Programming 1

Further Java:

Lecture 1:
- Encapsulation
- Packaging

Getting back on the Java horse

- Remember: Java is NOT Javascript!
- Remember: Javascript is NOT Java!

- There are similarities…
- There are differences…

- You will need to put aside some of what you've become accustomed to in Javascript and get your Java habits back ☺

Getting back on the Java horse

- Key things to remember about Java given that you're coming from Javascript and you've probably forgotten some important differences…)
  - Java is **strongly typed**
    - This means that you MUST specify the type of your variable when you declare it:
    - Java

```
String name = "Paul";
int age = 21;
double nfh = 99.999;
```

Javascript

```
var name = "Paul";
var age = 21;
var nfh = 99.999;
```

Getting back on the Java horse

- You can ONLY declare the variable once (unlike Javascript which lets you get away with re-declaring variables)
  - This means that you MUST specify the type of your variable when you declare it:
  - Java / Javascript

```
String name = "Paul";            var name = "Paul";
System.out.println(name);        console.log(name);
name = "Fred";                   name = "fred";
```

```
String name = "Paul";            var name = "Paul";
System.out.println(name);        console.log(name);
String name = "Fred";            var name = "fred";
```

Getting back on the Java horse

- You can ONLY declare the variable once (unlike Javascript which lets you get away with re-declaring variables)
  - This means that you MUST specify the type of your variable when you declare it:
    - Java / Javascript

```
String name = "Paul";              var name = "Paul";
System.out.println(name);          console.log(name);       both correct
name = "Fred";                     name = "fred";
```

~~String name = "Paul";~~
~~System.out.println(name);~~
~~String name = "Fred";~~

Would NOT compile

```
var name = "Paul";
console.log(name);
var name = "fred";
```

Is incorrect, but would run

Getting back on the Java horse

- Use System.out.println to print text on the console
- Create a Scanner object and then call **`nextLine`** or **`nextInt`** to get input from the console

Java

```
Scanner keys = new Scanner();

System.out.println("Type text");
String text = keys.nextLine();
System.out.println(text);

System.out.println("Type a number");
String num = keys.nextInt();
System.out.println(num);
```

Javascript

```
var text = prompt("Type text");
console.log(text);

var num = prompt("Type a number");
console.log(num);
```

Getting back on the Java horse

- Use System.out.println to print text on the console
- Create a Scanner object and then call **`nextLine`** or **`nextInt`** to get input from the console

Java

```
Scanner keys = new Scanner();

System.out.println("Type text");
String text = keys.nextLine();
System.out.println(text);

System.out.println("Type a number");
String num = keys.nextInt();
System.out.println(num);
```

Javascript

```
var text = prompt("Type text");
console.log(text);

var num = prompt("Type a number");
console.log(num);
```

You only need to declare the Scanner object ONCE!

Assign it to a variable (in this case, we used a variable called **`keys`**) and then you can refer to the variable and call **`nextLine`** and/or **`nextInt`** as many times as you need!

Getting back on the Java horse

- Java is an **object oriented language**
  - (This unit will focus predominantly on that characteristic of Java)
- EVERYTHING in Java is a **class**

- Ideally, a class should describe a real world thing
  - e.g. *student*, *ball*, *house*, *garden*, *pet*

Java: a class act

- A class has **attributes** and **methods**

- Attributes describe what distinguishes individual examples of a class from others ones
  - What makes one ball different from another?
  - What makes one pet different from another?

- Methods are things that the class can do
  - What can a ball do?
  - What can a pet do?

Java: a class act

- The code that defines a class does NOT refer to any specific instance of a class or object
- Consider if I defined a student in English:
  - *A student is an* ***individual*** *who attends a* ***university****. They have a* ***name****,* ***address****,* ***gender*** *and* ***ID number****. They attend a number of different* ***modules****.*
- and consider a specific student:
  - **Fred Smith** attends **Kingston University** and his K-number is **K1469123**. **He** lives in **Surbiton**, and he studies the modules [**Programming 1**, **IT Toolbox**, **System Environments** and **Neutron Bomb Juggling**.]
- The **attributes** are what distinguish Fred Smith from (say) Jill Jones
- The *definition* (in green) has nothing specific to do with Fred Smith. It just lays down the rules for students in general

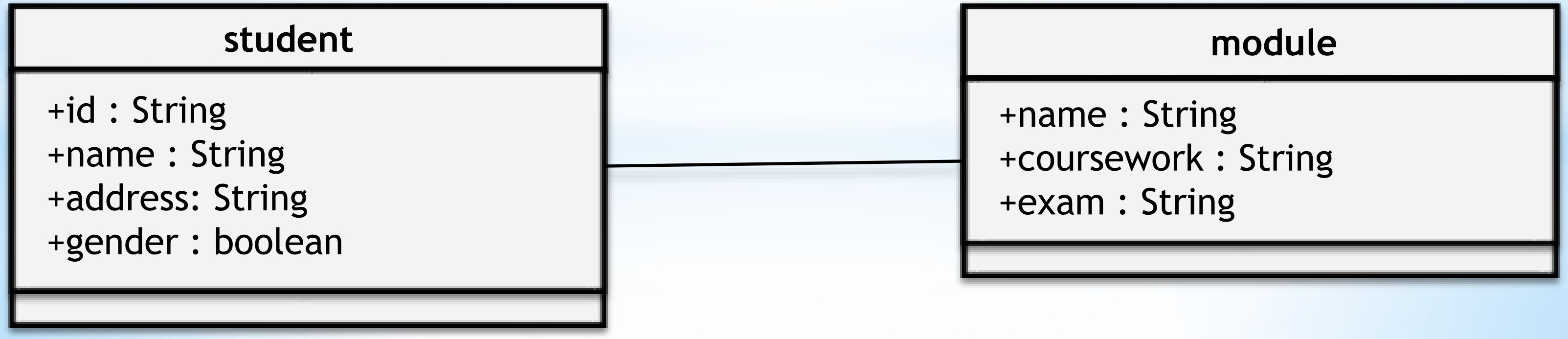
Java: a class act

- Classes can have relationships with other classes:
- Consider if I defined a student in English:
  - *A student is an **individual** who attends a **university**. They have a **name**, **address**, **gender** and **ID number**. They attend a number of different **modules**.*

- A module itself is another class, e.g.
  - A module has a name, a coursework and an exam

Java: a class act

- Classes can have relationships with other classes:
- Consider if I defined a student in English:
  - *A student is an **individual** who attends a **university**. They have a **name**, **address**, **gender** and **ID number**. They attend a number of different **modules***
- A module itself is another class, e.g.
  - A module has a name, a coursework and an exam



Java: a class act

- Defining the classes:

```
public class Student
{
   public String id;
   public String name;
   public String address;
   public String gender;

   // what about modules?
}
```

Java: a class act

- Defining the class:

```
public class Student
{
   public String id;
   public String name;
   public String address;
   public String gender;

   public Module[] modules;
}
```

```
public class Module
{
   public String name;
   public String coursework;
   public String exam;
}
```

Java: a class act

- Remember that when we define a class, we are not giving any details about any specific *instance* of a class
- When we define `Student`, we are not giving any details of (for example) Jack Smith:

```
public class Student
{
   public String id;
   public String name;
   public String address;
   public String gender;

   public Module[] modules;
}
```

Do you see any reference to an individual student here?!

If what you are trying to do refers to a specific individual, it does NOT go in the class!

Methods in the madness

- The methods of a class specify what a class can *do*
- Let's switch to "ball" as our example – it's a bit less abstract
- A ball has a **diameter** and a **colour** (attributes; things that distinguish one ball from another)
- A ball can **bounce** and **roll** (methods; things that objects of this class can do)

Back to our balls

- A Java class to define a ball might look like this

```
public class Ball
{
   public double diameter;
   public String colour;

   public void bounce()
   {
      System.out.println("Boinged "+this.diameter*2+" high");
   }

   public void roll()
   {
      System.out.println("Whee");
   }
}
```

Building an instance of a class

- If we want to create a single individual instance of a class, we use the **`new`** command
- This uses the class definition to create a new, individual, instance of that class
- This will have all the attributes and methods from the class
  - ...although we'll need to set the attributes for each new instance

```
Ball tennisBall = new Ball();
tennisBall.diameter = 6.35;
tennisBall.colour = "green";

tennisBall.bounce();
```

```
Ball golfBall = new Ball();
golfBall.diameter = 2.65;
golfBall.colour = "white";

golfBall.bounce();
```

- Methods might make use of the attributes
  - This means that when the method is called, the result can be instance-specific

Back to our balls

- What would the output be?

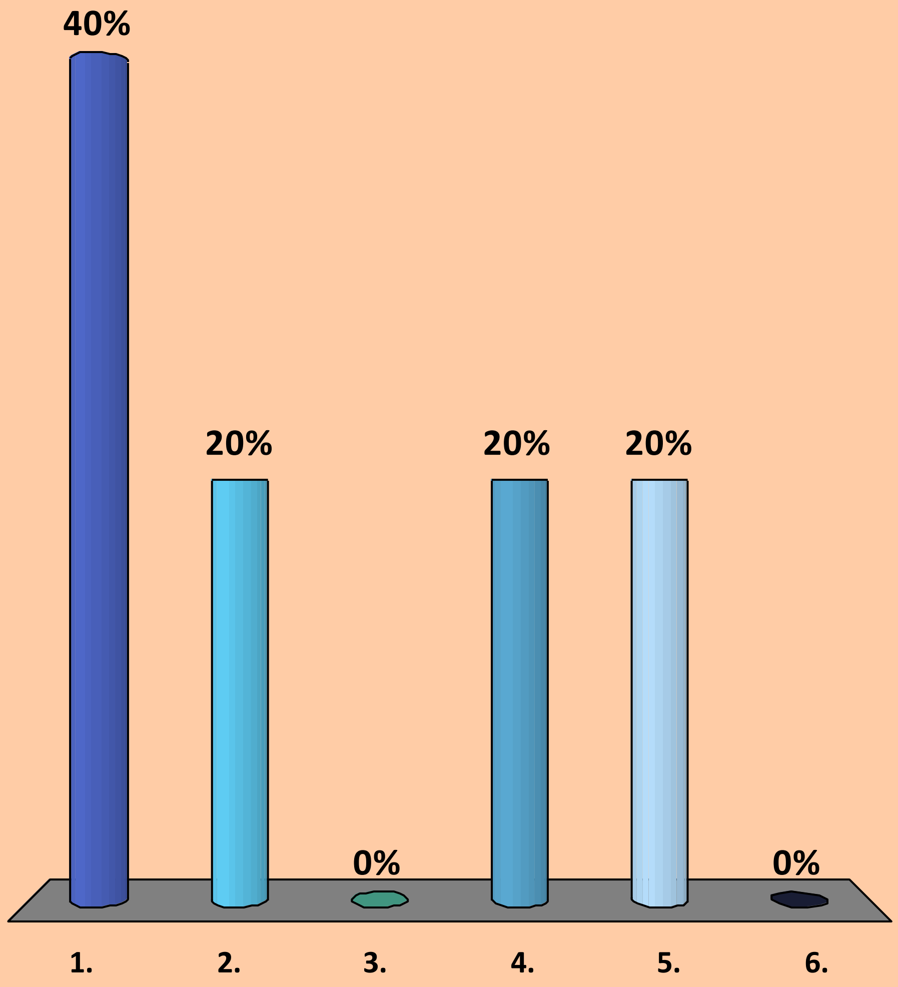
```
public class Ball
{
   public double diameter;
   public String colour;

   public void bounce()
   {
     System.out.println("Boinged "+this.diameter*2+" high");
   }

   public void roll()
   {
     System.out.println("Whee");
   }
}
```

```
Ball tennisBall = new Ball();
tennisBall.diameter = 6.35;
tennisBall.colour = "green";

tennisBall.bounce();
```

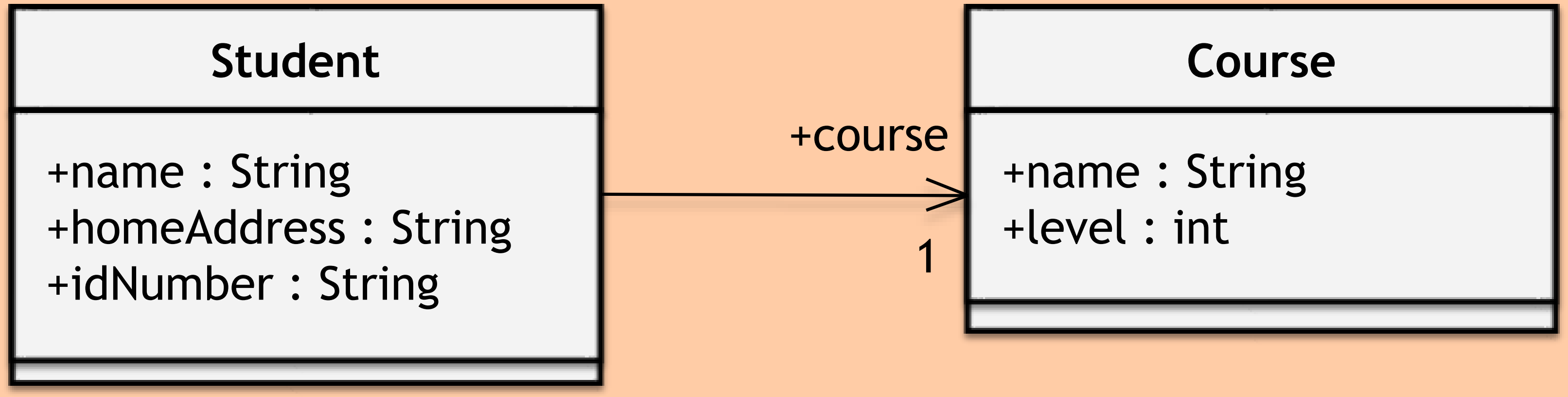
```
Ball golfBall = new Ball();
golfBall.diameter = 2.65;
golfBall.colour = "white";

golfBall.bounce();
```

Which of these would NOT be a method in a university class?

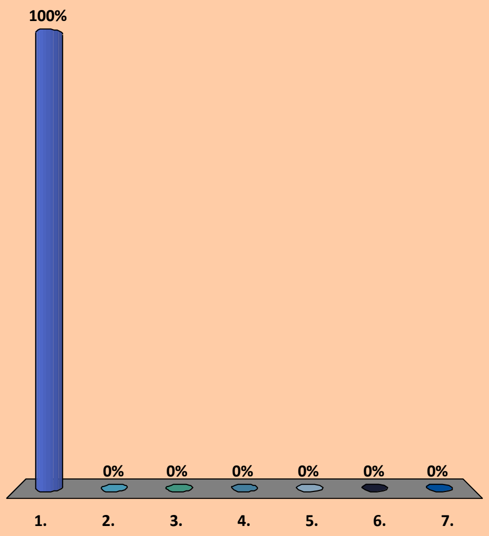
1. Take Exam
2. Fail Programming 1
3. Enrol on Module
4. Graduate Student
5. Assign Tutor to Workshop
6. Log attendance



What's the correct line to declare the attribute `course` within the class `Student`?



1. public Course course;
2. public course Course;
3. public String course;
4. public course String;
5. public Student course;
6. public Course student;
7. public course;



The main ~~man~~ method

- So… Java programs are a collection of related classes
- There might be dozens of different class files involved
- When you press that "run" button, the program has to start somewhere
- That somewhere is the *main method*

The main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

Main Method

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

Variable declaration

tennisBall

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

Object creation

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

Assigning a new object to a variable

tennisBall

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

Setting an attribute

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  p
  p

  p
  {
                          ing");
  }

  p
  {
                          ee");
  }
}
```

tennisBall

cricketBall

- A "main" class

```
public class Mainv
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
```

```
"Boing");

  public void roll()
  {
    System.out.println("Whee");

}
```

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
      cketBall.diameter = 9;
      cketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

what's your colour?

Accessing an attribute

Dissecting the main method

- The ball class

```
public class Ball
{
  public double diameter;
  public String colour;

  public void bounce()
  {
    System.out.println("Boing");
  }

  public void roll()
  {
    System.out.println("Whee");
  }
}
```

- A "main" class

```
public class Main
{
  public static void main (String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.diameter = 6.5;
    tennisBall.colour = "green";

    Ball cricketBall = new Ball();
    cricketBall.diameter = 9;
    cricketBall.colour = "red";

    System.out.println("A cricket ball is");
    System.out.println(cricketBall.colour);
    tennisBall.roll();
  }
}
```

Calling a method

Java in a nutshell

- **Java is OBJECT ORIENTED**
  - That means that everything has to be expressed in terms of classes that (might) relate to each other
  - Classes have attributes and methods
  - Somewhere in your project there will be a class that has a **main** method
  - This is where your program starts
  - The main method will create *instances* of your classes, set their attributes, call their methods
  - The act of creating instances of classes, calling methods, etc may well create new instances of classes…
  - This merry dance of different classes IS the average Java program!

Java in a nutshell

- Java is **STRONGLY TYPED**
  - Every variable MUST be declared when it is first used
  - Every variable MUST has its data type specified when it is declared
  - You (usually) cannot mix and match data types
  - If you break these rules your project will not compile

Other things you need to revise…

- Don't forget that methods can have **parameters** and **return values**
  - (these work exactly like those in Javascript functions… so shouldn't be too much of a problem for you by now…!)
- Classes can also have a special method called a **constructor**
  - A constructor is called whenever an instance of a class is created
  - Constructors are often used to "set up" a new instance of a class, populate attributes with default values, etc.

Visibility & encapsulation

- Up until now, all of our attributes in our classes have been `public`
- This is not ideal practice

Attribute visibility

- The ball class

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;
    return height ;
  }
}
```

- `public` means an attribute can be directly accessed from any other class
- But what about if invalid values are set?

public attributes and validation (or the lack thereof)

```
public class Ball
{
  public double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball oddBall = new Ball();
    oddBall.diameter = -10;
    double bounced = oddBall.bounce();
  }
}
```

- Will `bounced` contain a meaningful value with respect to the real world object our class is supposed to represent?

private attributes

- It is considered best practice to make ALL attributes of a class private, not public
- This means that they are only visible and modifiable from within the class itself…

private attributes

```
public class Ball
{
  private  double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball oddBall = new Ball();
    oddBall.diameter = -10;
    double bounced = oddBall.bounce();
  }
}
```

- This is called setting the attributes *visibility*
- …but now the Main class won't compile…

private attributes

```
public class Ball
{
  private double diameter;

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball oddBall = new Ball();
    ~~oddBall.diameter = -10;~~
    double bounced = oddBall.bounce();
  }
}
```

- …we can no longer set the diameter from outside the class Ball, because it is *private*
- So.. if best practice is to make attributes in our class private, how on earth do we get and set them?!

getters and setters

- We add a **setter** method so we can set the value of a private attribute
  - The setter method takes a parameter, and the value of this parameter is assigned to the attribute
- We add a **getter method** so we can retrieve the value of a private attribute
  - The getter method will *return* the current value of the attribute

getters and setters in use

```
public class Ball
{
  private double diameter;

  public void setDiameter(double d)
  {
     this.diameter = d;
  }

  public double getDiameter()
  {
     return this.diameter;
  }

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.setDiameter(6.5);
    double bounced = tennisBall.bounce();
    double tbd = tennisBall.getDiameter();
    System.out.println("Diameter is "+tbd);
    System.out.println("Bounced "+bounced);
  }
}
```

getters and setters in use

```
public class Ball
{
  private double diameter ;

  public void setDiameter(double d )
  {
     this.diameter = d;
  }

  public double getDiameter()
  {
     return this.diameter ;
  }

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball tennisBall = new Ball();
    tennisBall.setDiameter( 6.5 );
    double bounced = tennisBall.bounce();
    double tbd = tennisBall.getDiameter();
    System.out.println("Diameter is "+tbd);
    System.out.println("Bounced "+bounced);
  }
}
```

Why bother?

- …why not just make things public across the board and save all this effort?
- what about our oddBall instance of Ball?

```
public class Ball
{
  public double diameter;

  public void setDiameter(double d)
  {
    this.diameter = d;
  }
  // getter left off for space ☺

  public double bounce()
  {
    double height = this.diameter * 2;

    return height;
  }
}
```

```
public class Main
{
  public static void main(String[] args)
  {
    Ball oddBall = new Ball();
    oddBall.setDiameter(-10);
    double bounced = oddBall.bounce();
  }
}
```

Encapsulation

- We can add validation into our setters (and, potentially, our getters)
- We can put the logic, the controls for what is allowed into the attributes of a class in the class itself
- Think of it as a protective barrier between your class's attributes and everything else
- Things outside the class can thus not break the "rules" of what's allowed in the attributes of the class

```
public class Ball
{
  public double diameter;

  public void setDiameter(double d)
  {
    if (d < 0)
    {
      this.diameter = 10; // default
    }
    else
    {
      this.diameter = d;
    }
  }

  // getter and bounce left off
  // for space ☺

}
```

Using `this` to use the same variable name

- The usual convention when writing setters is to use the same name for the parameter as the attribute:

```
public void setDiameter(int diameter)
{
   this.diameter = diameter;
}
```

Using `this` to use the same variable name

- The usual convention when writing setters is to use the same name for the parameter as the attribute:

```
public void setDiameter(int diameter )
{                             parameter
   this.diameter = diameter ;
}      attribute
```

- The use of the **`this`** keyword explicitly refers to the attribute of the class, rather than the parameter which is only local to this setter method
- So in this case, using **`this`** means that there is no clash between the identically named variables

Packages

- Up until now, all of our classes have been in the same *package*
- This is not best practice!
- It is recommended that you put related classes in the same package
- You can then use a third visibility modified, **protected** to specify that attributes and/or methods are available only to other classes in the same package
  - (…although **protected** is used much less often than public and private…)

Playing with your package(s)

- Specify the package of a class by adding a like of code like the following at the top of your class
  - **`package com.mycompany.mypackage;`**
- Usually, the convention is to use your organisation's domain name backwards, e.g.
  - uk.ac.kingston
  - com.paulneve
  - uk.co.google

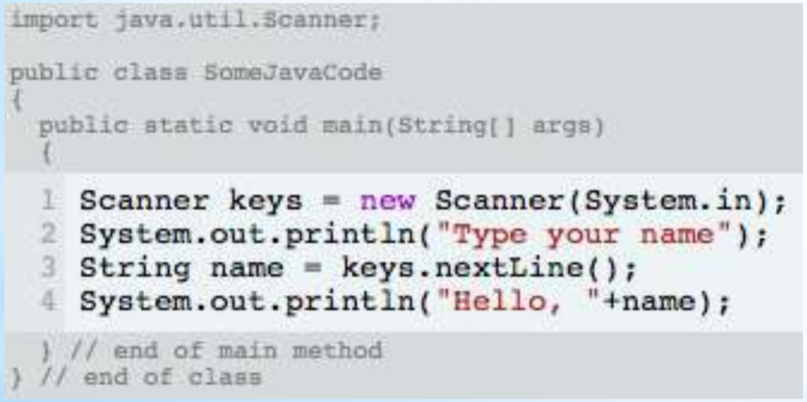
  then append your package name, e.g.
  - uk.ac.kingston.**model**

Packages and imports

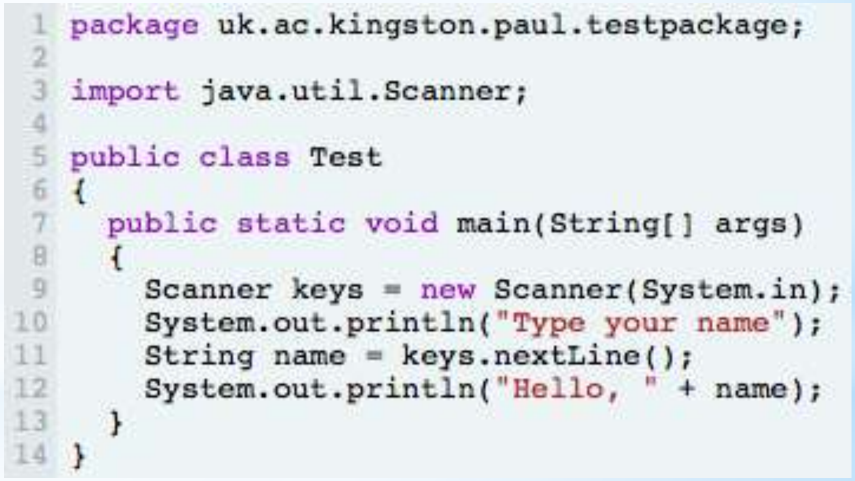
- If you want to use a class from a different package than the current class you need to **import** it
- Use the import statement at the top of the class (under any package declaration) to import an external class
- You may have used import before – can anyone tell me what for?

Packages and imports

- When we use a scanner, we have to import it from the package **java.util**
- **java.util** is a package that comes with Java itself
  - it contains a variety of classes and utilities – including the Scanner class



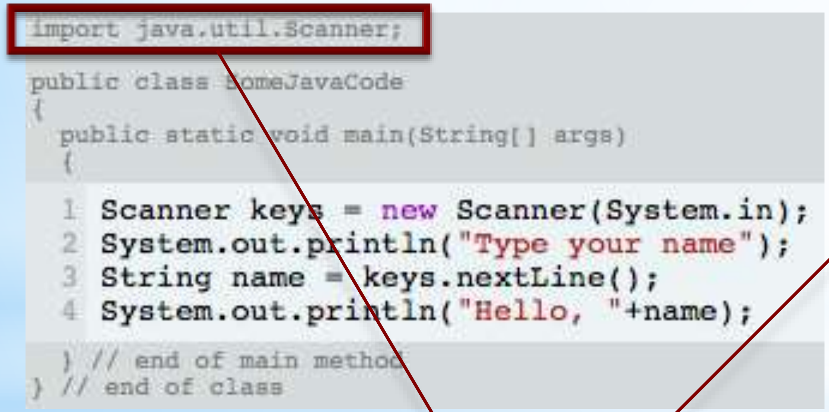
```
import java.util.Scanner;

public class SomeJavaCode
{
  public static void main(String[] args)
  {
    Scanner keys = new Scanner(System.in);
    System.out.println("Type your name");
    String name = keys.nextLine();
    System.out.println("Hello, "+name);
  } // end of main method
} // end of class
```



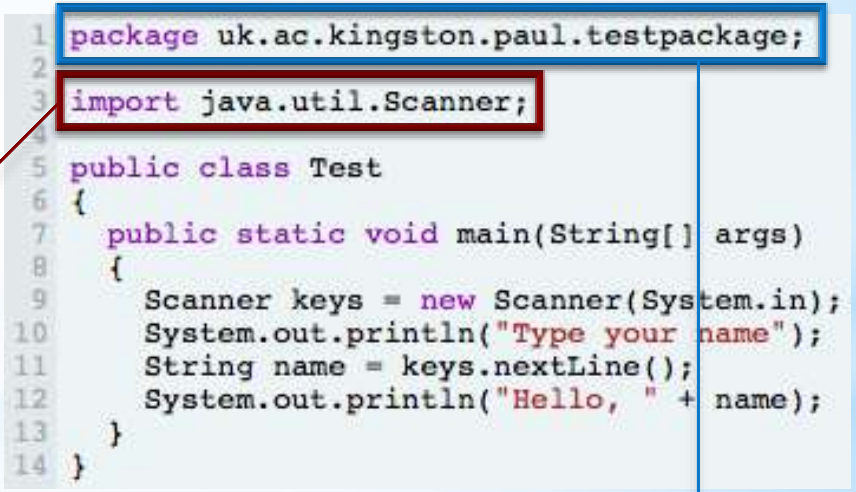
```
package uk.ac.kingston.paul.testpackage;

import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner keys = new Scanner(System.in);
    System.out.println("Type your name");
    String name = keys.nextLine();
    System.out.println("Hello, " + name);
  }
}
```

Packages and imports

- When we use a scanner, we have to import it from the package **java.util**
- **java.util** is a package that comes with Java itself
  - it contains a variety of classes and utilities – including the Scanner class



```java
import java.util.Scanner;

public class SomeJavaCode
{
  public static void main(String[] args)
  {
    Scanner keys = new Scanner(System.in);
    System.out.println("Type your name");
    String name = keys.nextLine();
    System.out.println("Hello, "+name);
  } // end of main method
} // end of class
```



```java
package uk.ac.kingston.paul.testpackage;

import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner keys = new Scanner(System.in);
    System.out.println("Type your name");
    String name = keys.nextLine();
    System.out.println("Hello, " + name);
  }
}
```

importing a class
from another package

specifying the
package of the
current class

Organising your code

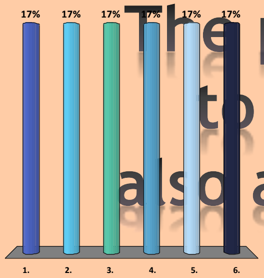
- When you have a class in a package, you are also creating a directory structure
- If you have a class that specifies a package name of uk.ac.kingston.paul, then it must be in the directory
  - **uk/ac/kingston/paul**
- If you were building a Java application from scratch, you would need to create these directories (many of which might be empty)
- Most integrated development environments (IDEs), including NoobLab, will do this for you
- Try creating a multiclass project, put your classes in packages, download your project from NoobLab as a ZIP, then unzip the ZIP and look at the directories…

Don't misplace your packages!

- If you have a class in a package, and you want to use it somewhere else, you MUST import it
- The only exception is if the current class is in the same package as the destination class
- For example, in the demo you'll see now during the lecture:
  - **Dog** is in the same package as **Cat**.
  - You can use **Dog** within **Cat** and vice-versa without importing
  - You cannot use either **Dog** or **Cat** in the Main class without importing them

Possible package structures

- Put things that are related together, e.g.
  - if you have a **Dog, Cat** and **Budgie** then perhaps they might go in a **pets** package
  - if you have classes that contain general utilities then perhaps they should go in a **utils** package
  - if you have a bunch of classes responsible for your data model, others that handle the control functions of your application, and others that handle what the end-user sees or views, perhaps you might have **model**, **view** and **controller** packages

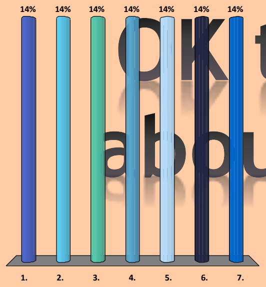


The programmer who wrote the class below was trying to use encapsulation to protect their attributes, and also add some "defaulting" behaviour, but the attribute *level* isn't being set properly. Why?

```
public class Player
{
  private String name;
  private int level;
  private String game;

   (all getters and setters for name
    and game are assumed, and the
    getter for level is also assumed)

  public void setLevel(int lev)
  {
    if (game.equalsIgnoreCase("golf"))
    {
      double level = 10;
    }
    else
    {
      double level = lev;
    }
  }
}
```

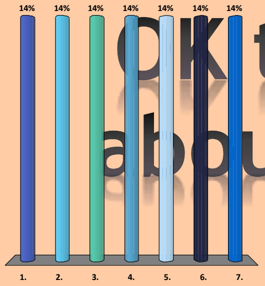
1. The parameter in the method signature is incorrect - it should be **`level`** not **`lev`**
2. The return type in the method signature is incorrect - it should return an **`int`** not a **`void`**
3. The lines where **`level`** is being set are incorrect
4. Option 2, plus also you will need a line at the end of the method to return what **`level`** is being set to
5. Trick question – **`level`** will get set just fine
6. Some other reason or Paul has cocked up the slide!

OK then, what would you need to change about the lines where `level` is being set?



```
public class Player
{
  private String name;
  private int level;
  private String game;

   (all getters and setters for name
    and game are assumed, and the
    getter for level is also assumed)

  public void setLevel(int lev)
  {
    if (game.equalsIgnoreCase("golf"))
    {
      double level = 10;
    }
    else
    {
      double level = lev;
    }
  }
}
```

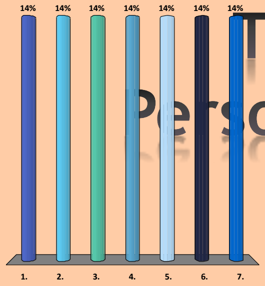
1. You should change the double type to an int
2. You should remove the data type entirely
3. You should remove the data type and append this. to the attribute name
4. Both option 2 and 3 would work in this particular case, but you should really do 2 from a best practice perspective
5. Both option 2 and 3 would work in this particular case, but you should really do 3 from a best practice perspective
6. Options 1, 2 and 3 would all work
7. Something else or Paul has cocked up the slide!



OK then, what would you need to change about the lines where `level` is being set?

```
public class Player
{
  private String name;
  private int level;
  private String game;

   (all getters and setters for name
    and game are assumed, and the
    getter for level is also assumed)

  public void setLevel(int lev)
  {
    if (game.equalsIgnoreCase("golf"))
    {
      double level = 10;
    }
    else
    {
      double level = lev;
    }
  }
}
```

1. You should change the double type to an int
2. You should remove the data type entirely
3. You should remove the data type and append this. to the attribute name
4. Both option 2 and 3 would work in this particular case, but you should really do 2 from a best practice perspective
5. Both option 2 and 3 would work in this particular case, but you should really do 3 from a best practice perspective
6. Options 1, 2 and 3 would all work
7. Something else or Paul has cocked up the slide!



The following code describes an OO model where a Person can have one or more Groups. However, it does not compile. What *should* you do to fix this?

Class 1:

```
package uk.ac.kingston.single;

public class Person
{
  private String name;
  private Group[] groups;

   (getters and setters assumed)
}
```

Class 2:

```
package uk.ac.kingston.many;

public class Group
{
  private String name;
  private int grading;

   (getters and setters also assumed)
}
```

1. You should remove the package lines in both classes
2. You should change the package line in Class 1 so that it is in the **`many`** package
3. You should change the package line in Class 2 so that it is in the **`single`** package
4. In class 1, you should add an import statement to import class 2
5. In class 2, you should add an import statement to import class 1
6. There is nothing wrong - the code is fine
7. Something else or Paul has cocked up the slide!

Summary

- Methods can be given **parameters** from and **return values** to the code that calls them
- **Visibility** affects whether or not an attribute (and a method, for that matter) can be accessed directly outside of its class
- Best practice is to make attributes **private**
- We then create **getter** and **setter** methods so we can access our attributes from elsewhere

Summary

- Packages are used to organise multi-class projects
- A package name must be specified as the FIRST line of a class
- Packages usually start with a reversed domain name and then you append whatever package name you want
- If a class is in a different package to another class, the second class has to use **import** before it can use the first one
- Packages correspond to a directory structure on the hard disk where the classes are stored