2020 CI401
Introduction to programming

Week 2.03
Scope, visibility and encapsulation

22nd February 2021

Roger Evans
Module leader

Lecture recording and attendance

- This lecture will be recorded and published in the module area
- The focus of recording is on the lecturer, not the audience
- If you are particularly concerned not to be part of the recording, it is ok to turn off your microphone and camera

- In addition, lecture attendance is now being routinely recorded (in all modules) to help the School Office monitor engagement

- (This slide is really a reminder to me to start recording and record attendance!)

Module structure

Semester 2

Week	Topic	Theme	Project
2.01	Project topics and assessment	Project	Set
2.02	Simple Inheritance	OO	Lab
2.03	Scope, Visibility and Encapsulation	OO	Lab
2.04	Testing - JUnit	Testing	Lab
2.05	Documentation - Javadoc	Doc	Study
2.06	Collections and generic types	Data	Study
2.07	IO: files and streams	Dvp	Study
	Easter Vacation 29 Mar – 16 Apr		
2.08	Numbers – the computer's view	Data	Study
2.09	Java vs Python		Submit
2.10	More algorithms – search and sort	Dvp	
2.11	How fast is my code?	Dvp	
2.12	Java 'under the hood'		
2.13	Revision week		Exam ↓

Code clinic reminder

- Join code: o727xhp
- Times:
  - Monday 4pm-5pm
  - Tuesday 2pm-3pm
  - Wednesday 2pm-3pm
  - Thursday 4pm-5pm
  - Friday 2pm-3pm

Seminars this week

- 1pm
  - Goran – normal seminar, covering current topics and labs
- 3pm
  - Stelios – support session for semester 1 material, if you are feeling a bit left behind
  - Goran and Karina – normal seminar, covering current topics and labs
  - Roger – special session on the ‘Maze’ project option. Introduction to a more technical starter project with less in-class support.
- Choose between the sessions without worrying about which one you are timetabled into. If your timetable won’t let you attend the support session you want, let Roger know (R.P.Evans@brighton.ac.uk)

Introduction

Names and secrets

- This lecture is all about giving things in our programs names, so we can refer to them, and keeping things secret, so other people can't!

- The technical name for that is encapsulation, but we get there in stages:
  - Names – why names are important in programming
  - Scope – how the structure of our code controls what a name will refer to
  - Visibility – how we use access specifiers to further control who can see names
  - Encapsulation – how we use these tools to control access to our code

Encapsulation

- In a professional environment your code is usually running as part of a bigger program with parts written by other people outside your control
- Encapsulation uses inheritance, scope and visibility to control who can see, access and modify your code
- ‘Encapsulation’ means ‘wrapping up’ your code in a protective ‘capsule’
- Encapsulation makes programs more modular and robust and is a key reason why object-oriented programming has improved our ability to implement and maintain really large software systems.

Why use encapsulation

- It provides a clear controlled interface for client software to use without risk of breaking it
- It protects your code (and your reputation) from mis-use
- It allows your internal code to be more efficient, because run-time checking can be restricted to the interface
- It makes you design your code in a more robust way, by thinking about how it will be used by others etc.

Names

Names in programming

- Programming languages are all about giving names to things
- Inside the computer everything is just numbers
  - Actually they are patterns (of zeros and ones - bits), but we treat a lot of the patterns as numbers (do sums with them etc).
- People aren't so good with numbers – we tend to get them mixed up
- People like names
- So one of the main jobs of even the lowest level programming languages – assembler code – is to let us give names to things in the program

What do we give names to?

- Classes
  - `String, Car, PorcheCayenne, Model, View, Controller`
- Variables
  - `name, age, firstName, lastName, numberofDoors, i, j, k, l`
- Methods
  - `main, run, range, shortDescription, longDescription, println`

- Function words
  - `class, if, for, return, break, switch`
- Types
  - `int, boolean, float, double, char, long, short`

Using good names is important

- So you can understand your code
- So others can understand your code
- So code is predictable (so your ‘guesses’ about it are generally correct)
- So you can re-use code

Using good names is hard

- We want names to be
  - Practical – not too long
  - Clear – systematic, not confusing
  - Unambiguous – when we use a name we want to know exactly what we are referring to
- But that is difficult
  - We need too many names for the same kind of thing (year, theYear, year1, year2, y, y2 …)
  - Lots of things use names in the same way – every Student object needs to use firstName, lastName etc – they can't all be different
  - Names are ambiguous anyway – bank (finance, river), fit (physical, appearance) etc.

Names in programming

- A huge amount of what's going on in programming languages is just about what things are called. If you write "year1" or "lastName" in your code, exactly what object are you referring to?
- One way of thinking about inheritance is just in terms of names for things – if you mention a variable in a subclass and that class doesn't know the variable, it just asks its superclass if it knows it (which might ask its superclass etc.)
- In some languages (Javascript, python …), that is almost literally what happens – an 'object' is just a list of all the names associated with this class and their values, and a link to another list for all the 'superclass' names (and a link to its superclass list etc.)

Scope

Variable declarations

- When we declare a variable, we are doing several things:

```
public String firstName = "John";
```

1. Creating a variable called `firstName`
2. Giving it the type `String`
3. Initialising it to the `String` value `"John"`

- Actually there's a bit more going on in step 1:
  - We are creating a variable to hold `Strings` (a `String`-shaped box)
  - We are assigning the name `firstName` to that box
  - We are telling the whole world about `firstName` (it is `public`)

Names, Variables and Scope

- ‘Variables’ are actually the boxes – we give them names so we can refer to them in our code
- The same name can refer to different boxes at different times.
- The way we map from names to boxes at any particular point in the program is called is called variable scope
- The instruction types in Java, and especially the use of curly brackets, control scope.
- (We will talk about the public part in the next section)

Scope – the Bike class

Take a class like Bike

At the top we have some instance variables (make, model etc.)

make is the name of a variable (box)

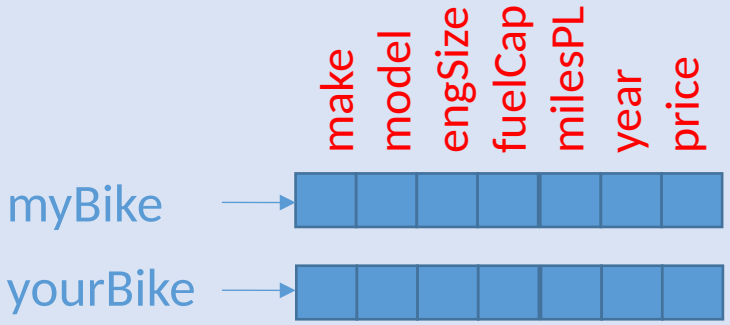
This variable can be used throughout the whole definition of the Bike class (ie from { after the class declaration to the matching } at the end)

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      public String make = "";
8      public String model = "";
9      public int engineSize;
10     public int fuelCapacity;
11     public int milesPerLitre;
12
13     // Properties that vary for each individual car
14     public int year;
15     public int price;
16
17     // Constructor
18     public Bike()
19     {
20         // nothing to do here - just use the built-in values
21     }
22
23     // Constructor to make particular bikes
24     public Bike(String theMake, String theModel, int theYear, int thePrice)
25     {
26         make = theMake;
27         model = theModel;
28         year = theYear;
29         price = thePrice;
30     }
31
32     // Method to compute the range - how far the car can go on a full tank
33     public int range()
34     {
```

Scope – the Bike class

make refers to a variable box in a Bike object

It will refer to a different thing depending which object it is (every Bike object will have its own box)



That's why you can only access variables (or methods) defined in Bike using an actual Bike object (eg myBike.make)

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      public String make = "";
8      public String model = "";
9      public int engineSize;
10     public int fuelCapacity;
11     public int milesPerLitre;
12
13     // Properties that vary for each individual car
14     public int year;
15     public int price;
16
17     // Constructor
18     public Bike()
19     {
20         // nothing to do here - just use the built-in values
21     }
22
23     // Constructor to make particular bikes
24     public Bike(String theMake, String theModel, int theYear, int thePrice)
25     {
26         make = theMake;
27         model = theModel;
28         year = theYear;
29         price = thePrice;
30     }
31
32     // Method to compute the range - how far the car can go on a full tank
33     public int range()
```

Scope – the Bike class

Now look at the second constructor

This has variables for each argument – theMake, theModel, theYear, thePrice

These are called local variables

The boxes for these variables only exist while this method is running (and so the name only makes sense between its curly brackets)

We use them to assign theMake to make (which exists for the whole class definition)

We say theMake has local scope, while make has class scope

If the name theMake is used in another method, it refers to a completely separate box

```
23    // Constructor to make particular bikes
24    public Bike(String theMake, String theModel, int theYear, int thePrice)
25    {
26        make = theMake;
27        model = theModel;
28        year = theYear;
29        price = thePrice;
30    }
```

Scope and curly brackets

- Notice how the two definitions of scope we just saw used curly brackets { … } to show where a variable could be used:
  - Class scope applies to the entire { … } block following the class header
  - Local scope applies to the entire { … } block following the method header
- In fact whenever we use curly brackets in Java, we create a scope – a block of code which can limit what a name refers to
- We have seen such blocks in a few other places already:
  - Loops
  - If statements
  - Switch statements

Local scope

- We saw that variables declared as arguments to a method have local scope in the method body
- We can also declare new variables inside the method body, and these also have local scope (only exist while the body code is running)
- In addition, we can declare variables in any block in a loop, if or switch statement, and they have local scope in that block
- Just to bend your mind a little further:
  - Variables declared in a loop header have local scope in the loop body
  - You can create blocks within blocks, and have variables with local scope across the whole outer block, or just the inner block

Local scope examples

```
for (String name: names) {
    String message = "hello " + name;
    System.out.println(message);
}
```

- name is declared in the loop header
- message is declared in the loop body block
- Both are local to the for loop body
- They don't exist in the code before or after the loop

```
if (code==UP) {
    String message = "Going up";
    model.process(message);
} else {
    String message = "unknown code";
    model.process(message);
}
```

- Each part of the if statement has a separate block
- The two local variables message are completely independent of each other (to Java, using the same name is just a coincidence)

Scope and other entities

- Scope applies to other named things in Java as well as variables – in particular, methods and classes
- Methods are a bit like variable declarations – they need a type (the return type) and a name, they can also have ‘public’ attached.
- But at their core there is a similar thing going on – associated a name with a thing (a piece of code) and a type.
- The same is also true for classes.
- In fact, it is possible to create local methods and classes (within blocks) though it is quite an advanced programming technique

Visibility

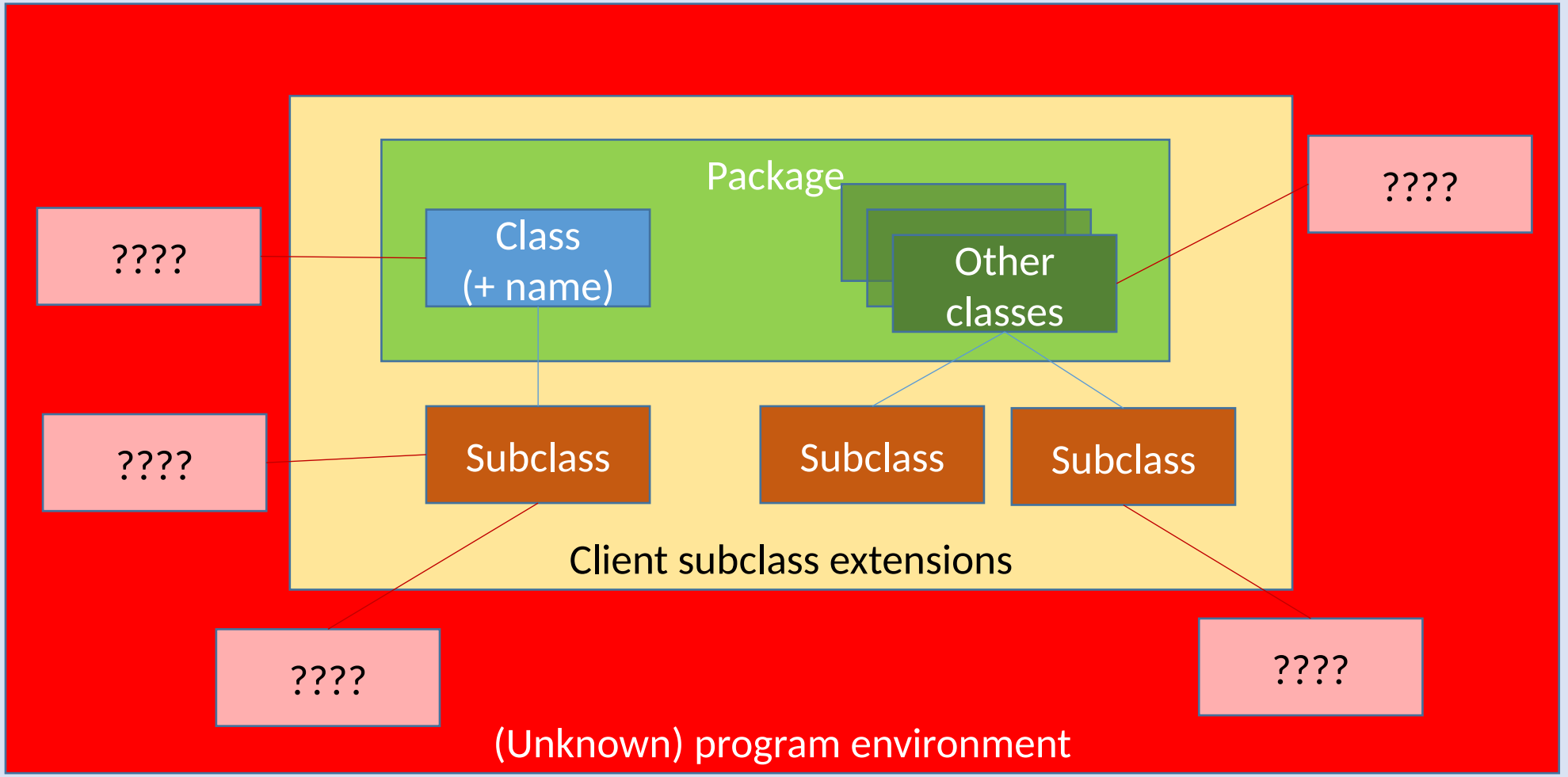
A software development scenario

- You and friends decide to write a game to sell to other developers (your clients)
- You each take responsibility for one class – Model, View and Controller
- In your own class you have some stuff which you want to be free to change later – you don't want ANYONE to rely on it (not even your team mates)
- Among you and your friends (the development team), there is stuff that you share to make the game work, but you don't want to share with anyone else
- Your clients can add features to the game, and have a contract with you so you can trust them, but they mustn't accidentally share or break something in your code.
- Your clients are allowed to sell/give your code as part of their product to people you have no contract with, or control over. You want to stop them breaking your code, accidentally or deliberately (and damaging your reputation).

Visibility

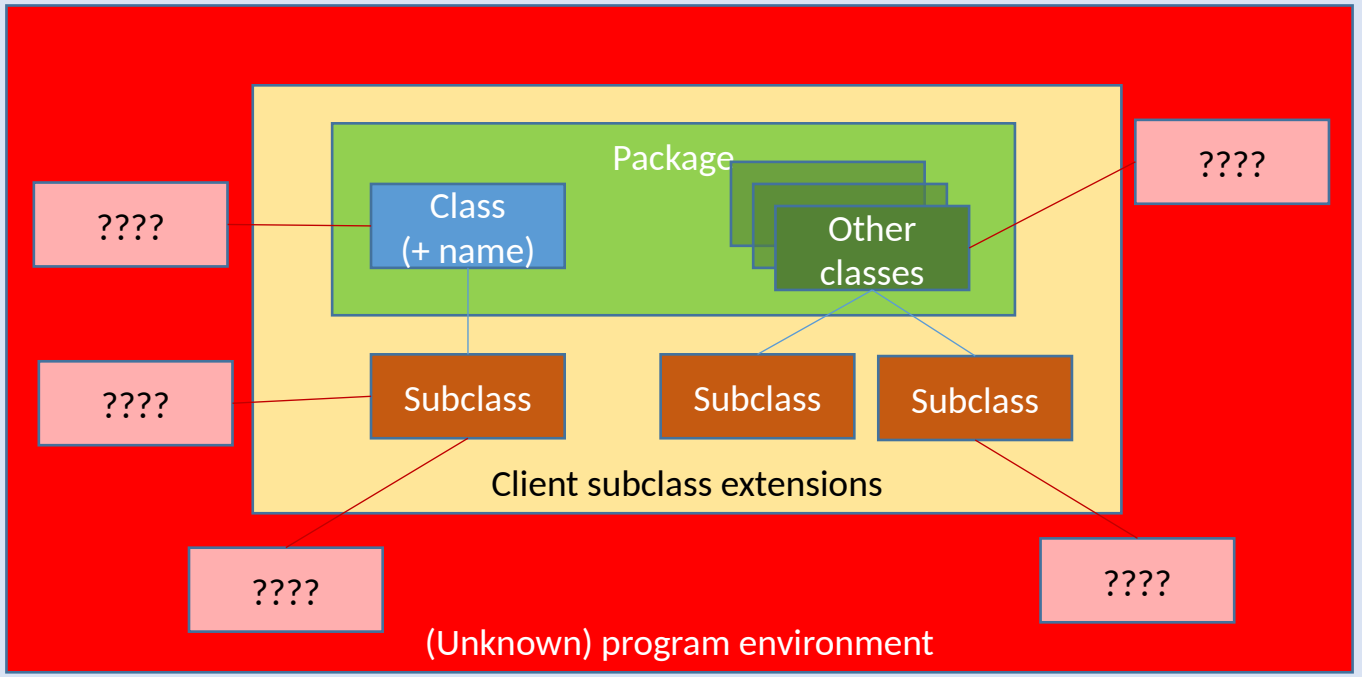
- Visibility is about which other people's code can 'see' the names in your code and so access your classes, variables or methods
- If someone can see it, then they can access it and possibly modify it
- We control visibility, to make sure our code can be seen, and possibly modified, only by the people we want.
- Java doesn't actually do this by referring to different groups of people, but by having a particular program architecture which can be mapped onto these groups, and controlling access to parts of that architecture
- Access is controlled by access specifier keywords

Java program architecture



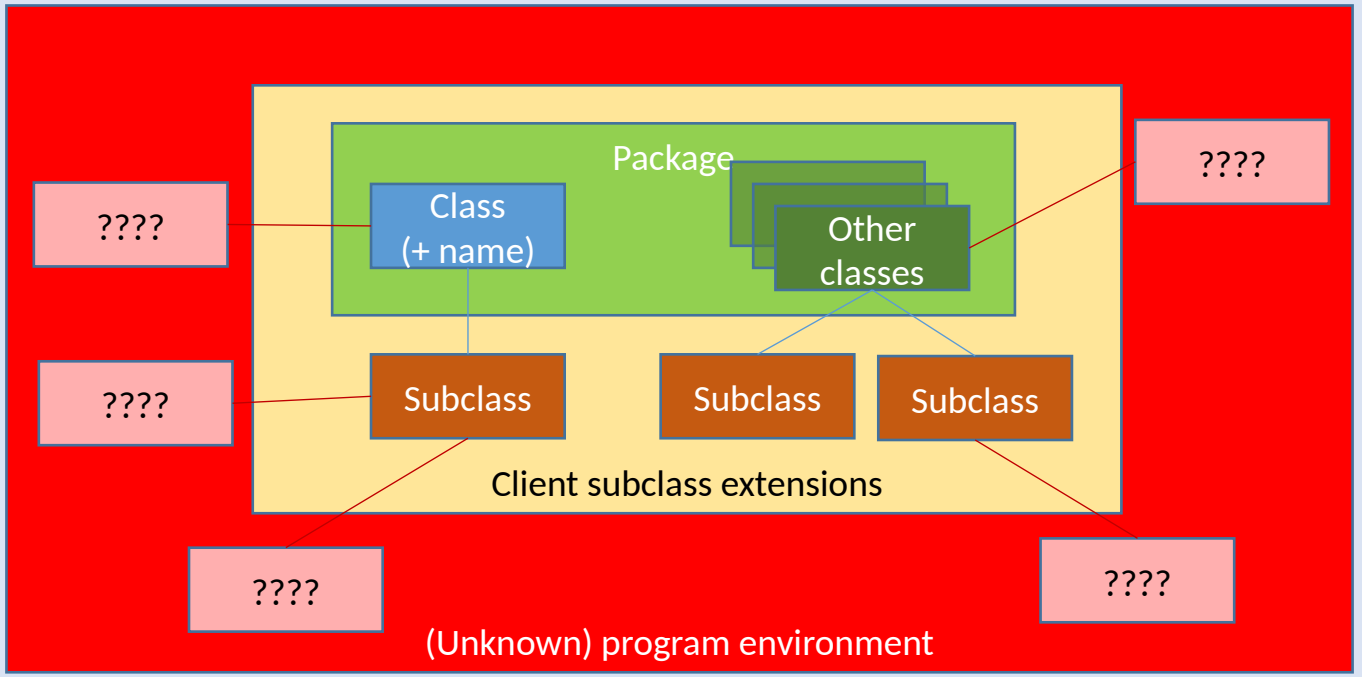
Java program architecture

- Class – the class containing the name, whose access you want to control
- Package – classes that work with the class to make a functional unit
- Subclasses intended to add features to the package
- Program – other unknown code that use the (extended) package



'Trust' architecture

- Class – you have complete control
- Package – your team, working together
- Subclasses – clients, working cooperatively with you
- Program – unknown agents who you don't want to trust



Access specifiers

- Many of the variable and method declarations we have used so far have included the word `public`
- Occasionally you might have seen `private`, or nothing at all
- These words are called access specifiers and control what other parts of a program can ‘see’ the variable.
- This mainly relates to variables or methods with class scope, because local scope (in methods, for loops, if statements etc.) restricts availability of variables anyway

Visibility table

Here is a table of all the possible access specifiers and how they control visibility:

	Class	Package	Subclass	Program
public	yes	yes	yes	yes
protected	yes	yes	yes	no
(no specifier)	yes	yes	no	no
private	yes	no	no	no

Columns – different components of the program

- class – the class the name is declared in (written by ‘you’)
- package – the collection of classes that make up this component/ library etc (written by ‘your development team’)
- subclass – code that is using your code (by subclassing it) and needs access to things, but may be external (written by ‘your clients’)
- program – the whole program your code is being used in (it could be written by ‘anyone’, you have no control over it)

Rows - access specifiers

- public – visible to code written by anyone
- protected – visible to code written by your team (other classes in the same package) and your clients on a ‘need to know’ basis (subclasses)
- (no specifier) – only visible to code written by your team (in same package)
- private – only visible to code in the current class

Visibility table – 'trust' version

	You	Your team	Your clients	Anyone
public	yes	yes	yes	yes
protected	yes	yes	yes	no
(no specifier)	yes	yes	no	no
private	yes	no	no	no

Encapsulation

Encapsulation – the basic idea

- The basic idea in encapsulation is that you keep all the implementation details of your code hidden
- All you provide to the outside world are method calls for accessing things - in particular you never provide direct access to variables, (unless they are final variables – see below)
- That way, other people can't do anything naughty with your code, and you are able to change the way the code works without affecting anyone who uses it

Encapsulation – the Bike class

Back to Bike again!

Everything is marked as public – ouch!!!

Anyone can change the details of it – eg set the milesPerLitre to 5

What can we do to stop this?

Look at every instance of public and decide if it is right

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      public String make = "";
8      public String model = "";
9      public int engineSize;
10     public int fuelCapacity;
11     public int milesPerLitre;
12
13     // Properties that vary for each individual car
14     public int year;
15     public int price;
16
17     // Constructor
18     public Bike()
19     {
20         // nothing to do here - just use the built-in values
21     }
22
23     // Constructor to make particular bikes
24     public Bike(String theMake, String theModel, int theYear, int thePrice)
25     {
26         make = theMake;
27         model = theModel;
28         year = theYear;
29         price = thePrice;
30     }
31
32     // Method to compute the range - how far the car can go on a full tank
33     public int range()
```

Encapsulation – variables shared with subclasses

The first block of variables need to be visible, but only by subclasses – make them protected

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      protected String make = "";
8      protected String model = "";
9      protected int engineSize;
10     protected int fuelCapacity;
11     protected int milesPerLitre;
12
13     // Properties that vary for each individual car
14     public int year;
15     public int price;
16
17     // Constructor
18     public Bike()
19     {
20         // nothing to do here - just use the built-in values
21     }
22
23     // Constructor to make particular bikes
24     public Bike(String theMake, String theModel, int theYear, int thePrice)
25     {
26         make = theMake;
27         model = theModel;
28         year = theYear;
29         price = thePrice;
30     }
31
32     // Method to compute the range - how far the car can go on a full tank
33     public int range()
```

Encapsulation – private variables

The next two are variables we might want to allow public access to, but only under our control.

Set them private, but create public getters and setters for them.

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      protected String make = "";
8      protected String model = "";
9      protected int engineSize;
10     protected int fuelCapacity;
11     protected int milesPerLitre;
12
13     // Properties that vary for each individual car
14     private int year;
15     private int price;
16
17     public int getYear() { return year; };
18     public void setYear(int theYear) { year = theYear; }
19
20     public int getPrice() { return price; };
21     public void setPrice (int thePrice) { if (thePrice > 0) price = thePrice; }
22
23     // Constructor
24     public Bike()
25     {
26         // nothing to do here - just use the built-in values
27     }
28
29     // Constructor to make particular bikes
30     public Bike(String theMake, String theModel, int theYear, int thePrice)
31     {
32         make = theMake;
33         model = theModel;
34         year = theYear;
```

Encapsulation – getters and setters

Getters and setters are methods which just return (get) or set the value of a variable.

They generally have standard names – getYear, setYear etc.

They let us control access to the variables, for example we can ensure that price will always be > 0, (unless we change it in this class itself).

We can also change exactly how information is stored – for example, store prices in dollars instead of pounds, but make getPrice and setPrice do an automatic conversion

```
3  public class Bike
4
5  {
6      // Properties of a particular kind of bike
7      protected String make = "";
8      protected String model = "";
9      protected int engineSize;
10     protected int fuelCapacity;
11     protected int milesPerLitre;
12
13     // Properties that vary for each individual car
14     private int year;
15     private int price;
16
17     public int getYear() { return year; };
18     public void setYear(int theYear) { year = theYear; }
19
20     public int getPrice() { return price; };
21     public void setPrice (int thePrice) { if (thePrice > 0) price = thePrice; }
22
23     // Constructor
24     public Bike()
25     {
26         // nothing to do here - just use the built-in values
27     }
28
29     // Constructor to make particular bikes
30     public Bike(String theMake, String theModel, int theYear, int thePrice)
31     {
32         make = theMake;
33         model = theModel;
34         year = theYear;
```

Encapsulation – the Bike class

We could also review the other methods – make sure they should all be public, but they probably should in this case

We might consider renaming ‘range’ to getRange, to make it ‘look like’ a variable. (but without a setRange version)

(Having a getter but no setter is a way of making a variable ‘read-only’)

```
18
19      public int range()
20      {
21          int theRange = fuelCapacity*milesPerLitre;
22          return theRange;
23      }
24
```

Extras

A couple of slightly more advanced topics

Visibility and the final keyword

- The access specifiers allow different parts of the program to see a name
- Used on their own, if you can see a name, you can also modify the object it refers to.
- 'Modify' can mean different things:
  - Variable – it means you can change its value
  - Method – it means you can override it
  - Class – it means you can create a subclass
- Sometimes we want others to see a value but not modify it
- We can use the keyword final to mark this:
  - `public final double PI = 3.14159;`
    visible but can't be changed
  - `public final void run() { … }`
    visible but can't be overridden in a subclass
  - `public final class KiaPicanto { … }`
    visible but can't be subclassed

static variables and methods

- Class variables and methods are visible in a whole class and associated with individual instances of that class
- Sometimes we want variables or methods which are just associated with the class, not each particular instance.
- We can use the keyword static to mark this:
  - `public static String classMessage =“Welcome to CI104”;`
    A variable which is associated with a class, not a particular object, and can be accessed using the class name (eg Student.classMessage)
  - `public static void main() { … }`
    A method that can be run without an object, using the classname (eg Car.main())
- Static methods can only reference static variables and other static methods (but ordinary methods can access ordinary and static variables and methods)

Lab exercises
Week 2.03

Week 2.03 Lab work - Encapsulation

- Go back to the TrustyCars lab from last week and try and use the encapsulation techniques discussed in this week's lecture
- For example, make the instance variables in the Car or Bike classes private and add getter and setter methods for them which are public
- Then update the 'description' methods to use the getter methods instead of the variables directly .

Project work

- The solution lab for the Breakout exercise is published this week. This is the baseline if you choose Breakout for your project. (If you have gone further, or have a different solution, don’t worry, you can carry on with your own version.)
- Carry on looking at the ATM project if you need to – solution will be provided next week
- If you are interested in the more advanced Maze project, try to attend Roger’s seminar this week.

Project work - encapsulation

- Look for places to add encapsulation to Breakout or the ATM. It's always a good idea to look for data classes for this – GameObj in Breakout and BankAccount in ATM. But the other classes (especially the Models) provide opportunities too

- Remember, if you do this as part of your assessment project, it will count towards your marks. But this means tutors can't help you with the details – we can only help with the TrustyCars examples.