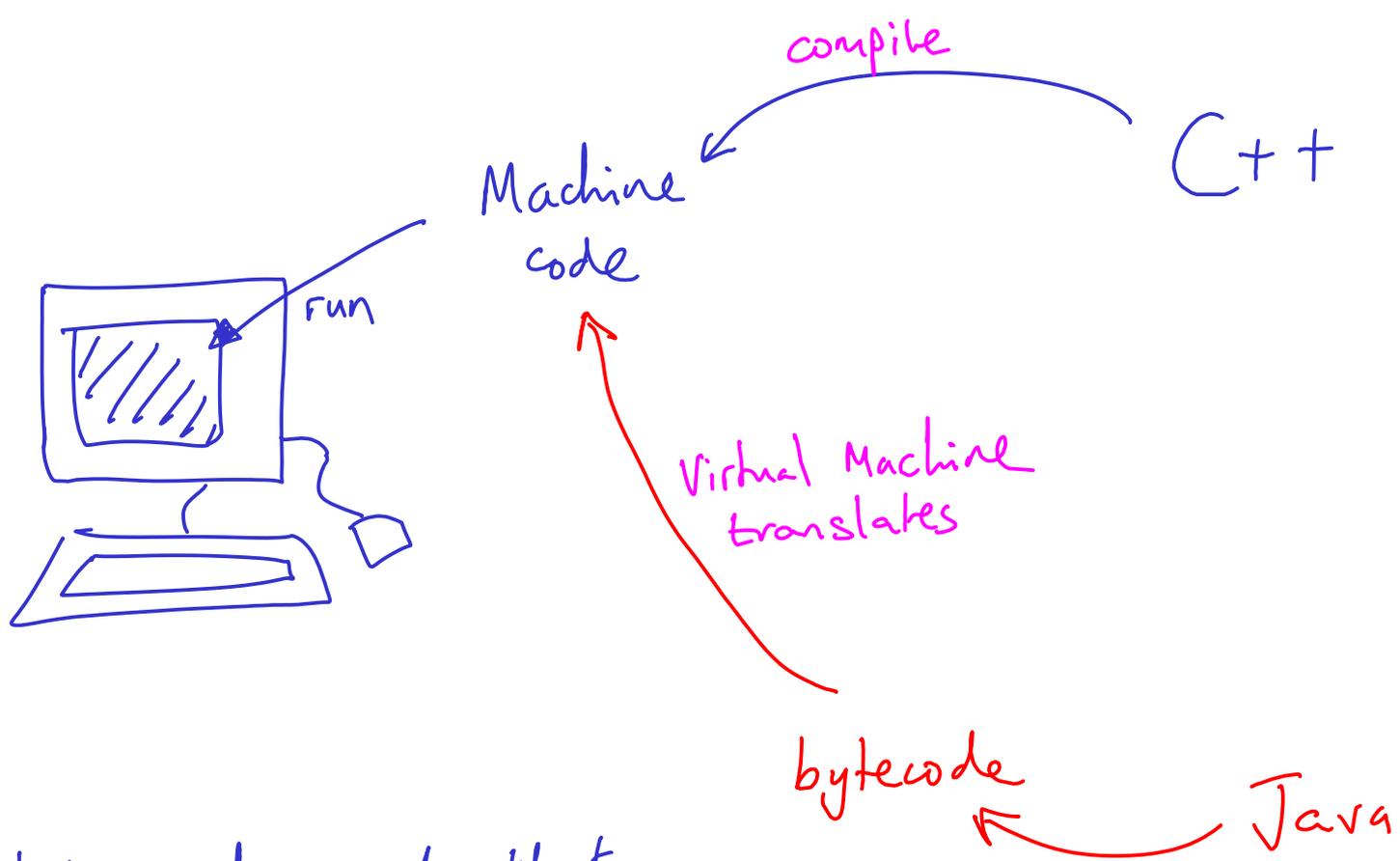
bytecode

Java

C++ : Distribute machine code that is specific to the architecture

Java : Distribute bytecode that runs on a VM on any architecture

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]

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

Definitions

- Class
  - A class is a grouping of conceptually-related attributes and methods

- Object
  - An object is a specific instance of a class

Practicalities

- So you've written a Java source file (MyProgram.java)
- First you need to compile it
  - javac MyProgram.java
  - This creates MyProgram.class, which is the bytecode version (i.e. A set of instructions that make sense to a virtual machine).
- Now we need to run a virtual machine to actually use it
  - java MyProgram
- Easy!

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++
```

xptr1

X

xptr2
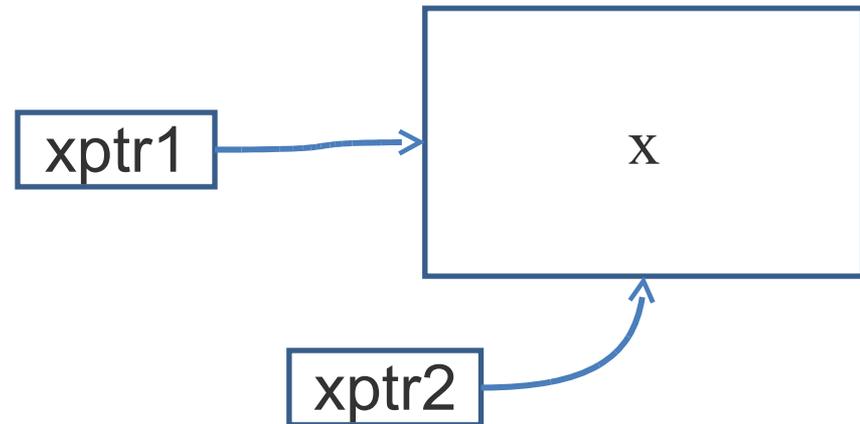
- A pointer is just the memory address of the first memory slot used by the object
- The pointer type tells the compiler how many slots the whole object uses

Java's Solution?

- You <u>can't</u> 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!

References

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

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

| P1 (50) |
| P2 (50) |
| r1 |
| r2 |

Person object
(name="Bob")

```
p2 = 73;

r2.name="Steve";
```

| P1 (50) |
| P2 (73) |
| r1 |
| r2 |

Person object
(name="Steve")
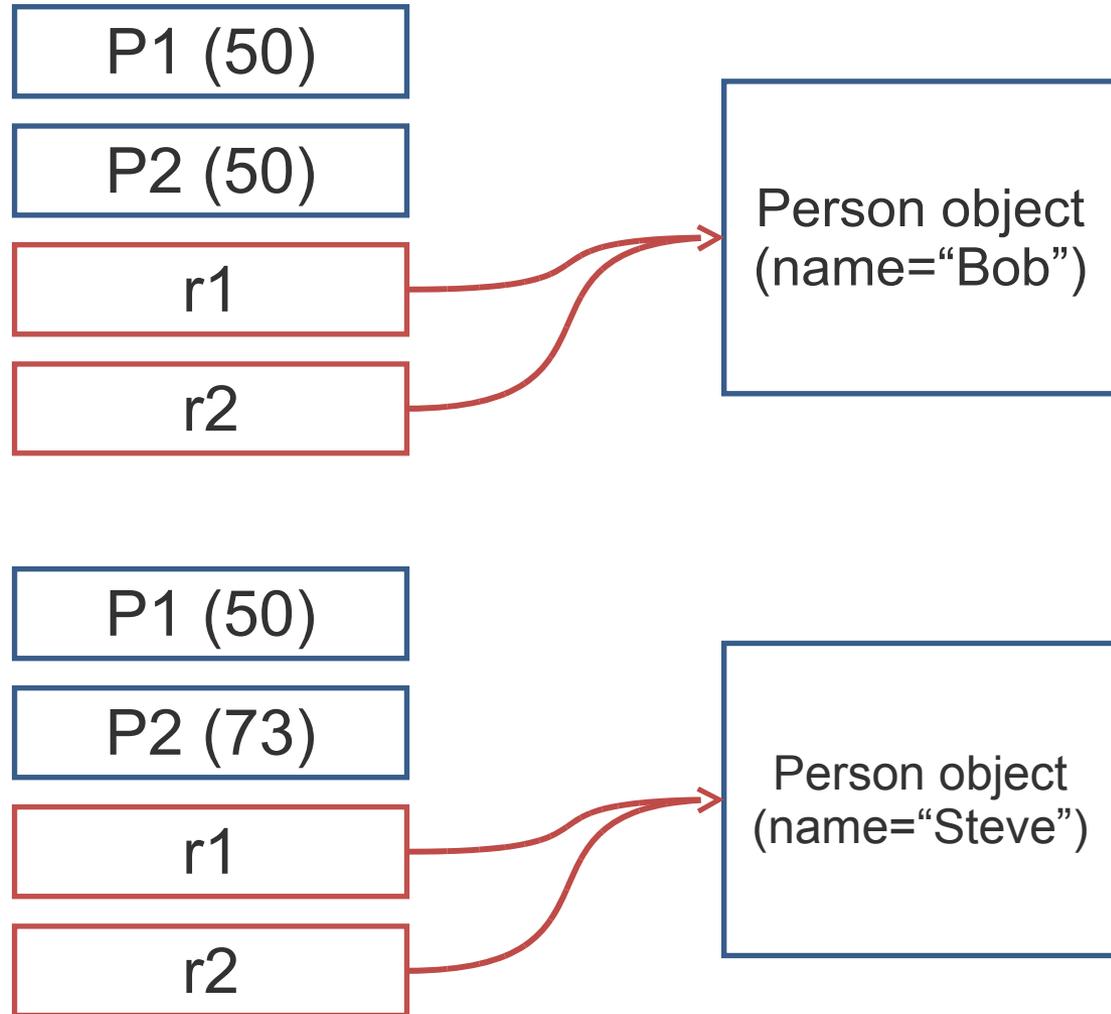
Pass By Value and By Reference

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

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

int num =70   Creates a primitive type called num and with value 70.

Person person   Creates a reference called person

new Person()   Creates a Person object in memory and returns a reference to it.

Person = new Person()

person ref → Person object

Before myfunction()

num = 70

person

Person object
"Bob"

Start of myfunction()

num = 70

person

x = 70

P

Person object
"Bob"

End of myfunction()

num = 70

person

x = 71

p

Person
"Alice
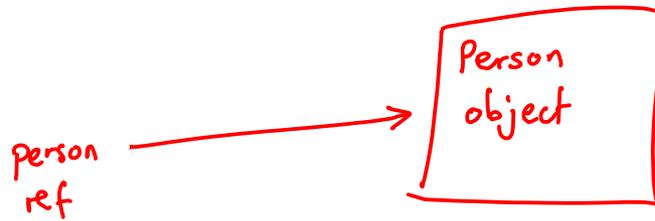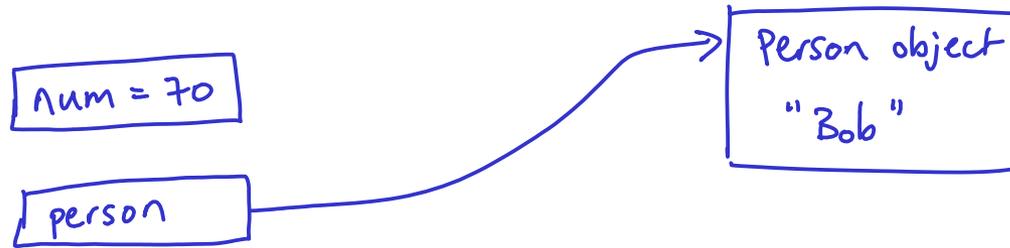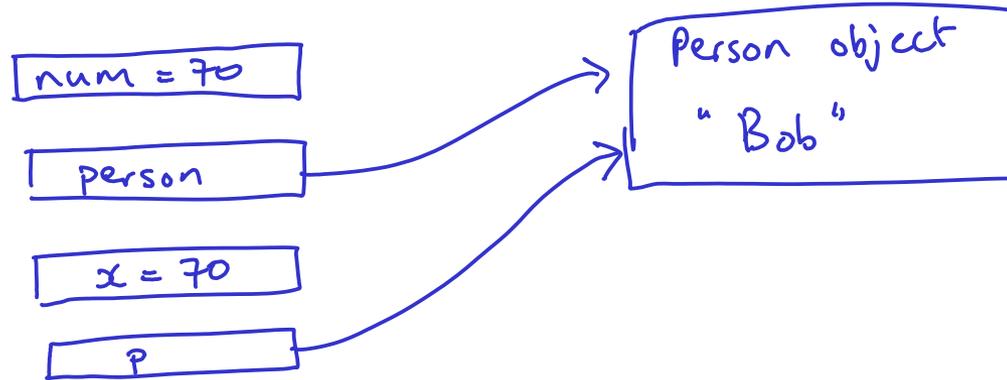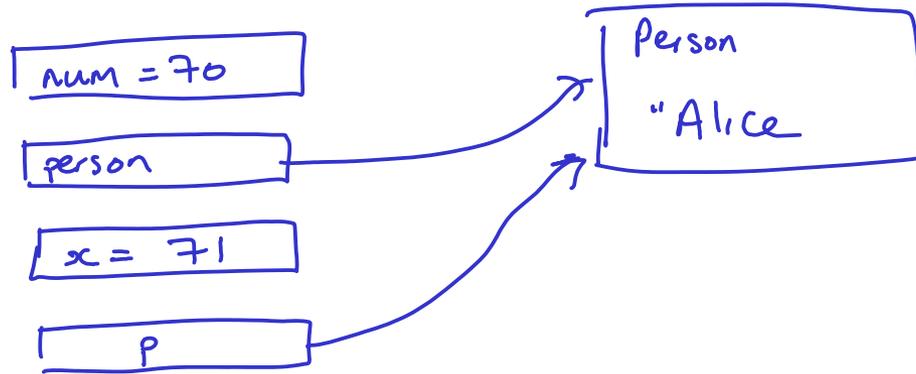
After myfunction()

x, p are deleted, leaving:

num = 70

person
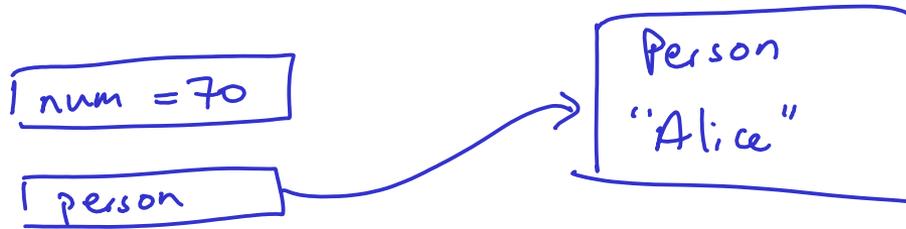
Person
"Alice"

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"

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)

```
┌──────────────────────────────┐
│        MyFancyClass          │
├──────────────────────────────┤
│                              │
│  - age : int                 │  ← State
│                              │
├──────────────────────────────┤
│                              │
│  + SetAge(age: int) : void   │  ← Behaviour
│                              │
│                              │
└──────────────────────────────┘
```

"-" means
private access

"+" means
public access

The "has-a" Association

College	1		0...*	Student

Correct arrow: →
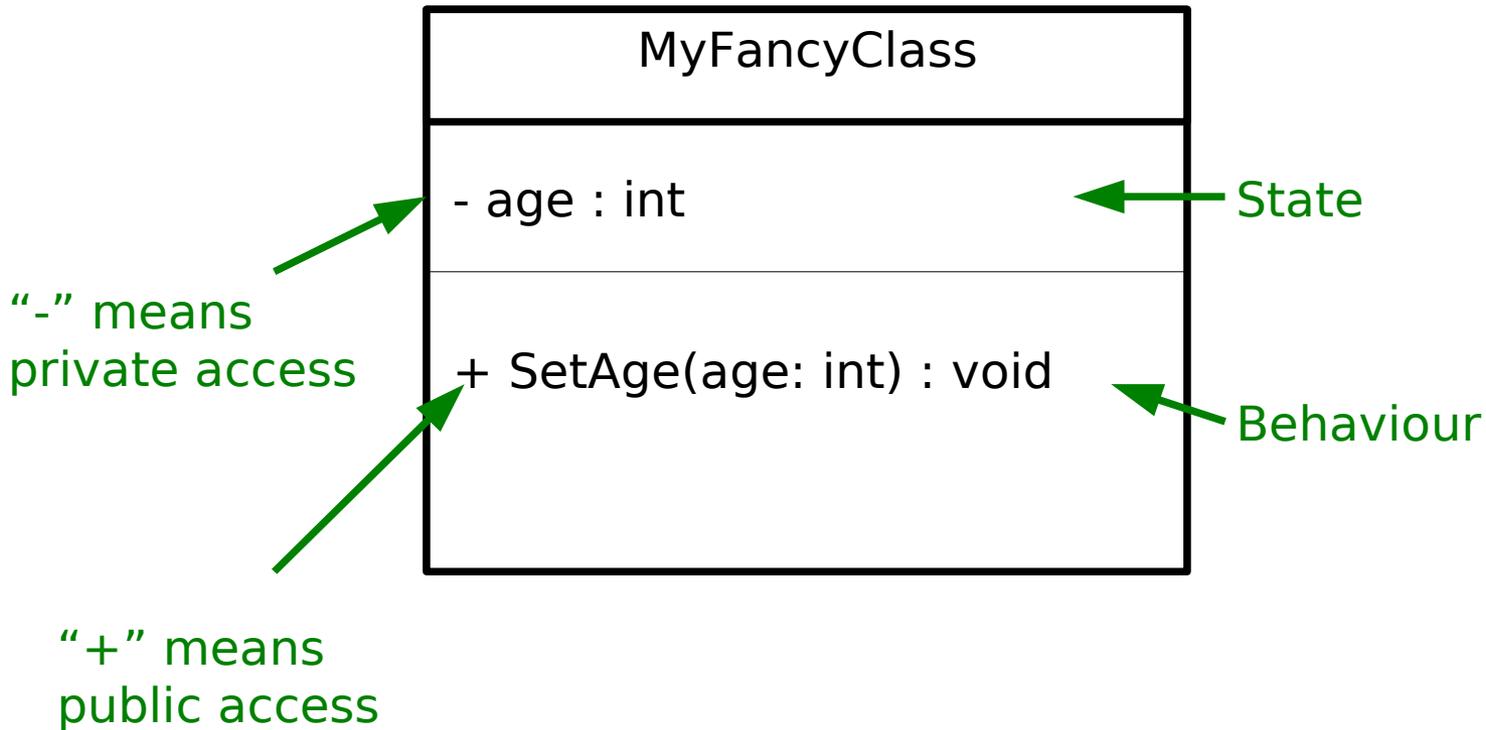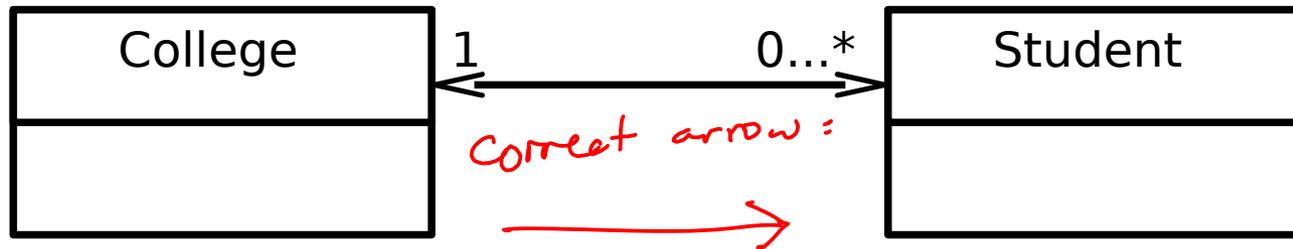
- Arrow going left to right says "a College has zero or more students"

- 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.

Anatomy of an OOP Program (Java)

Class name

```java
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

Anatomy of an OOP Program (C++)

Class name

```cpp
class MyFancyClass {

public:

        int someNumber;
        public String someText;

        void someMethod() {


        }

};

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

}
```

Class state

Class behaviour

'Magic' start point
for the program

Create an object of
type MyFancyClass

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();
}
```

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

+ Perform checks

Inheritance

```
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

```
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

"has-a"

Specialise

```
┌─────────────────────┐
│       Person        │
├─────────────────────┤
│                     │
│                     │
│                     │
│                     │
└─────────────────────┘
```

Object

"is-a"

```
┌─────────────────┐     ┌─────────────────┐
│     Student     │     │     Lecturer    │
├─────────────────┤     ├─────────────────┤
│                 │     │                 │
│                 │     │                 │
│                 │     │                 │
└─────────────────┘     └─────────────────┘
```

Also known as an "is-a" relation

As in "Student **is-a** Person"

Generalise.
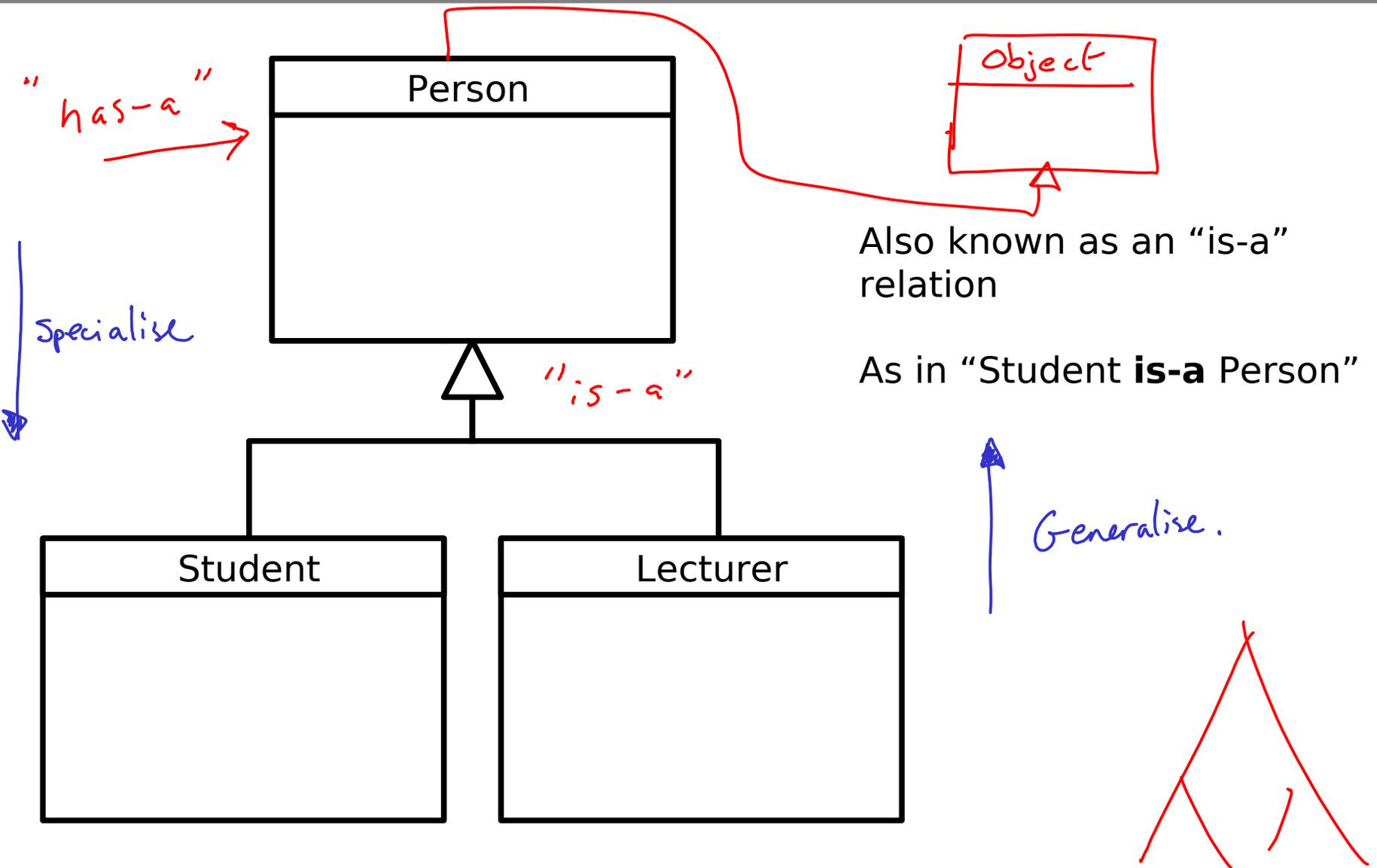
Casting/Conversions

- As we descend our inheritance *tree* we specialise by adding more detail (a salary variable here, a dance() method there)

- So, in some sense, a Student object has all the information we need to make a Person (and some extra).

- It turns out to be quite useful to group things by their common ancestry in the inheritance tree

- We can do that semantically by expressions like:
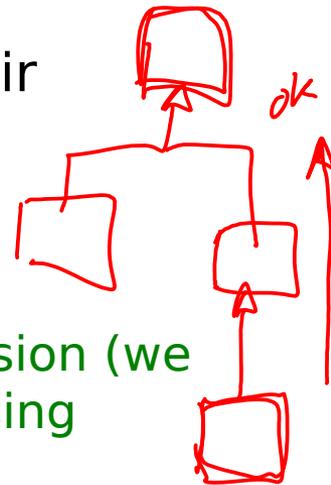
```
Student s = new Student();
Person p = (Person)s;
```

This is a *widening* conversion (we move up the tree, increasing generality: always OK)

```
Person p = new Person();
Student s = (Student)p;
```
X

This would be a *narrowing* conversion (we try to move down the tree, but it's not allowed here because the real object doesn't have all the info to be a Student)

Variables + 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 as a private variable and so cannot access it

```
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;
  }
}
```

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 being shadowed: i.e. nothing is being replaced. This is contrast to the behaviour with methods…
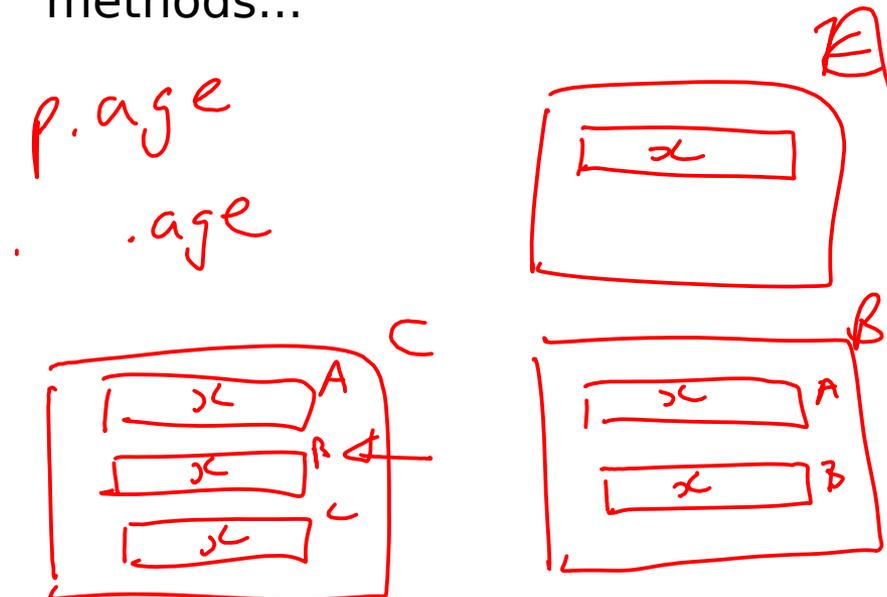
Methods + Inheritance: Overriding

- 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...

Person defines a 'default' implementation of dance()
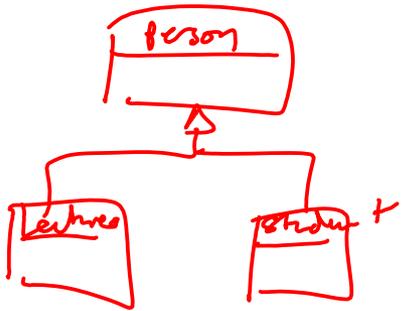
```
class Person {
  public void dance() {
    jiggle_a_bit();
  }
}

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

class Lecturer extends Person {
}
```

Student overrides the default

Lecturer just inherits the default implementation and jiggles

(Subtype) Polymorphism

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

- Assuming Person has a default dance() method, what should happen here??
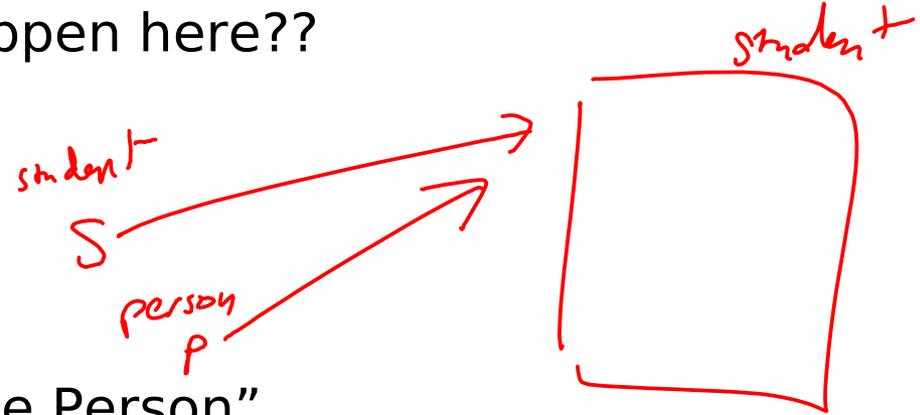
- **Option 1**
  - Compiler says "p is of type Person"
  - So p.dance() should do the default dance() action in Person

- **Option 2**
  - Compiler says "The object in memory is really a Student"
  - So p.dance() should run the Student dance() method

Polymorphic behaviour

The Canonical Example

Circle
+ draw()

Square
+ draw()

Oval
+ draw()

Star
+ draw()

- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects
- **Option 1**
  - Keep a list of Circle objects, a list of Square objects,...
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?
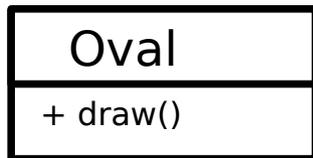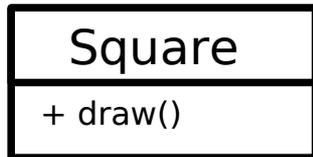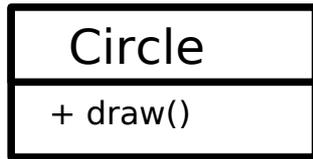
The Canonical Example
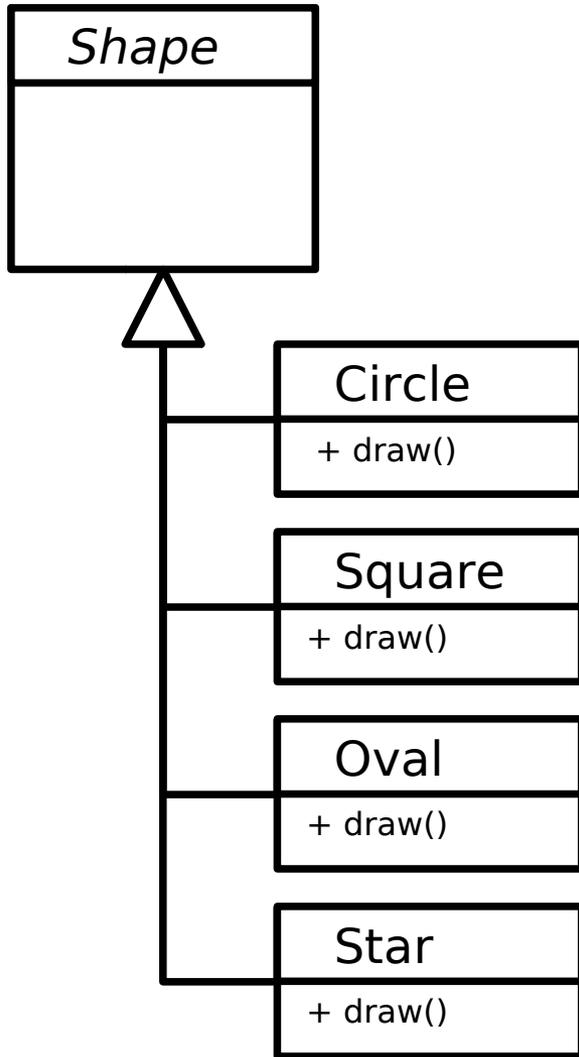
- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

  ```
  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

Shape
- x_position: int - y_position: int + *draw()*

Circle
+ draw()

Square
+ draw()

Oval
+ draw()

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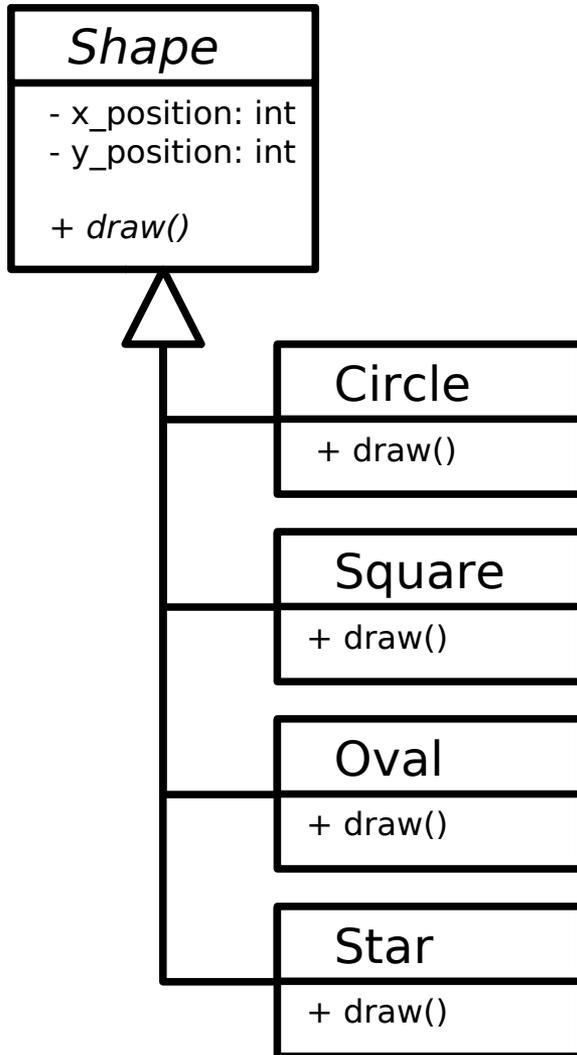
- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

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

  - What if we want to add a new shape?

Implementations

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

- Polymorphism is an extremely important concept that you need to make <u>sure</u> you understand…

Abstract Methods

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

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

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

- There are times when we have a definite concept but we expect every specialism of it to have a different implementation (like the draw() method in the Shape example). We want to enforce that idea without providing a default method

- E.g. We want to enforce that all objects that are Persons support a dance() method
  - But we don't now think that there's a default dance()

- We specify an **abstract** dance method in the Person class
  - i.e. we don't fill in any implementation (code) at all in Person.

Abstract Classes

- Before we could write Person p = new Person()
- But now p.dance() is undefined
- Therefore we have implicitly made the class abstract ie. It cannot be directly instantiated to an object
- Languages require some way to tell them that the class is meant to be abstract and it wasn't a mistake:

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

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

- Note that an abstract class can contain state variables that get inherited as normal
- Note also that, in Java, we can declare a class as abstract despite not specifying an abstract method in it!!

Representing Abstract Classes

Multiple Inheritance

```
┌─────────────┐   ┌─────────────┐
│   Student   │   │   Lecturer  │
├─────────────┤   ├─────────────┤
│             │   │             │
└─────────────┘   └─────────────┘
       △                 △
       │                 │
       └────────┬────────┘
                │
    ┌───────────────────────┐
    │    StudentLecturer     │
    ├───────────────────────┤
    │                       │
    └───────────────────────┘
```
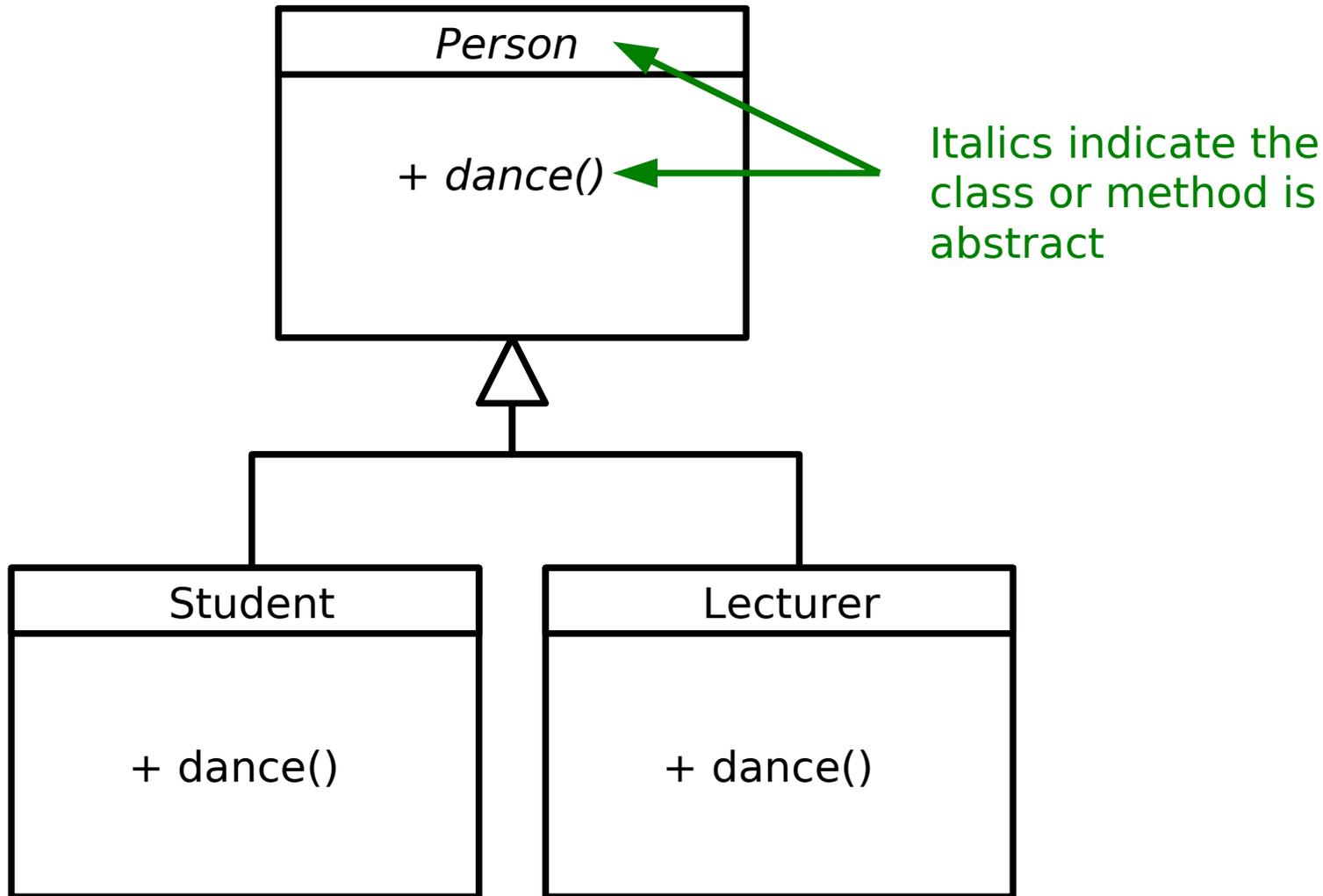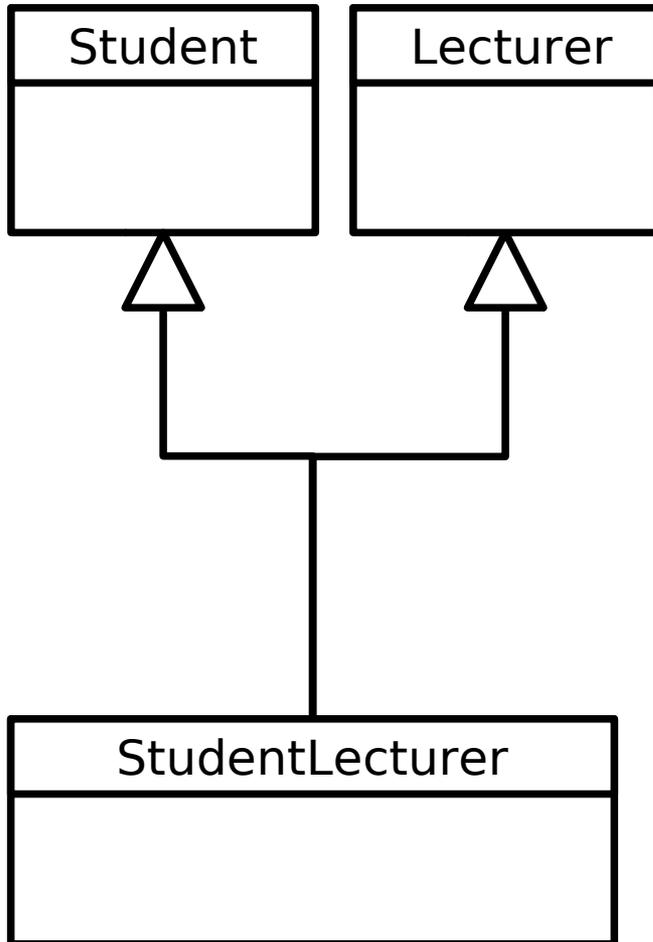
- What if we have a Lecturer who studies for another degree?
- If we do as shown, we have a bit of a problem
  - StudentLecturer inherits two different dance() methods
  - So which one should it use if we instruct a StudentLecturer to dance()?
- The Java designers felt that this kind of problem mostly occurs when you have designed your class hierarchy badly
- Their solution? **You can only extend (inherit) from one class in Java**
  - (which may itself inherit from another…)
  - This is a Java oddity (C++ allows multiple class inheritance)

Interfaces (Java only)

- Java has the notion of an **interface** which is like a class except:
  - There is no state whatsoever
  - <u>All</u> methods are abstract
- For an interface, there can then be no clashes of methods or variables to worry about, so we can allow multiple inheritance

```
<<interface>>
Drivable

+ turn()
+ brake()
```

```
<<interface>>
Identifiable

+ getIdentifier()
```

```
Bicycle

+ turn()
+ brake()
```

```
Car

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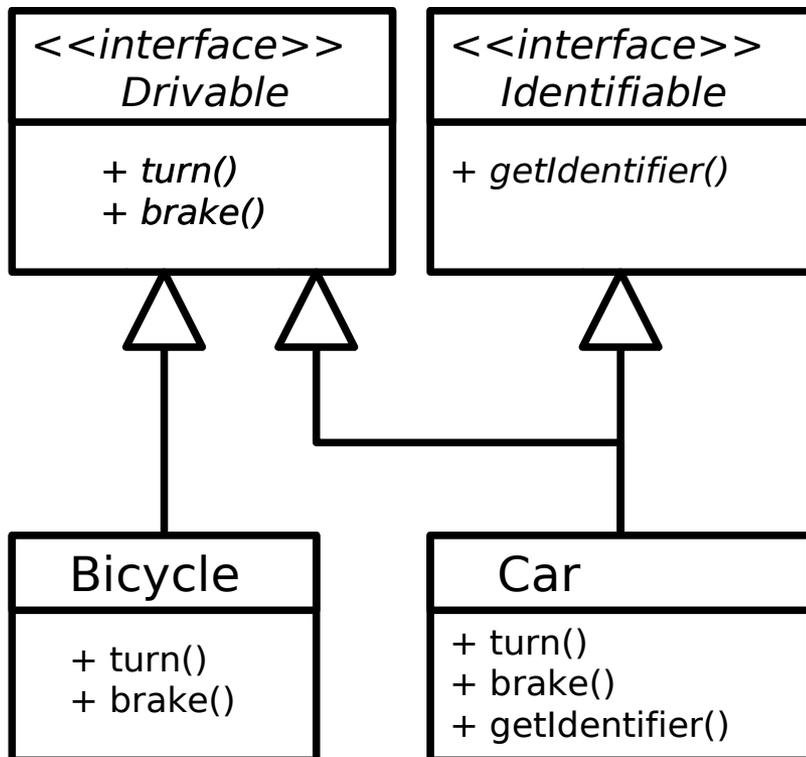
```
Interface Drivable {
    public void turn();
    public void brake();
}
```

abstract assumed for interfaces

```
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() {...}
}
```

Recap

- Important OOP concepts you need to understand:

  - Modularity (classes, objects)
  - Data Encapsulation
  - Inheritance
  - Abstraction
  - Polymorphism

Lifecycle of an Object

Constructors

`MyObject m = new MyObject();`

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.

- It's a method that gets called when the object is constructed, and it goes by the name of a constructor (it's not rocket science).

- We use constructors to initialise the state of the class in a convenient way.
  - A constructor has the same name as the class
  - A constructor has no return type specified

Examples

```java
public class Person {
  private String mName;

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

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

}
```

Java

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

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


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

C++

Default Constructor

```
public class Person {
  private String mName;

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

}
```

- If you specify no constructor at all, the Java fills in an empty one for you
- The default constructor takes no arguments

Multiple Constructors

```
public class Student {
   private String mName;
   private int mScore;

   public Student(String s) {
      mName=s;
      mScore=0;
   }
   public Student(String s, int sc) {
       mName=s;
       mScore=sc;
   }


   public static void main(String[] args) {
     Student s1 = new Student("Bob");
     Student s2 = new Student("Bob",55);
   }
 }
```

- You can specify as many constructors as you like.
- Each constructor must have a different signature (argument list)

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();
```

Animal
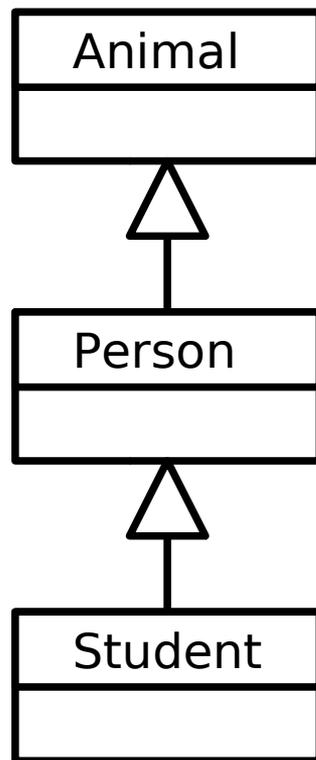
1. Call Animal()

Person

2. Call Person()

Student

3. Call Student()

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

```cpp
class FileReader {
  public:

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

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

  private :
    FILE *file;
}
```

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

  // Construct a FileReader Object
  FileReader *f = new FileReader();

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

C++

Cleaning Up

- A typical program creates lots of objects, not all of which need to stick around all the time

- **Approach 1:**
  - Allow the programmer to specify when objects should be deleted from memory
  - Lots of control, but what if they forget to delete an object?

- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object is finished with if the programmer doesn't explicitly tell you it is?

Cleaning Up (Java)

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

Person object
#ref = 2

r1

r2

r1 = null;
r2 = null;

Person object
#ref = 0

Deletable

r1

r2
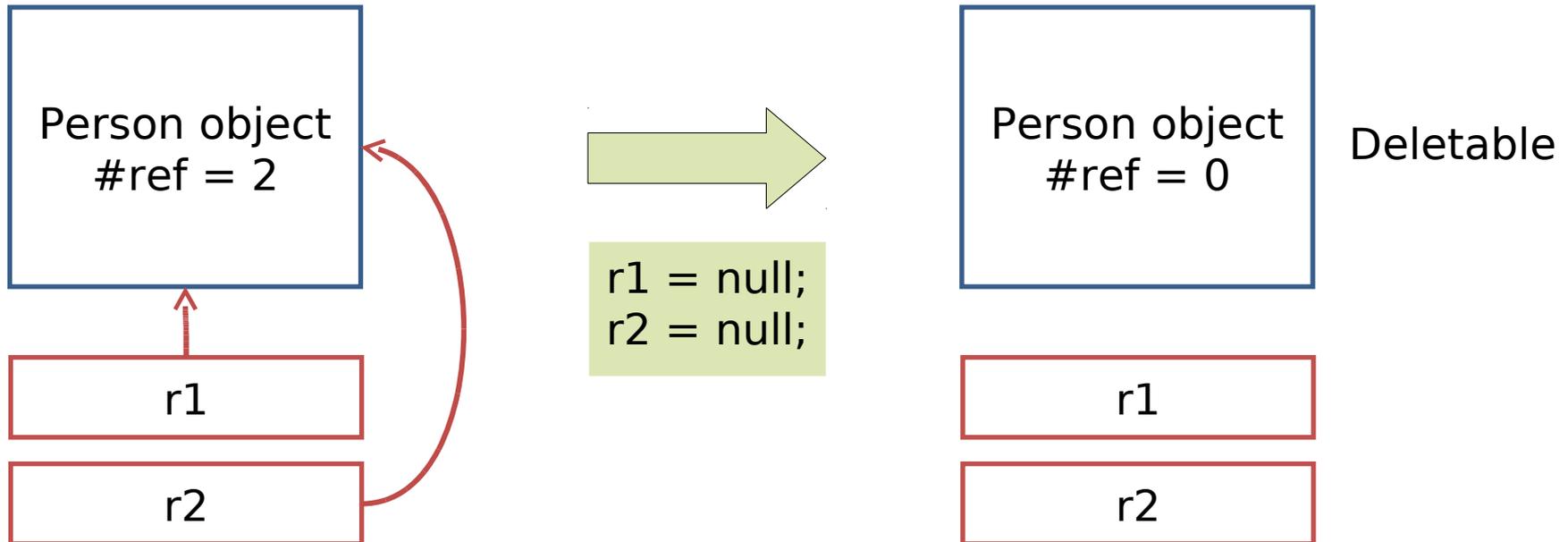
Cleaning Up (Java)

- **Good:**
  - System cleans up after us

- **Bad:**
  - It has to keep searching for objects with no references. This requires effort on the part of the CPU so it degrades performance.
  - We can't easily predict when an object will be deleted

Cleaning Up (Java)

- So we can't tell when a destructor would run – so Java doesn't have them!!

- It does have the notion of a **finalizer** that gets run when an object is garbage collected

  - BUT there's no guarantee an object will <u>ever</u> get garbage collected in Java...

  - Garbage Collection != Destruction

Class-Level Data

Class-Level Data and Functionality

```
public class ShopItem {
  private float price;
  private float VATRate = 0.175;

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

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

}
```

- Imagine we have a class ShopItem. Every ShopItem has an individual core price to which we need to add VAT
- Two issues here:
  1. If the VAT rate changes, we need to find every ShopItem object and run SetVATRate(...) on it. We could end up with different items having different VAT rates when they shouldn't...
  2. It is inefficient. Every time we create a new ShopItem object, we allocate another 32 bits of memory just to store exactly the same number!

- What we have is a piece of information that is class-level not object level

  - Each individual object has the same value at all times

- We throw in the **static** keyword:

```
public class ShopItem {
  private float price;
  private static float VATRate;

  ....
}
```

Variable created only once and has the lifetime of the program, not the object

Class-Level Data and Functionality

17.5

17.5

17.5

17.5
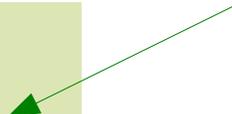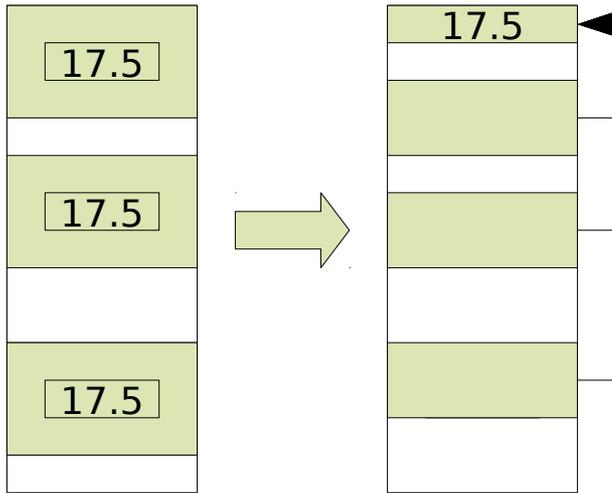
- We now have one place to update
- More efficient memory usage

- Can also make methods **static** too
  - A static method must be instance independent i.e. it can't rely on member variables in any way
- Sometimes this is obviously needed. E.g

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

Must be able to run this function without creating an object of type Whatever (which we would have to do in the main()..!)

Why use other static functions?

- A static function is like a function in ML – it can depend only on its arguments

  - Easier to debug (not dependent on any state)

  - Self documenting

  - Allows us to group related methods in a Class, but not require us to create an object to run them

  - The compiler can produce more efficient code since no specific object is involved

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

vs

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

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

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

Vector2D Example

- We will create a class that represents a 2D vector

Vector2D
- mX: float - mY : float + Vector2D(x:float, y:float) + GetX() : float + GetY() : float + Add(Vector2D v) : void