2020 CI401
Introduction to programming

Week 2.02
Simple inheritance

15th 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 ↓

Project preparation timeline

Week	Date (w/b)	Project	Related topic/lab exercise
2.01	08-Feb-2021	Starter projects and topics announced	Introduction to coursework
2.02	15-Feb-2021	Supported project lab work	Simple Inheritance
2.03	22-Feb-2021	Supported project lab work	Scope, Visibility and Encapsulation
2.04	01-Mar-2021	Assessment baseline starter projects	Testing - JUnit
2.05	08-Mar-2021	Independent project work	Documentation - Javadoc
2.06	15-Mar-2021	Independent project work	Collections and generic types
2.07	22-Mar-2021	Independent project work	IO: Files and Streams
2.08	19-Apr-2021	Independent project work	
2.09	26-Apr-2021	Independent project work	
		Project deadline – 30-April-2021, 3pm	

Code clinic is back!

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

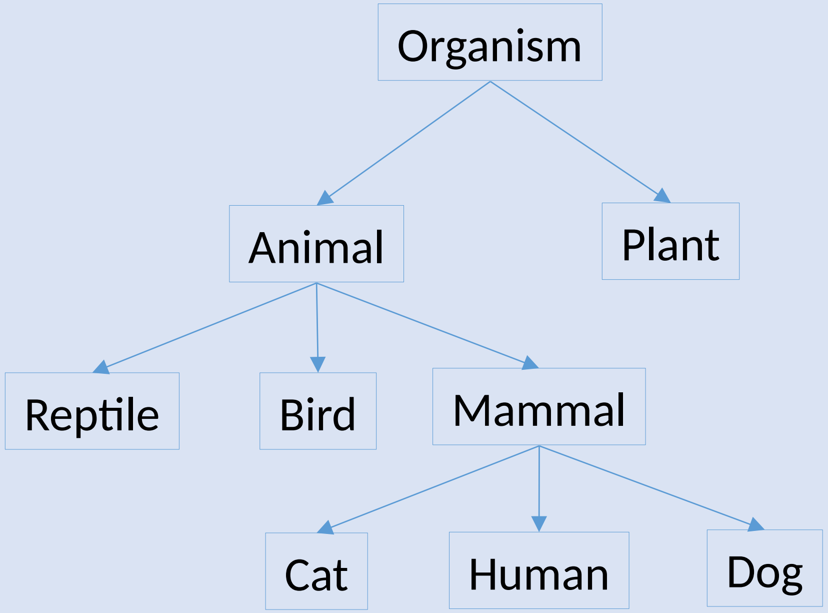
- We are also looking at providing some catch-up support in seminars from next week

Inheritance

Inheritance

- Inheritance is one of the most important features of object-oriented programming
- Inheritance allows classes to share information (variables and methods)
- This make it easier to write large programs in smaller chunks, and to test code once which is then shared in many places
- So programming becomes more modular and robust

Inheritance in the real world



Organism

Animal

Plant

Reptile

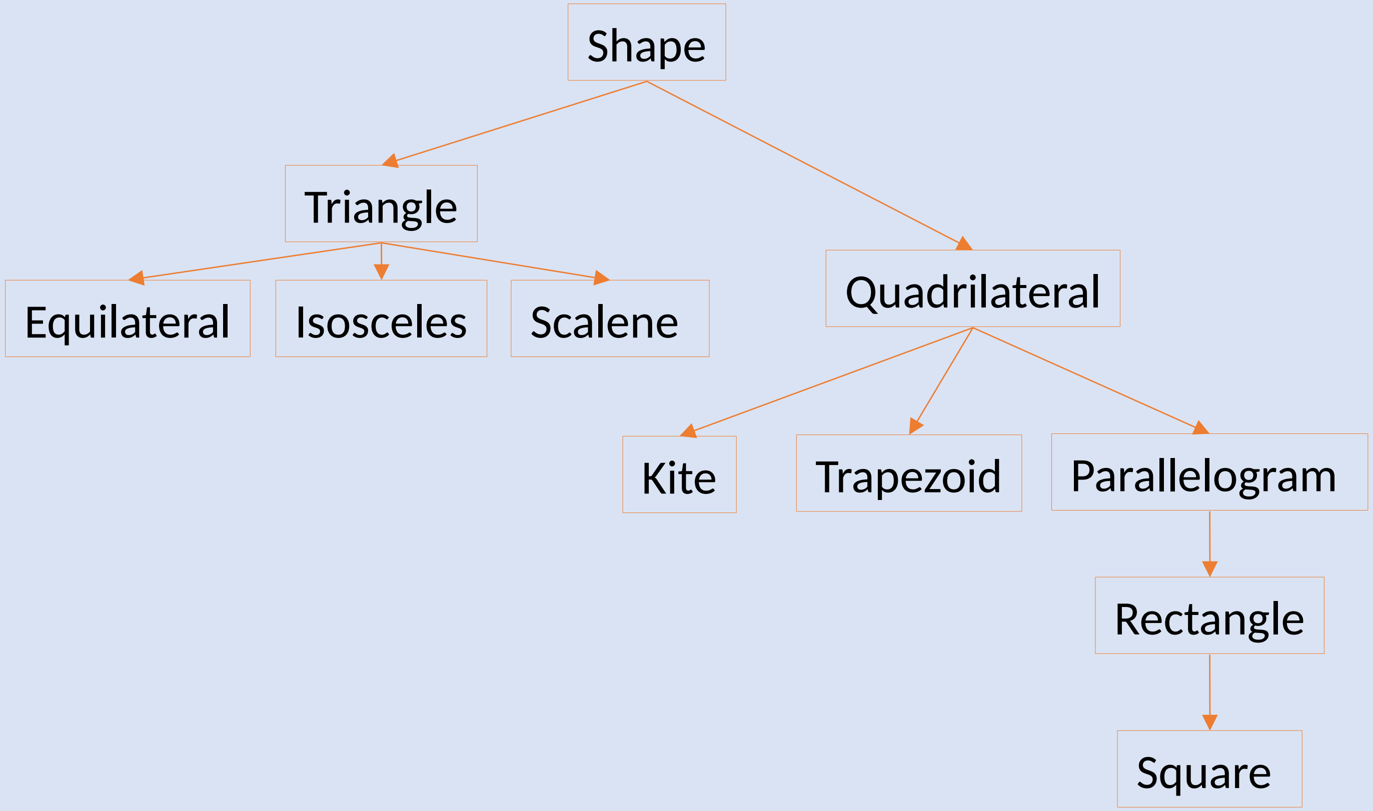
Bird

Mammal

Cat

Human

Dog



Shape

Triangle

Quadrilateral

Equilateral

Isosceles

Scalene

Kite

Trapezoid

Parallelogram

Rectangle

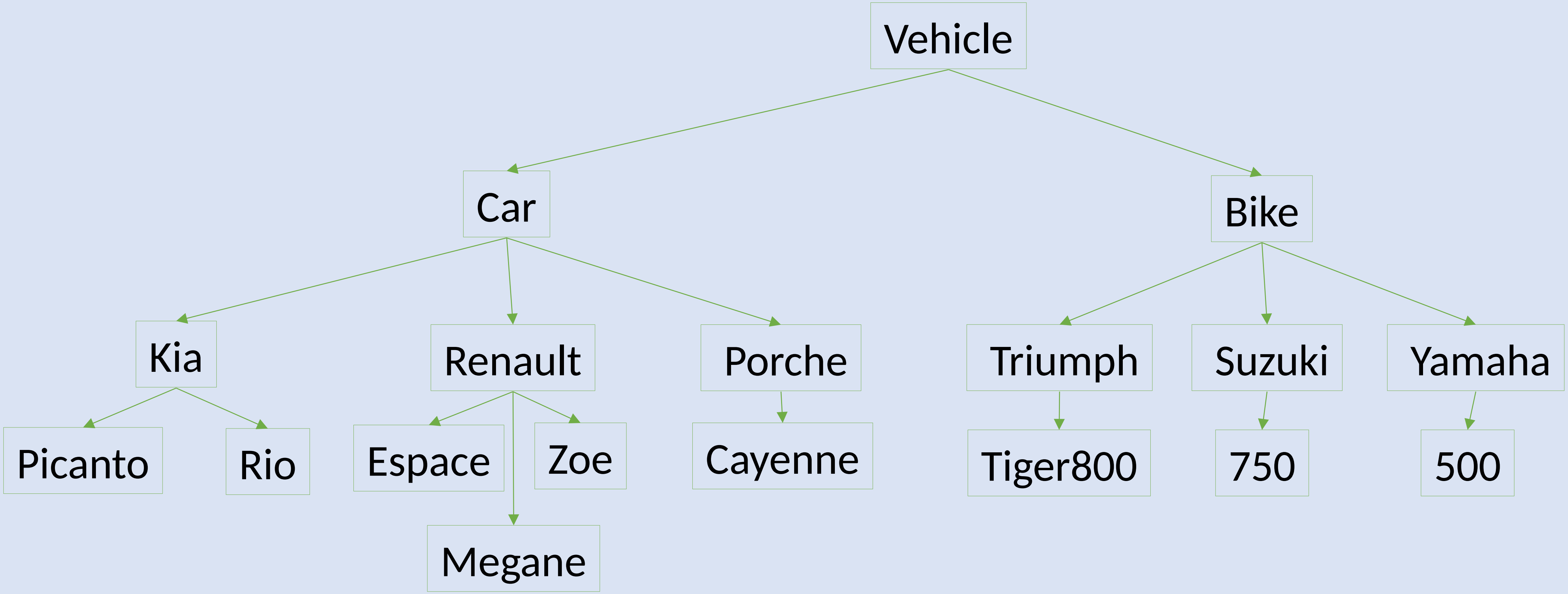
Square

- Known as is-a hierarchies, inheritance hierarchies or ontologies

Inheritance in Java

- We already use classes to represent kinds of thing
- We use inheritance to represent the links between them
- Classes inherit information from the class above them (their parent)
- They share inherited information with the classes next to them (their siblings)
- They can also add new information or change inherited information
- Classes below them (their children) inherit their information (including anything they inherit from their parent)

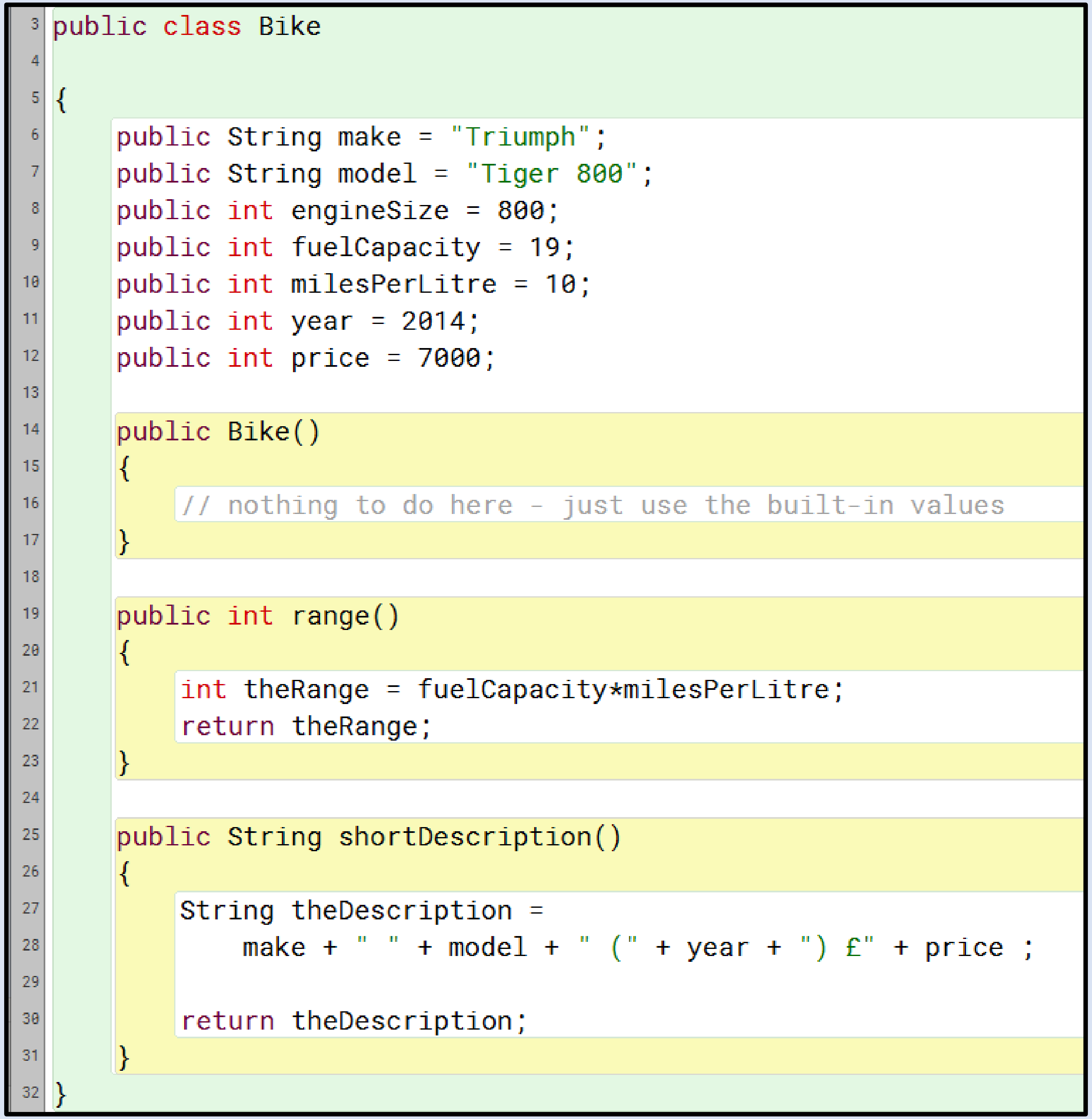
An inheritance hierarchy for vehicles



Sharing information between classes

Here’s a class for Triumph bikes

Remember, classes contain the information needed to create and work with objects



```
public class Bike

{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int year = 2014;
    public int price = 7000;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

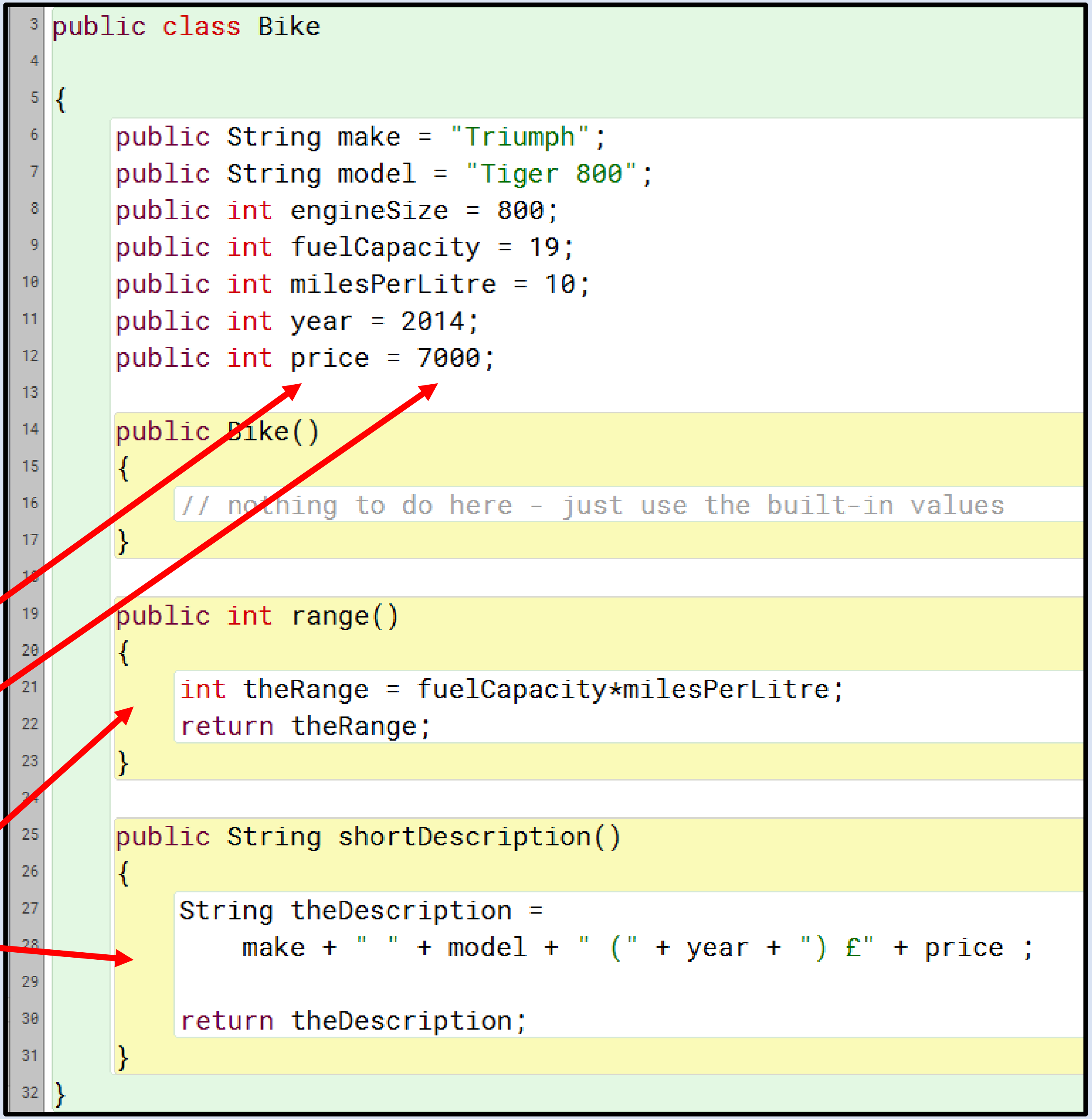
Sharing information between classes

Here’s a class for Triumph bikes

Remember, classes contain the information needed to create and work with objects

This includes

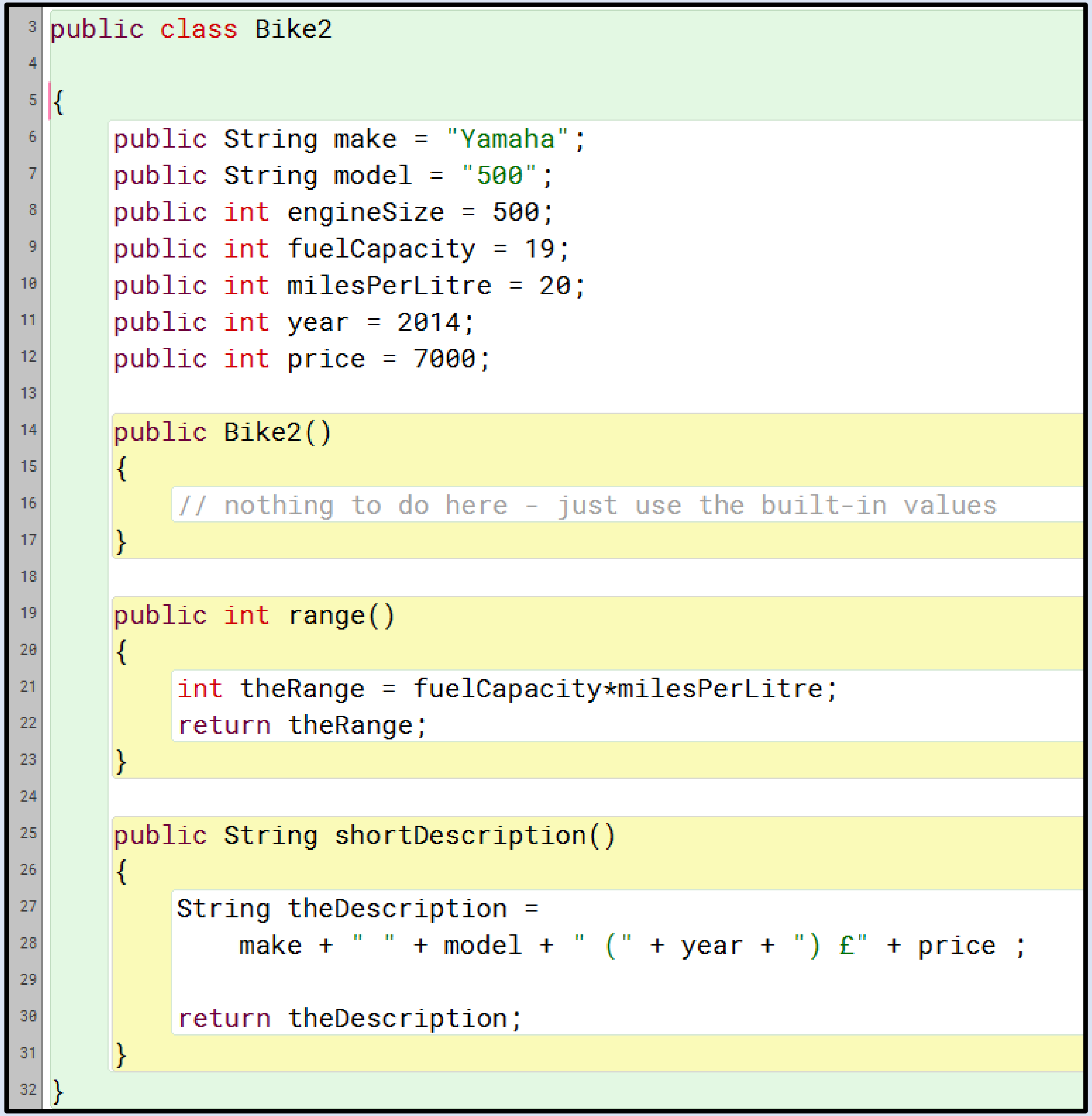
- Variables (‘state’)
- Initial values
- Methods (‘behaviour’)



```
public class Bike

{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int year = 2014;
    public int price = 7000;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

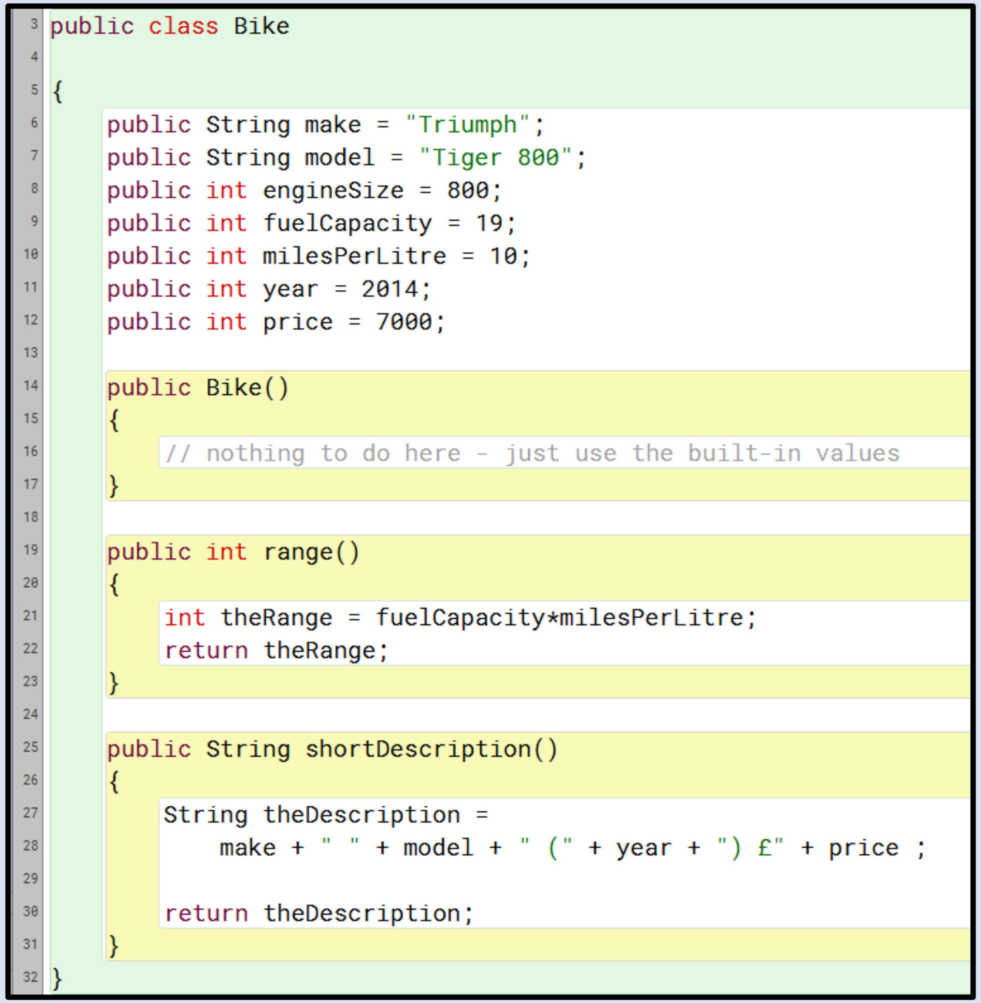
Sharing information between classes

Now let’s write another bike class for Yamahas

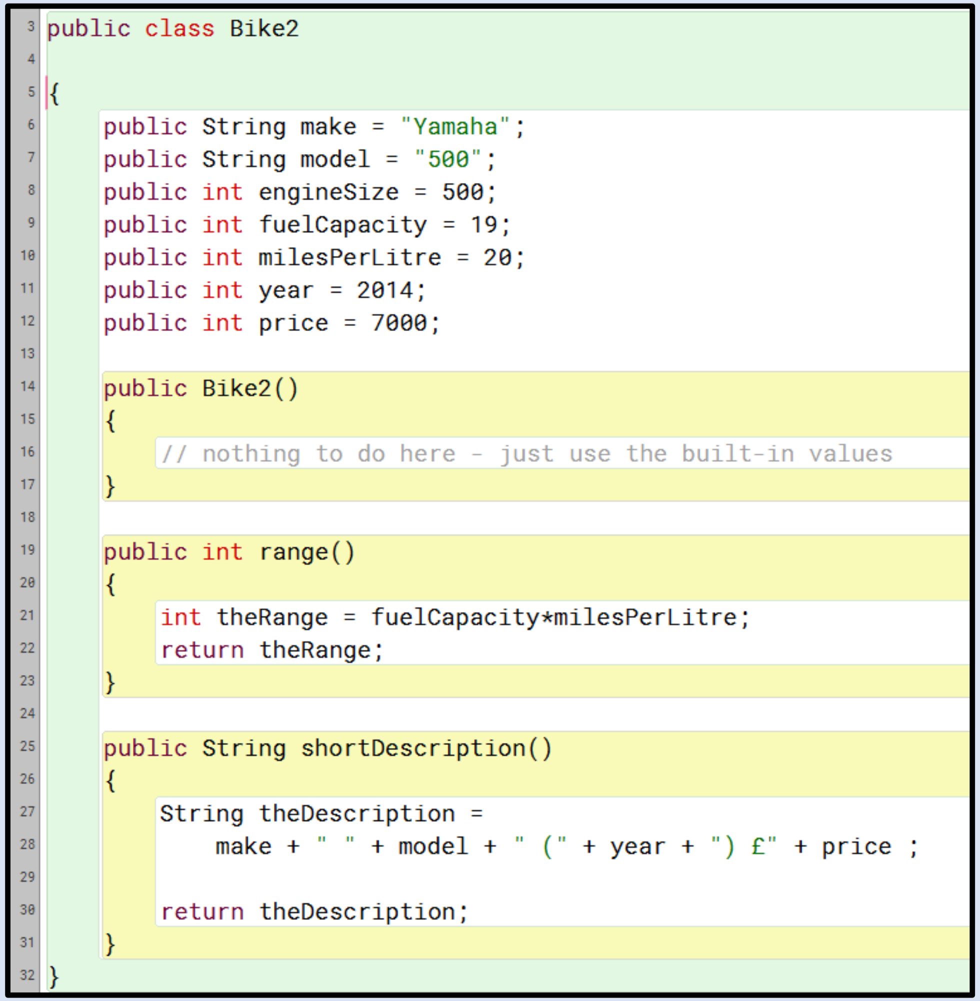


```
public class Bike2

{
    public String make = "Yamaha";
    public String model = "500";
    public int engineSize = 500;
    public int fuelCapacity = 19;
    public int milesPerLitre = 20;
    public int year = 2014;
    public int price = 7000;

    public Bike2()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Spot the difference …

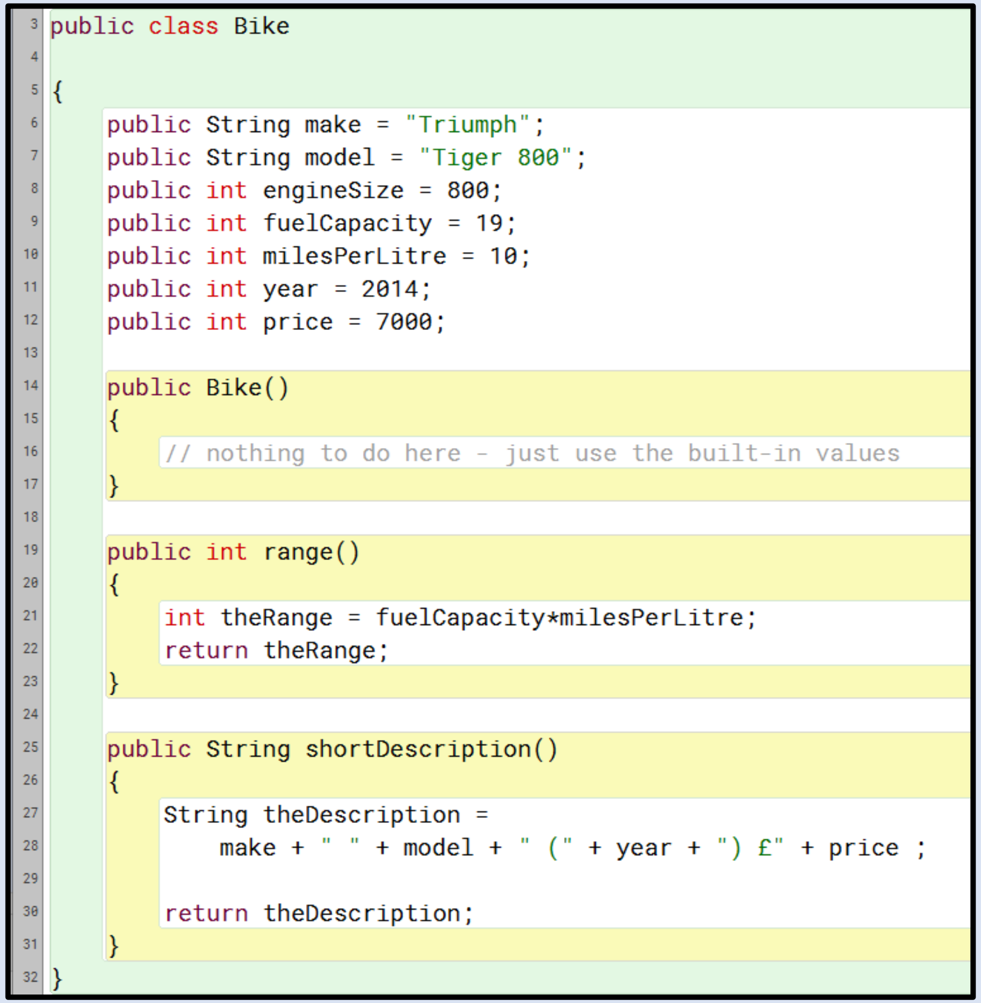


```
public class Bike

{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int year = 2014;
    public int price = 7000;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

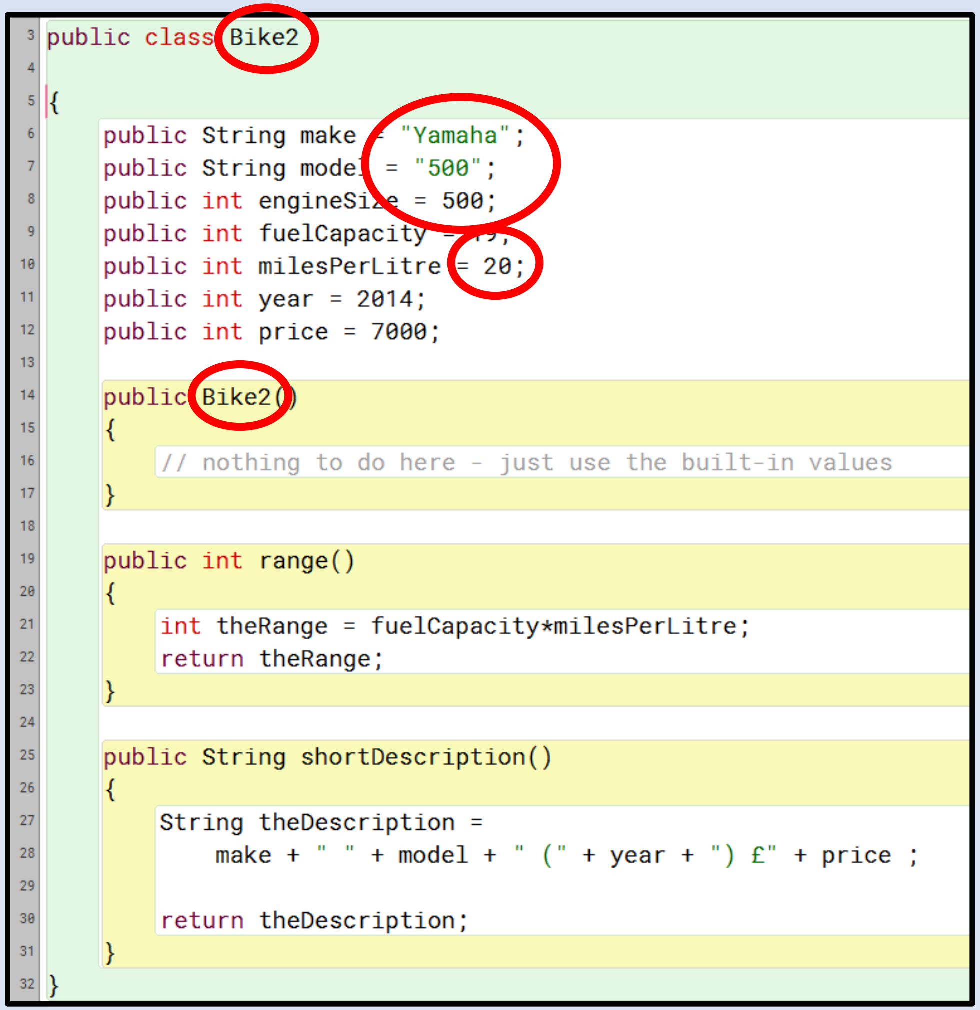


```
public class Bike2

{
    public String make = "Yamaha";
    public String model = "500";
    public int engineSize = 500;
    public int fuelCapacity = 19;
    public int milesPerLitre = 20;
    public int year = 2014;
    public int price = 7000;

    public Bike2()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Spot the difference …

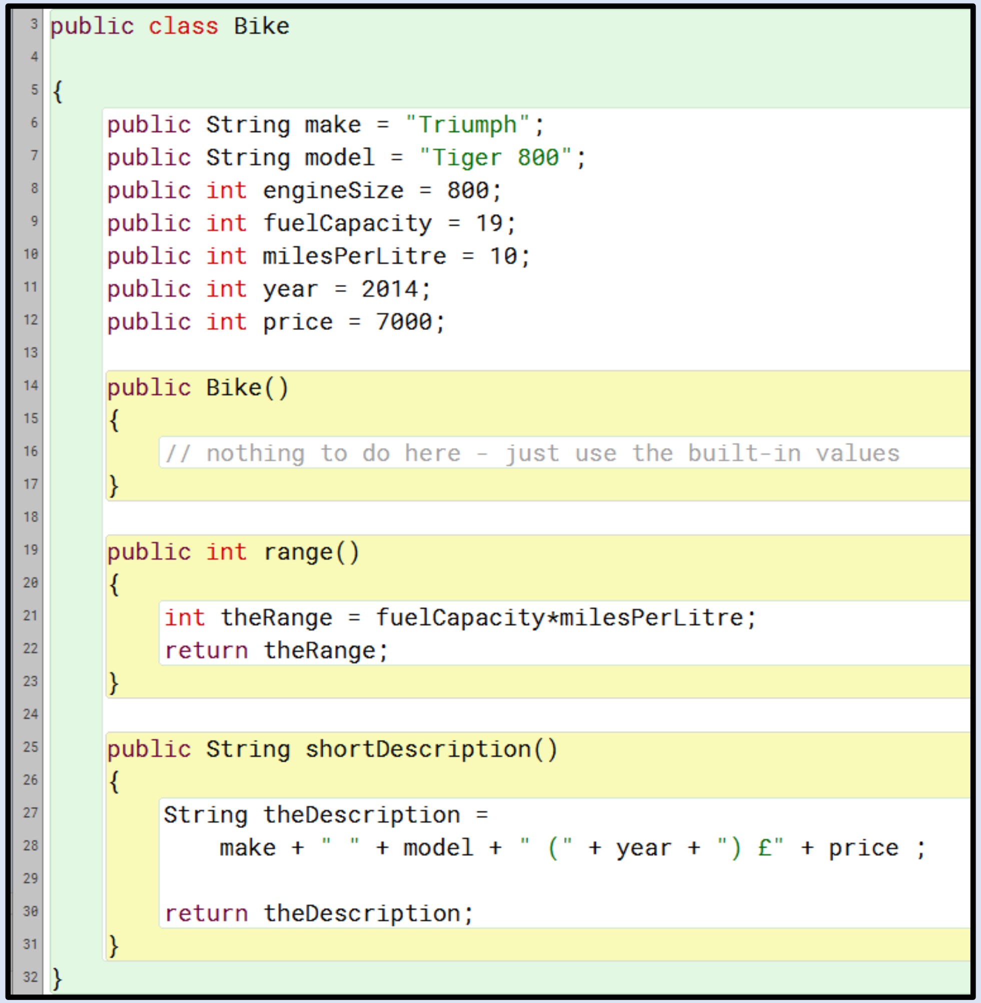


```java
public class Bike

{
	public String make = "Triumph";
	public String model = "Tiger 800";
	public int engineSize = 800;
	public int fuelCapacity = 19;
	public int milesPerLitre = 10;
	public int year = 2014;
	public int price = 7000;

	public Bike()
	{
		// nothing to do here - just use the built-in values
	}

	public int range()
	{
		int theRange = fuelCapacity*milesPerLitre;
		return theRange;
	}

	public String shortDescription()
	{
		String theDescription =
			make + " " + model + " (" + year + ") £" + price ;

		return theDescription;
	}
}
```

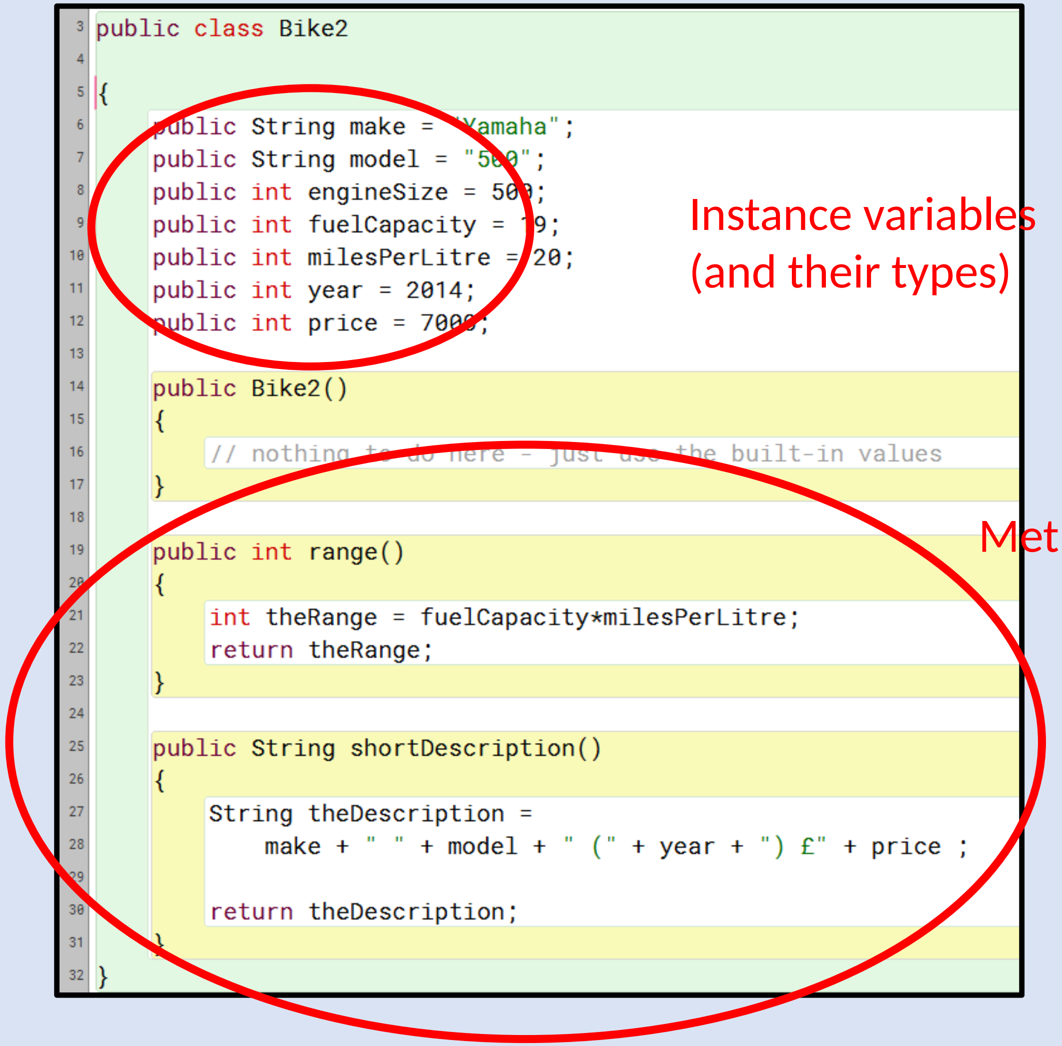


```java
public class Bike2

{
	public String make = "Yamaha";
	public String model = "500";
	public int engineSize = 500;
	public int fuelCapacity = 19;
	public int milesPerLitre = 20;
	public int year = 2014;
	public int price = 7000;

	public Bike2()
	{
		// nothing to do here - just use the built-in values
	}

	public int range()
	{
		int theRange = fuelCapacity*milesPerLitre;
		return theRange;
	}

	public String shortDescription()
	{
		String theDescription =
			make + " " + model + " (" + year + ") £" + price ;

		return theDescription;
	}
}
```

Spot the similarity …



```java
public class Bike

{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int year = 2014;
    public int price = 7000;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```



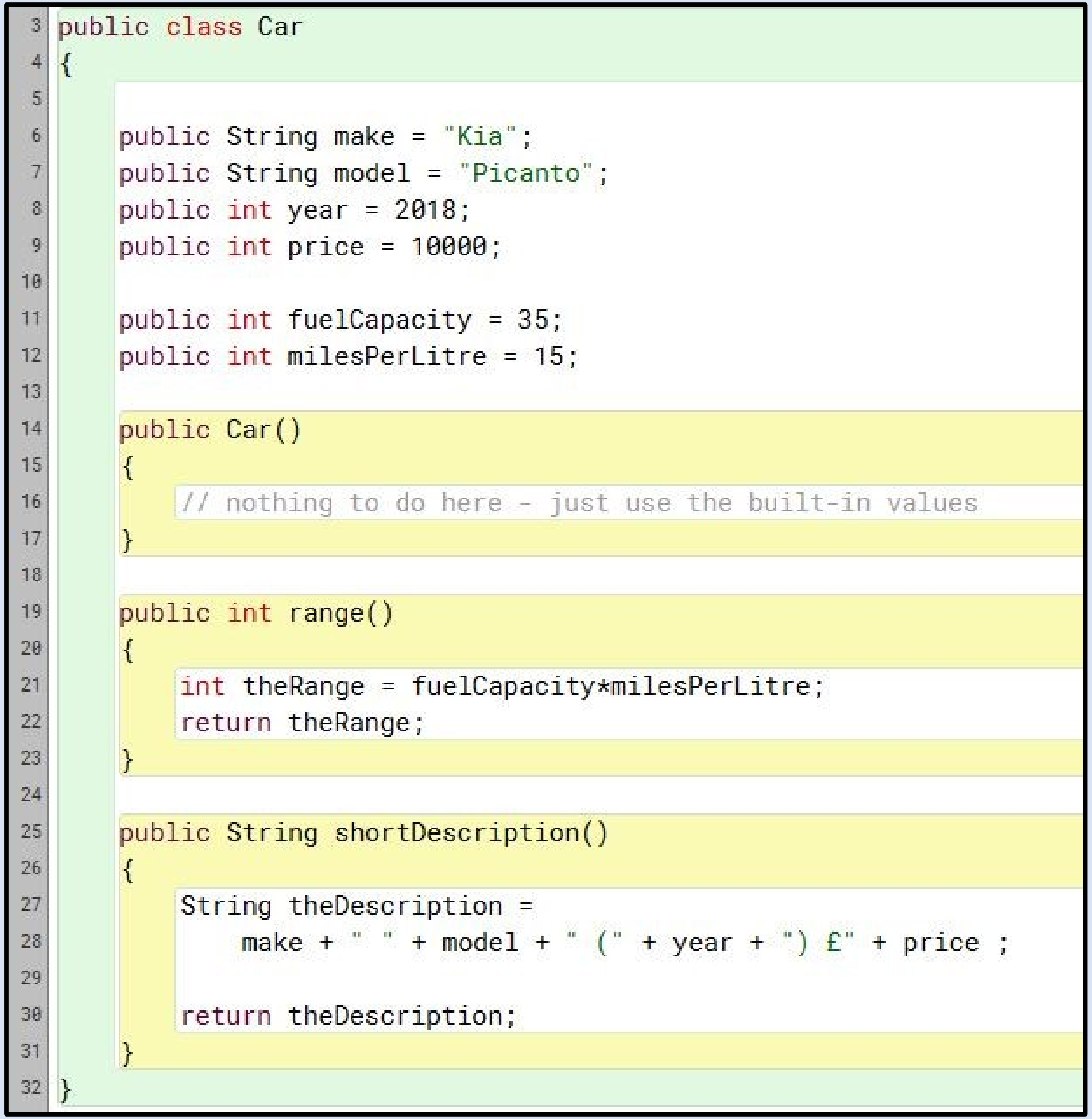
```java
public class Bike2

{
    public String make = "Yamaha";
    public String model = "500";
    public int engineSize = 500;
    public int fuelCapacity = 19;
    public int milesPerLitre = 20;
    public int year = 2014;
    public int price = 7000;

    public Bike2()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Instance variables
(and their types)

Methods

Sharing information between classes

Even when we start writing classes for Cars, there is a lot of similarity



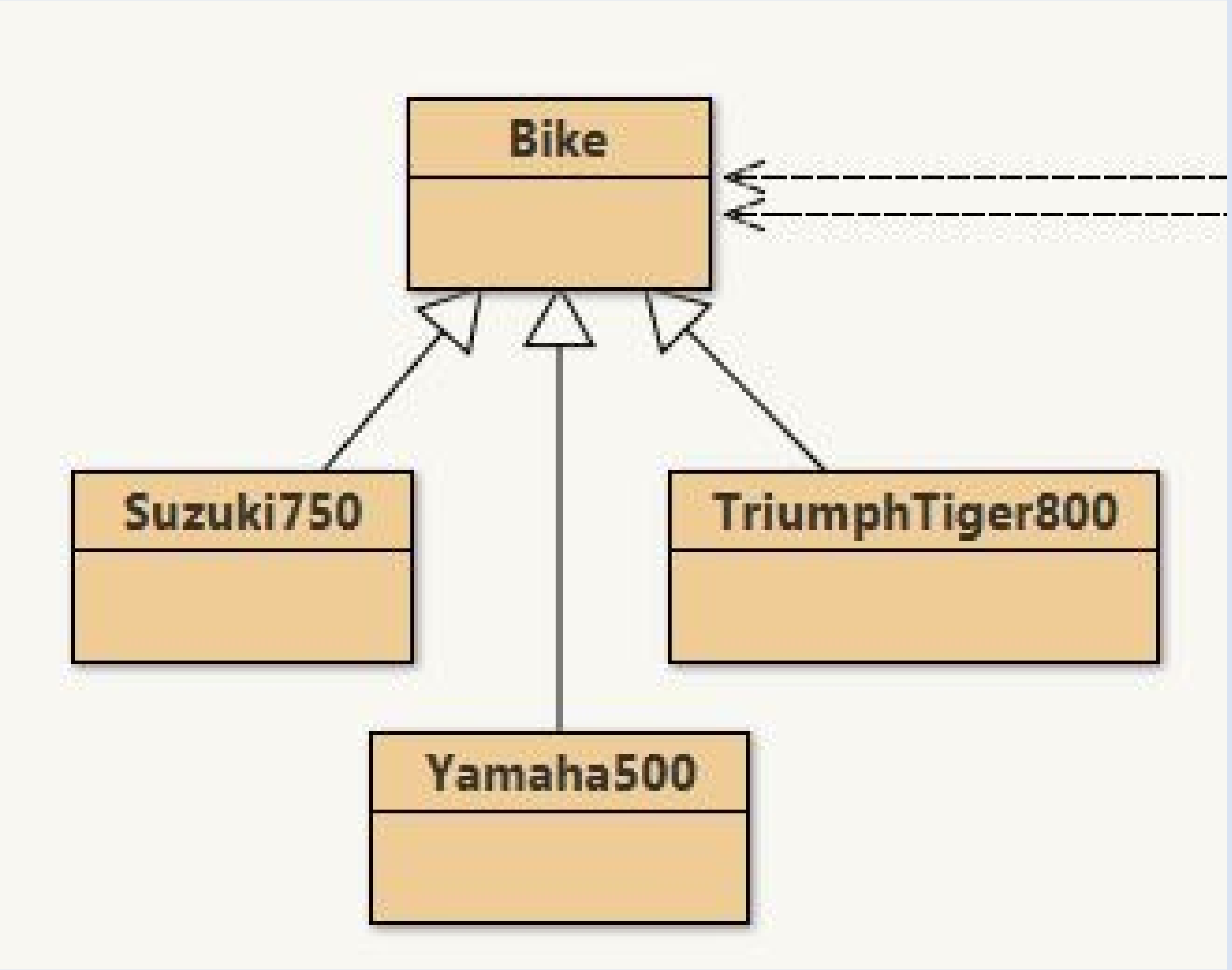
```
public class Car
{

    public String make = "Kia";
    public String model = "Picanto";
    public int year = 2018;
    public int price = 10000;

    public int fuelCapacity = 35;
    public int milesPerLitre = 15;

    public Car()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Sharing information between classes

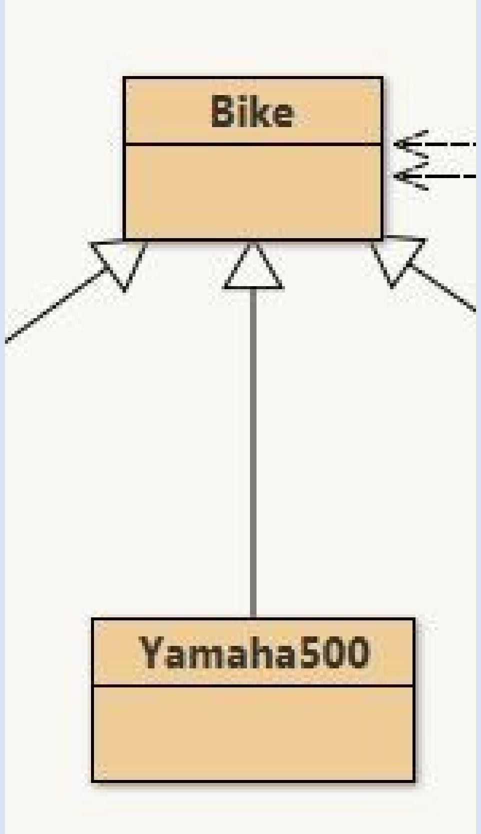
Java provides a way of sharing information between classes

This is called inheritance

Here we have three different kinds of bike, sharing information in a class called Bike



Sharing information between classes – some terminology



- The class the arrow points to is called the superclass or parent

- The link between two classes is called an inheritance link or an is-a link.
  (In BlueJ, it has an arrow head).

- The class at the beginning of the arrow is called the subclass or child.

Sharing information between classes

We say

Yamaha500 is a subclass/child of Bike

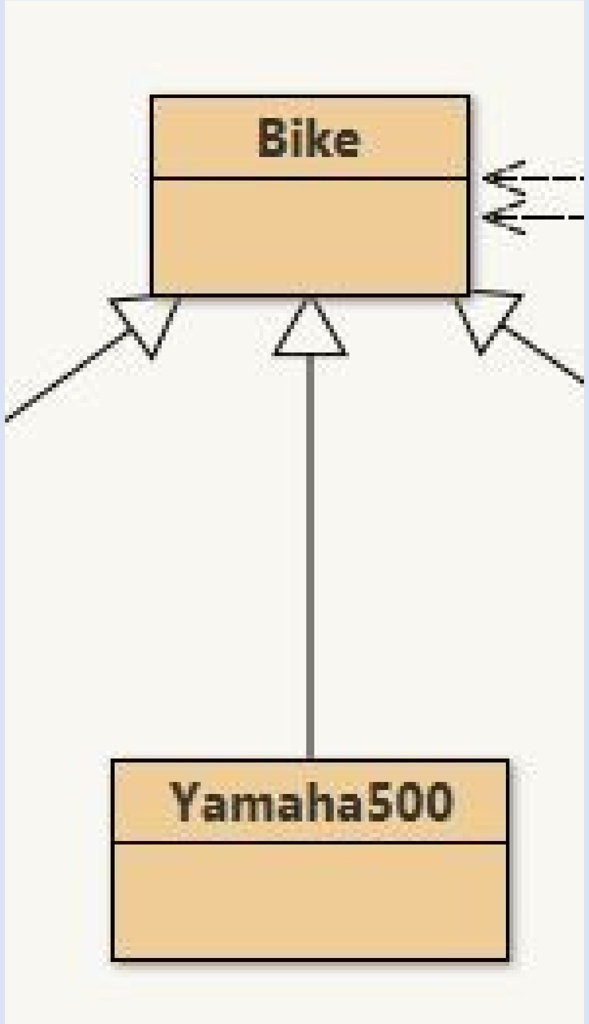
or

Bike is a superclass/parent of Yamaha500

or

Yamaha500 extends Bike

(they all mean the same thing)



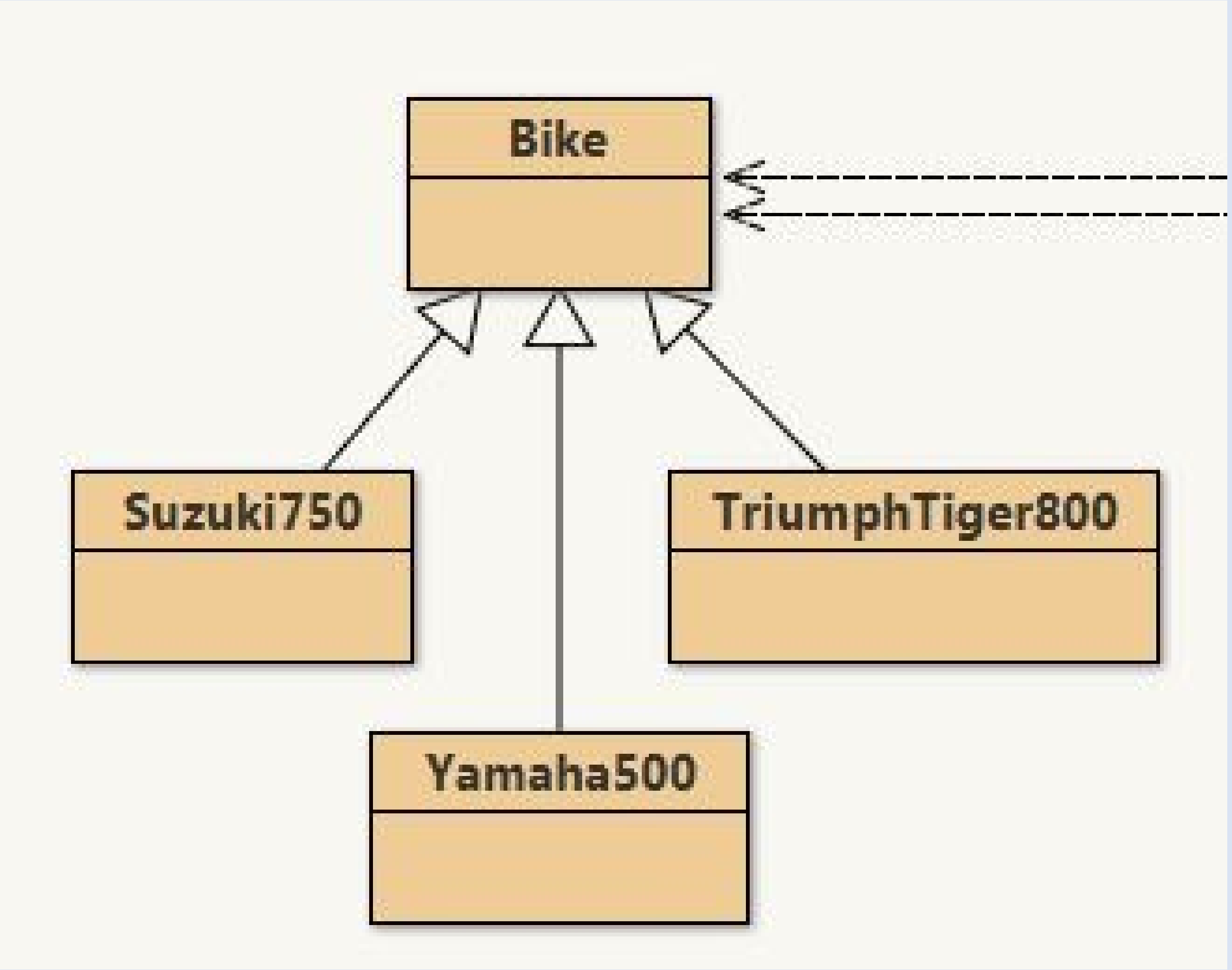
Sharing information between classes

There is one important rule about inheritance in Java:

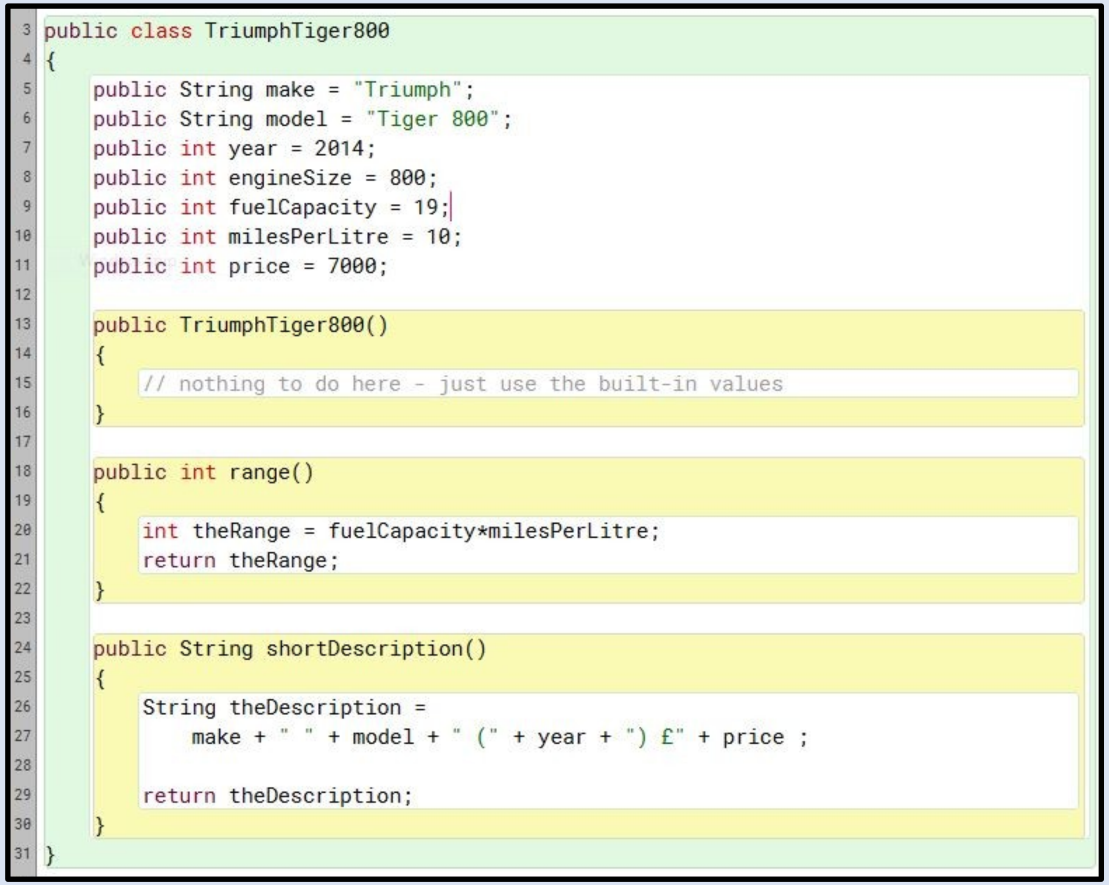
> A class can only have one superclass

(but a class can have many subclasses)

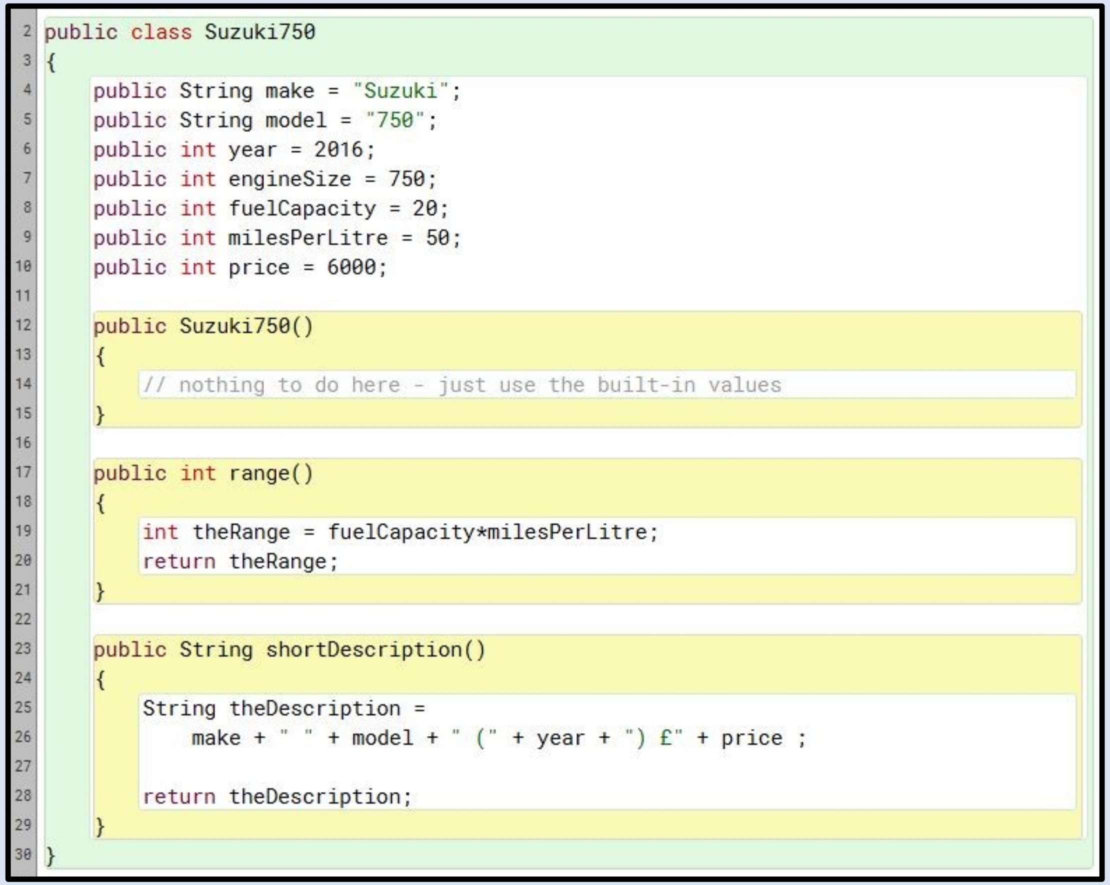
(and a class can have a superclass AND subclasses)



How does this work?

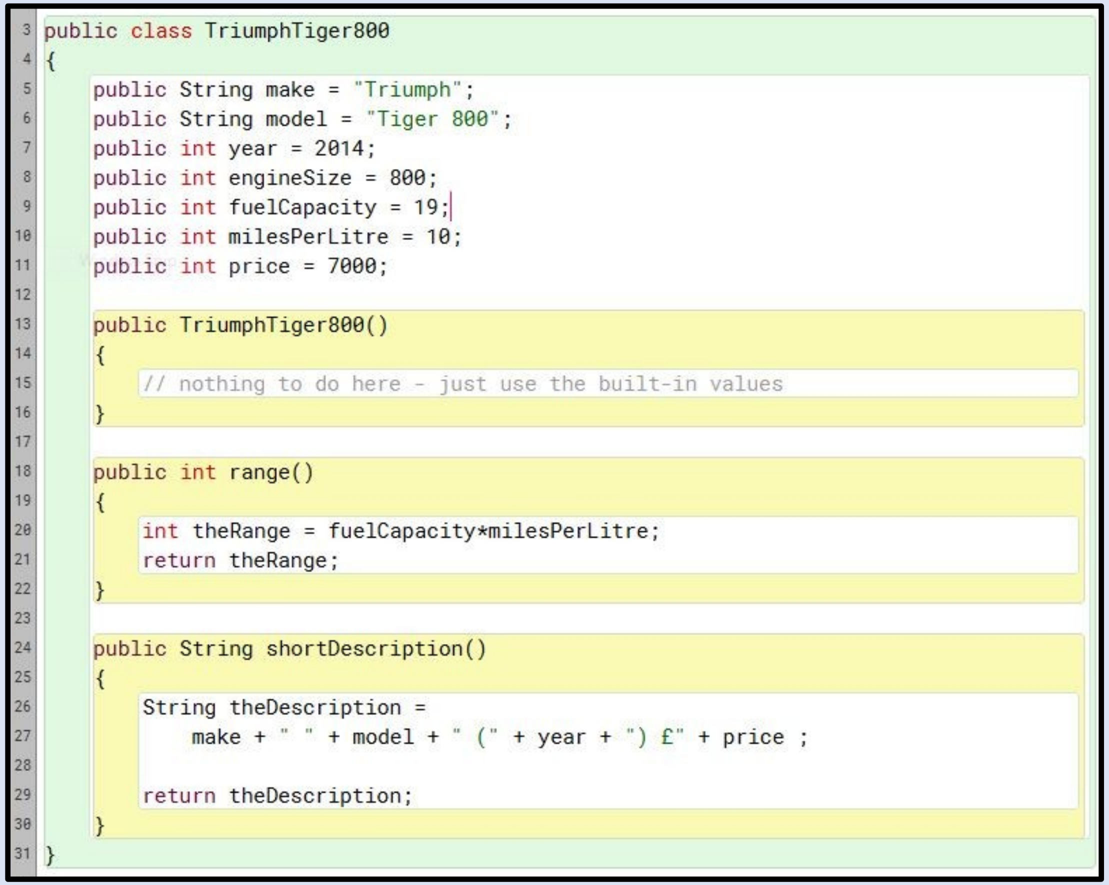


```java
public class TriumphTiger800
{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int year = 2014;
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int price = 7000;

    public TriumphTiger800()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

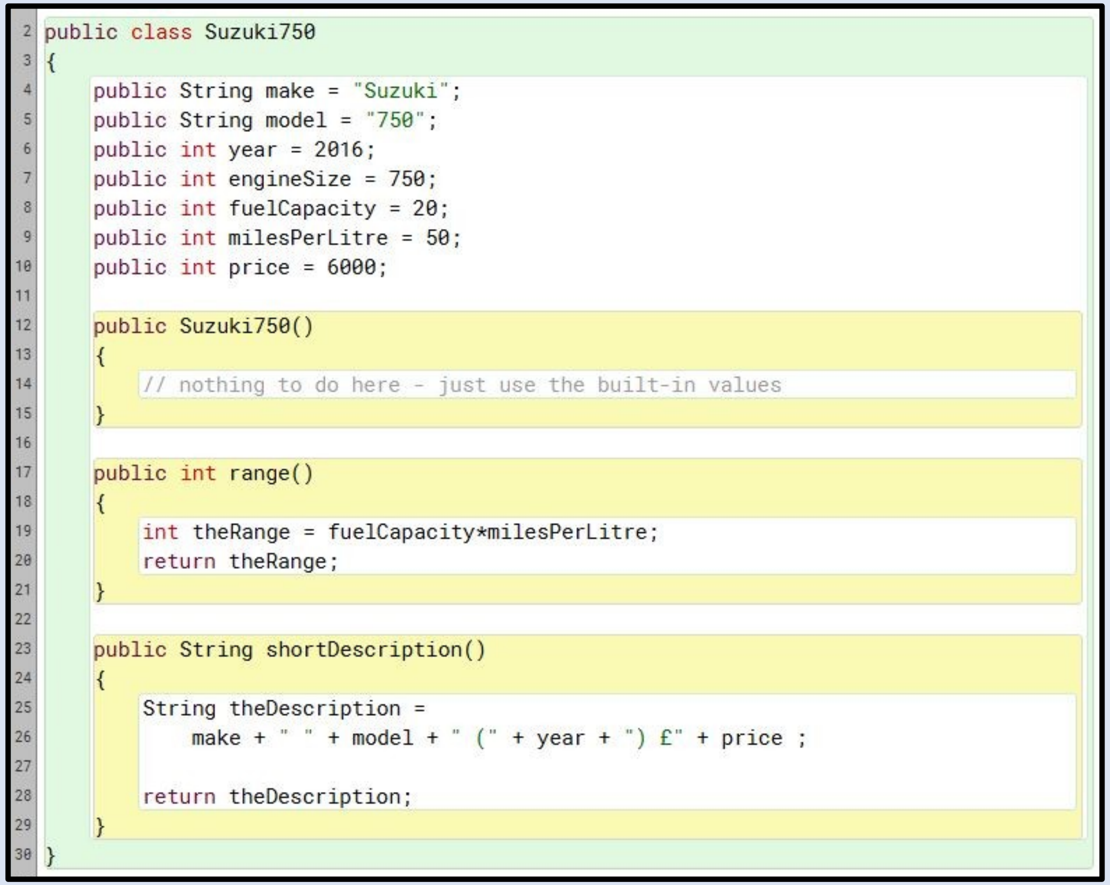


```java
public class Suzuki750
{
    public String make = "Suzuki";
    public String model = "750";
    public int year = 2016;
    public int engineSize = 750;
    public int fuelCapacity = 20;
    public int milesPerLitre = 50;
    public int price = 6000;

    public Suzuki750()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Step 1 – put all the things the two classes share into a new class

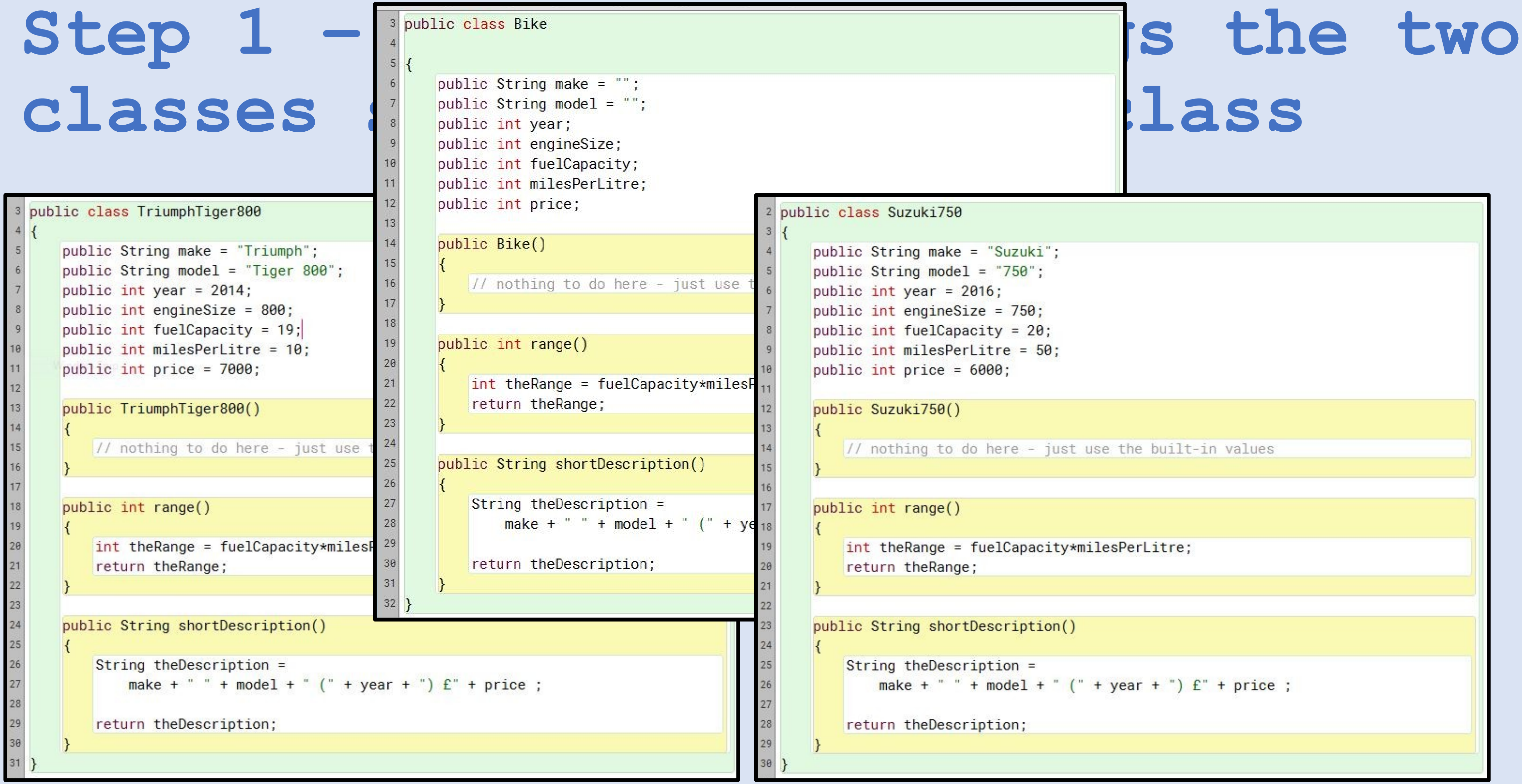


```
public class TriumphTiger800
{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int year = 2014;
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int price = 7000;

    public TriumphTiger800()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```



```
public class Suzuki750
{
    public String make = "Suzuki";
    public String model = "750";
    public int year = 2016;
    public int engineSize = 750;
    public int fuelCapacity = 20;
    public int milesPerLitre = 50;
    public int price = 6000;

    public Suzuki750()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Step 1 – s the two classes lass

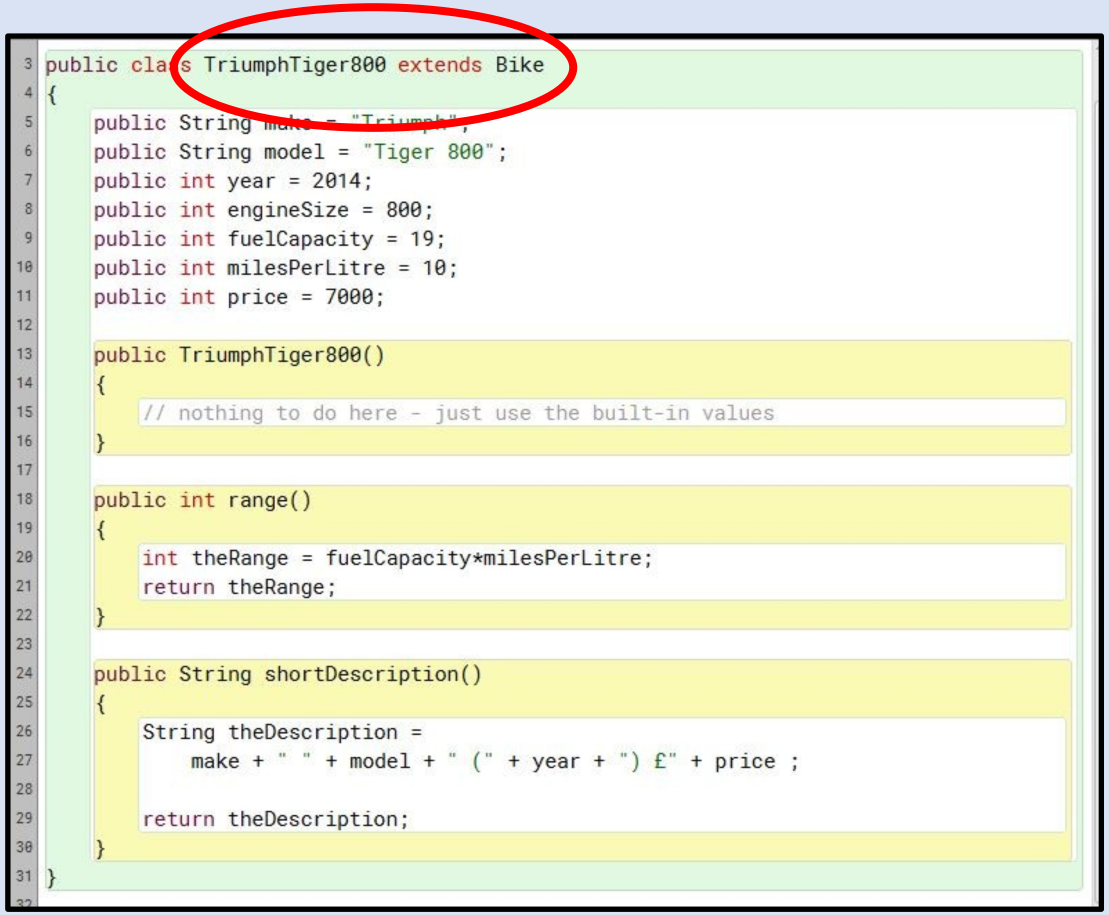


```
public class TriumphTiger800
{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int year = 2014;
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int price = 7000;

    public TriumphTiger800()
    {
        // nothing to do here - just use t
    }

    public int range()
    {
        int theRange = fuelCapacity*milesP
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

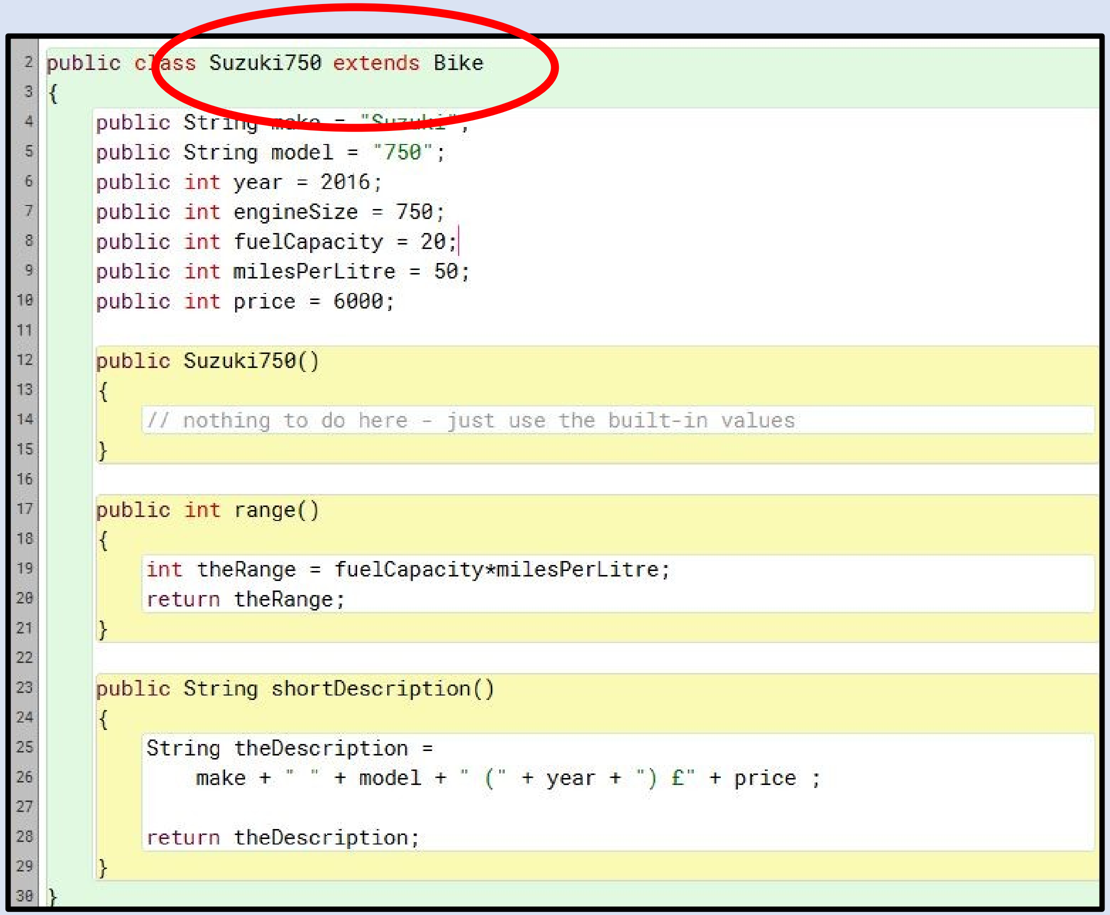
```
public class Bike

{
    public String make = "";
    public String model = "";
    public int year;
    public int engineSize;
    public int fuelCapacity;
    public int milesPerLitre;
    public int price;

    public Bike()
    {
        // nothing to do here - just use t
    }

    public int range()
    {
        int theRange = fuelCapacity*milesP
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + ye

        return theDescription;
    }
}
```

```
public class Suzuki750
{
    public String make = "Suzuki";
    public String model = "750";
    public int year = 2016;
    public int engineSize = 750;
    public int fuelCapacity = 20;
    public int milesPerLitre = 50;
    public int price = 6000;

    public Suzuki750()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Step 2 – Tell Java about the inheritance link

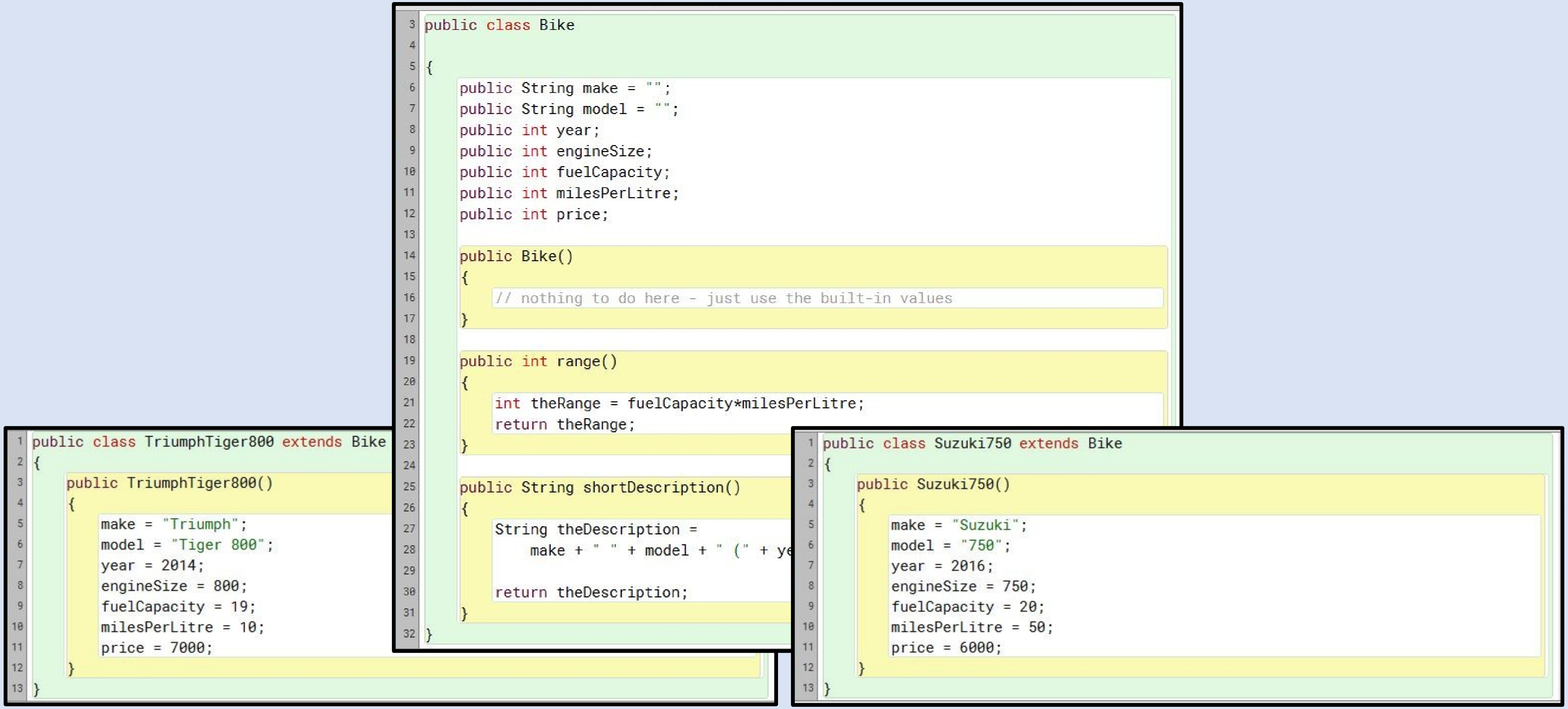


```java
public class TriumphTiger800 extends Bike
{
    public String make = "Triumph";
    public String model = "Tiger 800";
    public int year = 2014;
    public int engineSize = 800;
    public int fuelCapacity = 19;
    public int milesPerLitre = 10;
    public int price = 7000;

    public TriumphTiger800()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```



```java
public class Suzuki750 extends Bike
{
    public String make = "Suzuki";
    public String model = "750";
    public int year = 2016;
    public int engineSize = 750;
    public int fuelCapacity = 20;
    public int milesPerLitre = 50;
    public int price = 6000;

    public Suzuki750()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

Step 3 – remove the shared material from subclasses

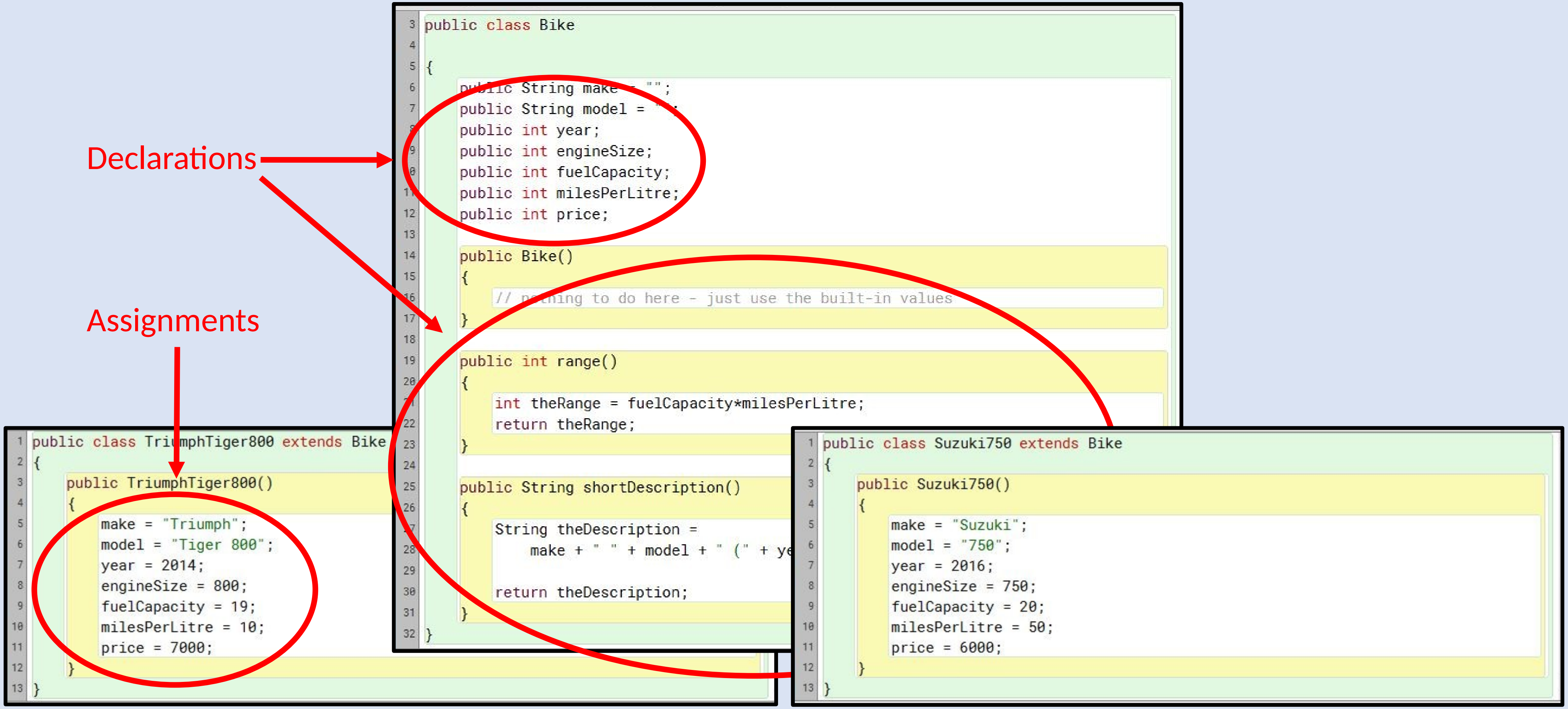


```java
public class Bike

{
    public String make = "";
    public String model = "";
    public int year;
    public int engineSize;
    public int fuelCapacity;
    public int milesPerLitre;
    public int price;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + ye

        return theDescription;
    }
}
```

```java
public class TriumphTiger800 extends Bike
{
    public TriumphTiger800()
    {
        make = "Triumph";
        model = "Tiger 800";
        year = 2014;
        engineSize = 800;
        fuelCapacity = 19;
        milesPerLitre = 10;
        price = 7000;
    }
}
```

```java
public class Suzuki750 extends Bike
{
    public Suzuki750()
    {
        make = "Suzuki";
        model = "750";
        year = 2016;
        engineSize = 750;
        fuelCapacity = 20;
        milesPerLitre = 50;
        price = 6000;
    }
}
```

Sharing variables and methods ('declarations')



Declarations

Assignments

```
public class Bike

{
    public String make = "";
    public String model = "";
    public int year;
    public int engineSize;
    public int fuelCapacity;
    public int milesPerLitre;
    public int price;

    public Bike()
    {
        // nothing to do here - just use the built-in values
    }

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + ye

        return theDescription;
    }
}
```

```
public class TriumphTiger800 extends Bike
{
    public TriumphTiger800()
    {
        make = "Triumph";
        model = "Tiger 800";
        year = 2014;
        engineSize = 800;
        fuelCapacity = 19;
        milesPerLitre = 10;
        price = 7000;
    }
}
```

```
public class Suzuki750 extends Bike
{
    public Suzuki750()
    {
        make = "Suzuki";
        model = "750";
        year = 2016;
        engineSize = 750;
        fuelCapacity = 20;
        milesPerLitre = 50;
        price = 6000;
    }
}
```

Using inherited classes

- The subclasses behave exactly as they did before – it is as if you had written the superclass code in the subclass

- You can access the variables and run the methods from Bike on objects which are created using Suzuki750:

```
Suzuki750 myBike = new Suzuki750();
System.out.println(myBike.range());
int dateOfManufacture = myBike.year;
```

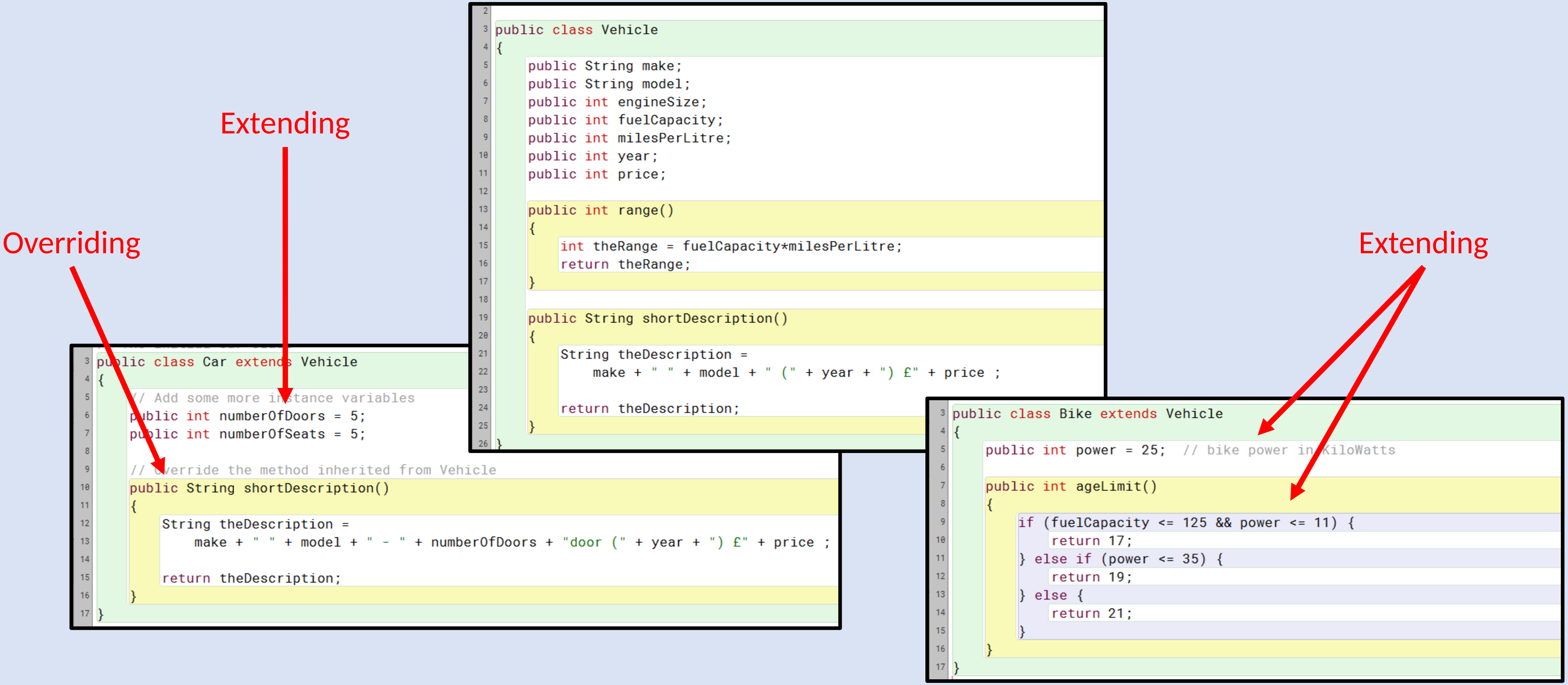
Extension and overriding

Subclasses would not be very much use if they were just exactly the same as their superclass.

A subclass can differ from its superclass in two ways:

- It can extend its superclass by adding new variables and methods of its own.
- It can override methods and variables inherited from its superclass – have different initial values for variables, and define its own versions of methods, instead of using the ones their superclass provides

Extending and overriding



```
public class Vehicle
{
    public String make;
    public String model;
    public int engineSize;
    public int fuelCapacity;
    public int milesPerLitre;
    public int year;
    public int price;

    public int range()
    {
        int theRange = fuelCapacity*milesPerLitre;
        return theRange;
    }

    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " (" + year + ") £" + price ;

        return theDescription;
    }
}
```

```
public class Car extends Vehicle
{
    // Add some more instance variables
    public int numberOfDoors = 5;
    public int numberOfSeats = 5;

    // Override the method inherited from Vehicle
    public String shortDescription()
    {
        String theDescription =
            make + " " + model + " - " + numberOfDoors + "door (" + year + ") £" + price ;

        return theDescription;
    }
}
```

```
public class Bike extends Vehicle
{
    public int power = 25;  // bike power in KiloWatts

    public int ageLimit()
    {
        if (fuelCapacity <= 125 && power <= 11) {
            return 17;
        } else if (power <= 35) {
            return 19;
        } else {
            return 21;
        }
    }
}
```

Two views of inheritance

We can think of inheritance in two ways:

- We have some classes, and we want to create a more general class which defines things they have in common (so we have TriumphTiger800, Suzuik750 and Yamaha500, and we want to define Bike) – we call this abstraction or generalisation
- We have a class, and we want to define some special cases of that class (so we have Bike, and then we add classes for individual bike types, such as Suzuki750 etc) – we call this specialisation

Both views are valid and useful – it’s really a matter of perspective

Why is inheritance a good idea

- Less code – we had two ‘copies’ of all the methods etc. Now we only have one
- Easy to add more classes – just need the subclass files, not all the methods
- Easier to debug and test – we only need to get the code in Bike right once, and we can test it separately from the individual subclasses
- Better structure for your program – easier for other people to understand or modify
- You can use superclass types in your program (eg a method which takes a Bike as argument, but doesn’t care what kind of Bike)

Why is inheritance not always so easy

- You have to choose one structure for your inheritance – this is not always easy, and sometimes you change your mind and have to change your code (this is called refactoring)
- Code gets too interconnected (we call this tightly coupled) – if you change something in a superclass, one of its subclasses might break
- It can be hard to work out where in the code a particular method is implemented – in this class, its superclass, a superclass of that etc?
- It can make programs a bit slower because they also sometimes have to work out where a method is

Introduction to ATM

Starter project – ATM

- The ATM project simulates a cashpoint (except that it doesn’t give you any money 🔍)
- In this week’s lab exercise, you are given a version of ATM that does everything except the actual bank functions (deposit, withdraw etc).
- The lab exercise is to turn it into a ATM (with help from tutors if you need it)
- The solution to this lab is the starting point for your independent project work, if you choose to do ATM.

Demonstrating ATM

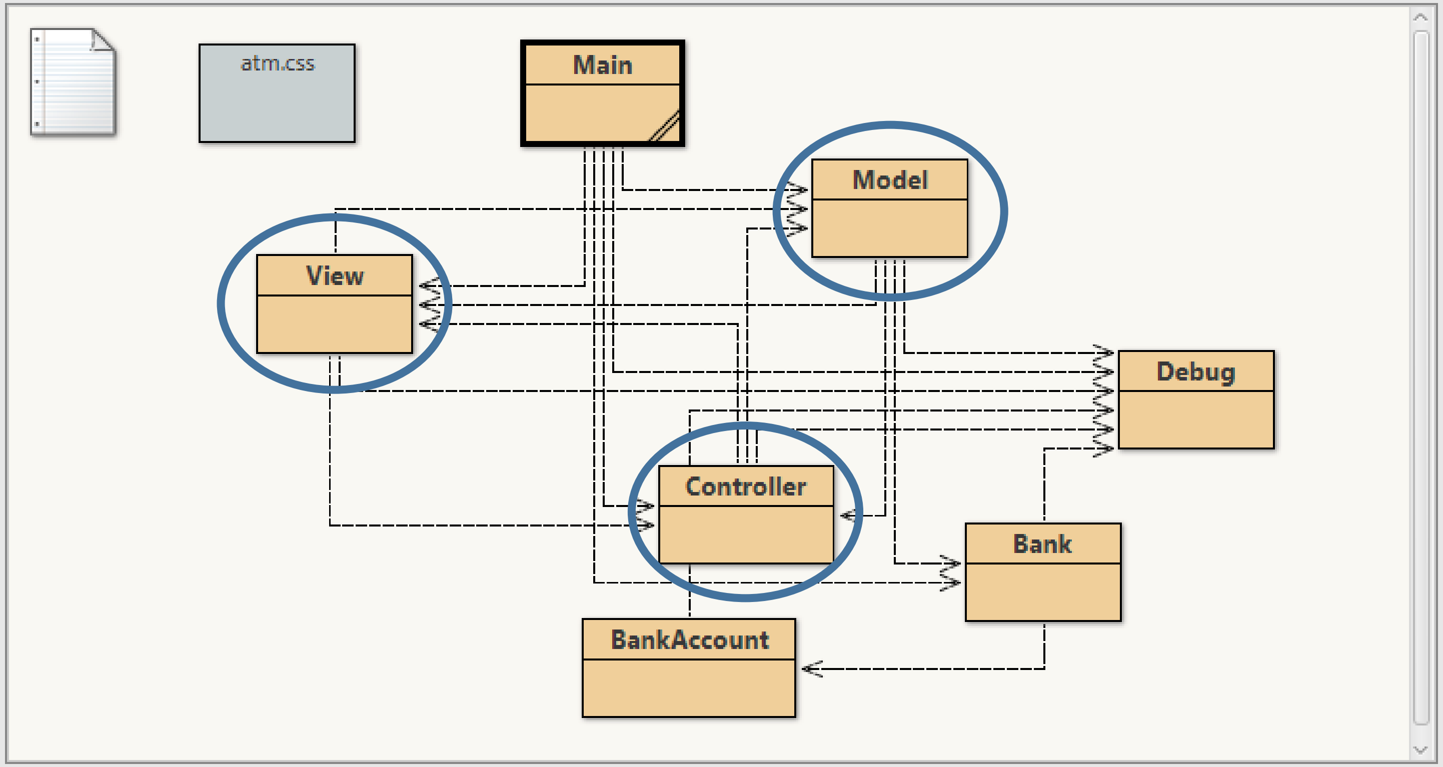
- The solution system – an ATM system which lets you log into your account, check balance and deposit and withdraw money

- The lab exercise system – has no banking functionality! You can still type in your account number and password, and click buttons, but it won’t ever find your account or carry out the banking commands

MVC in the ATM

Here’s the class diagram

- The Model contains the business logic – logging in, communicating with the bank etc.
- The View has all the buttons and layout features we see in the GUI. When the model changes, it updates its display
- The Controller accepts the button presses from the view, and turns them into instructions for the model

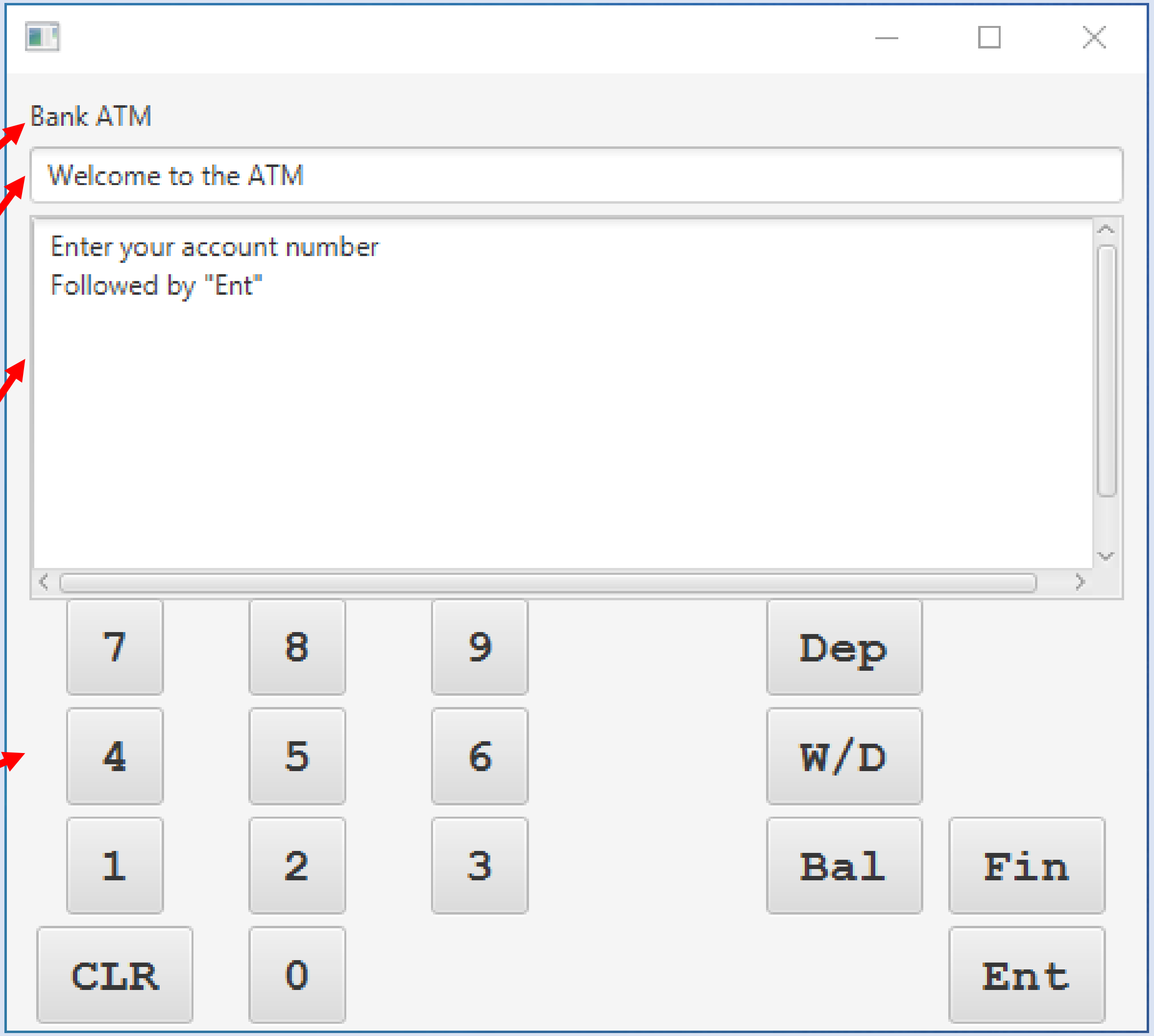


The View

The View object is a fairly straightforward JavaFX UI.

It is a GridPane containing:

- A Label for the title
- A Textbox for messages/typing
- A TextArea for the reply/info (actually wrapped in ScrollPane so it can be bigger than shows on screen)
- A TilePane full of Buttons



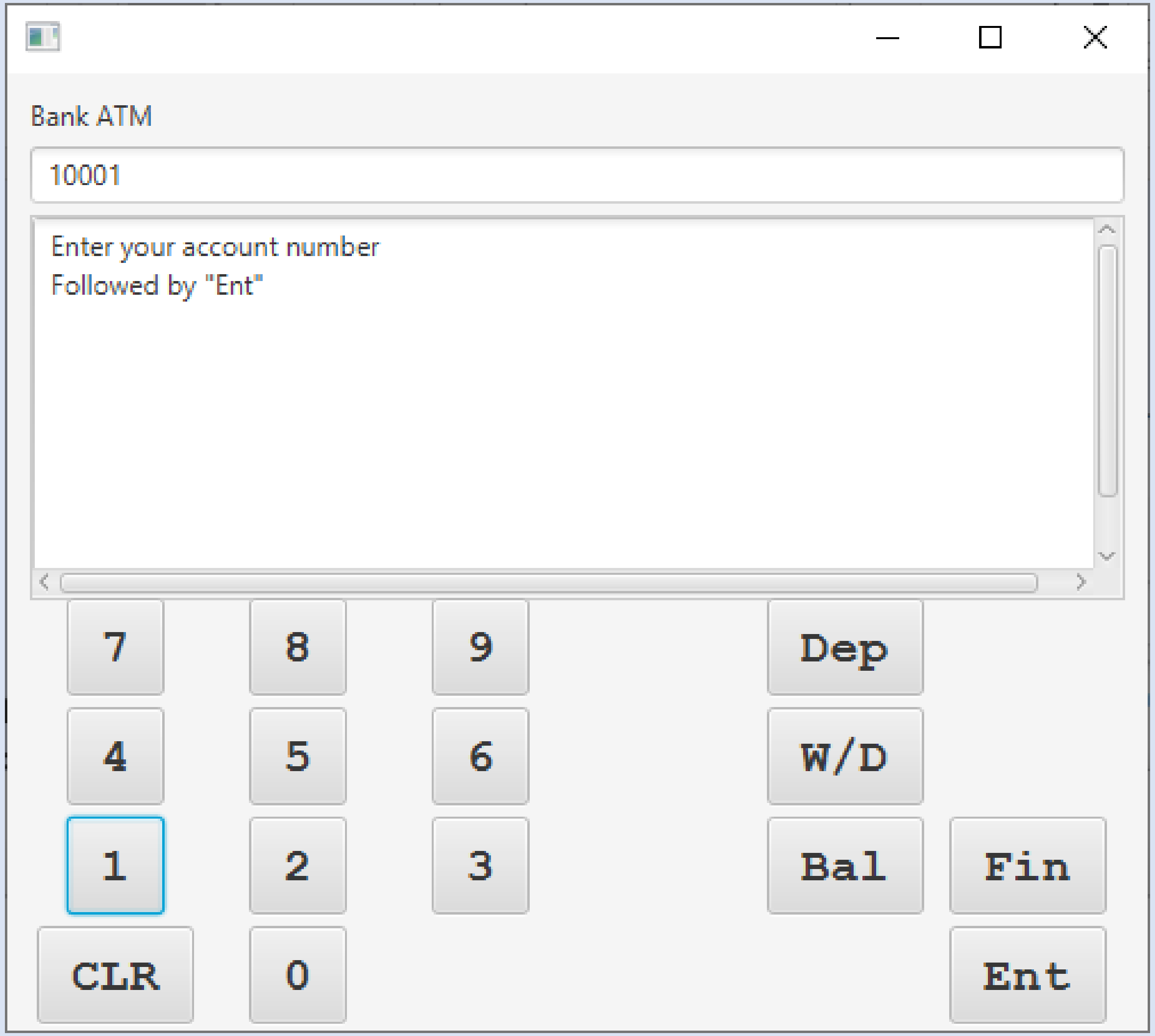
Bank ATM

Welcome to the ATM

Enter your account number
Followed by "Ent"

The View

- When you click on numbers, they appear in the message area
- When you click on Ent (Enter) the reply/info area changes to ask for your password



Bank ATM

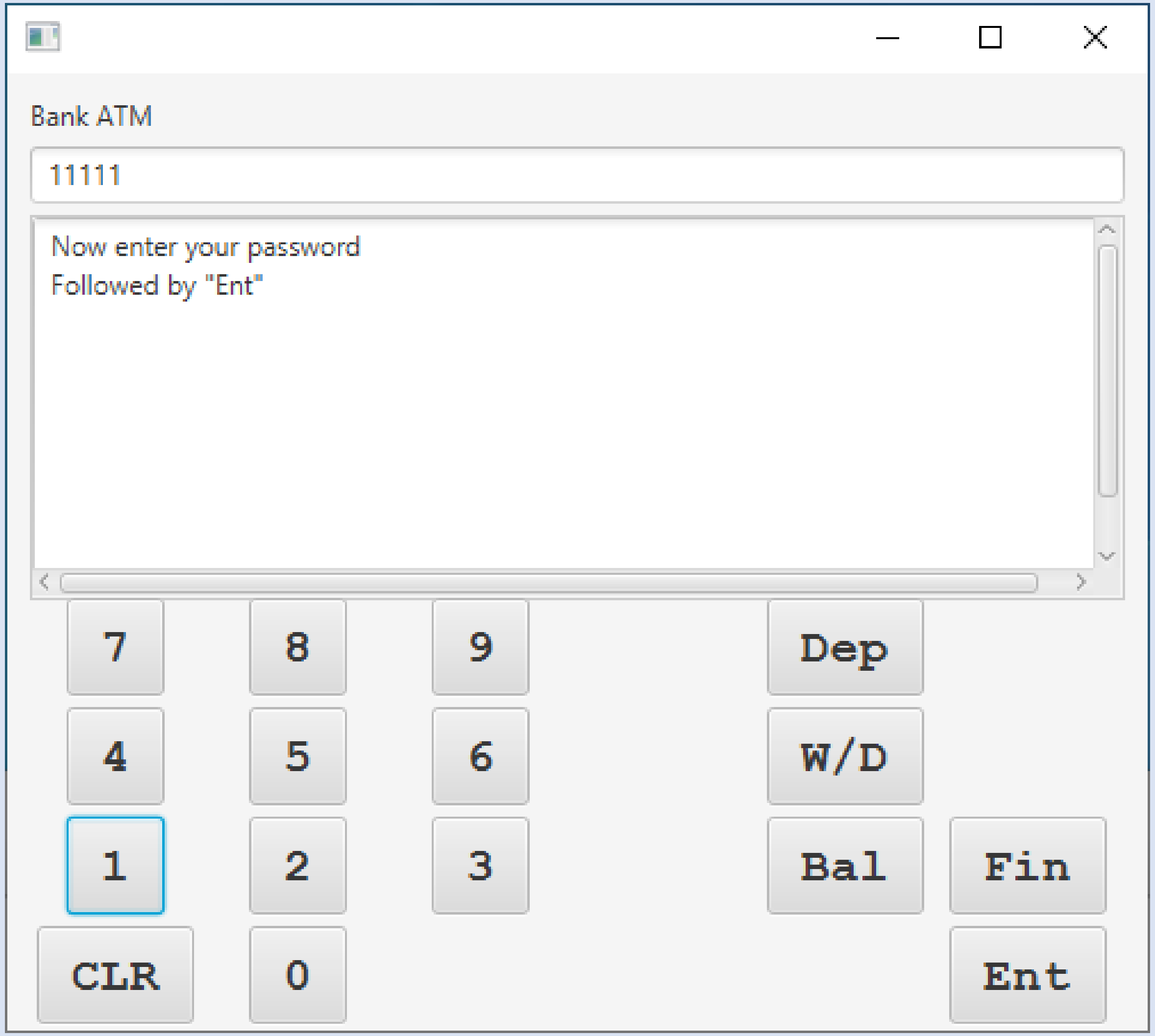
10001

Enter your account number
Followed by "Ent"

The View

- When you click on more numbers, they appear in the message area
- When you click on Ent (Enter) the reply/info area, it tries to log you into your account
- (you will need to add some code to get past this point – see below)

- Each button click generates a message to the Model (via the Controller), which executes the requested action and then tells the View to update the screen



Bank ATM

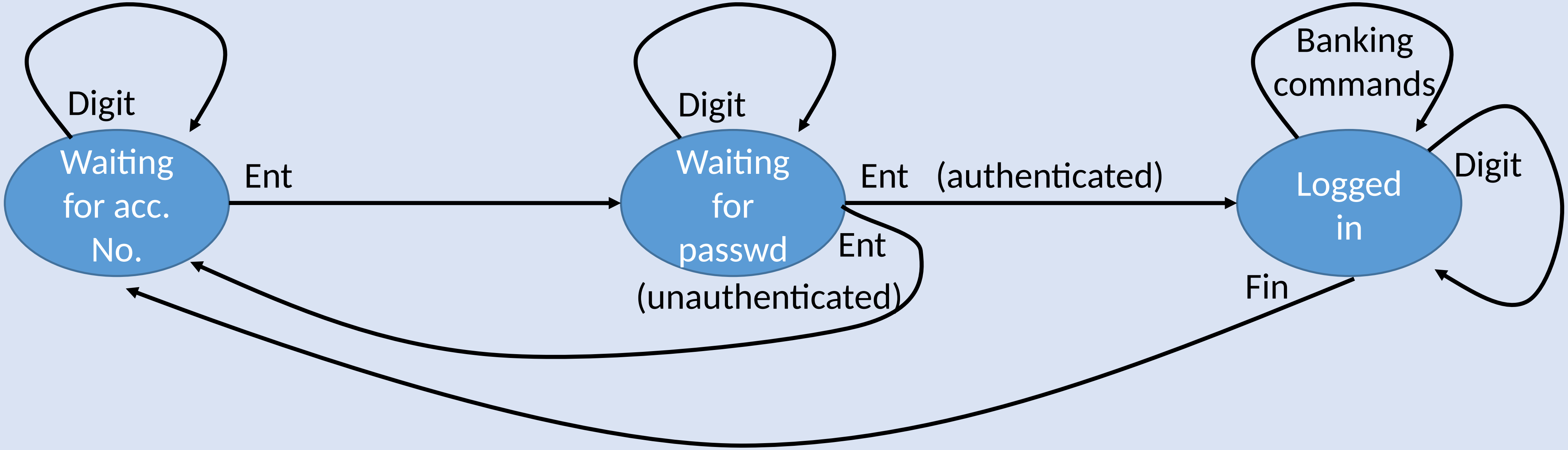
11111

Now enter your password
Followed by "Ent"

The Model class

- The Model:
  - receives messages from the Controller when buttons are clicked on
  - manages user login by always being in one of three states:
    - waiting for account number
    - waiting for password
    - logged in
  - communicates with the Bank object to
    - authenticate account number and password and log in
    - execute banking instructions (when logged in)
  - updates the View with just three Strings:
    - the title message
    - the message area
    - the reply/info area

The Model as a `finite state machine' (FSM)



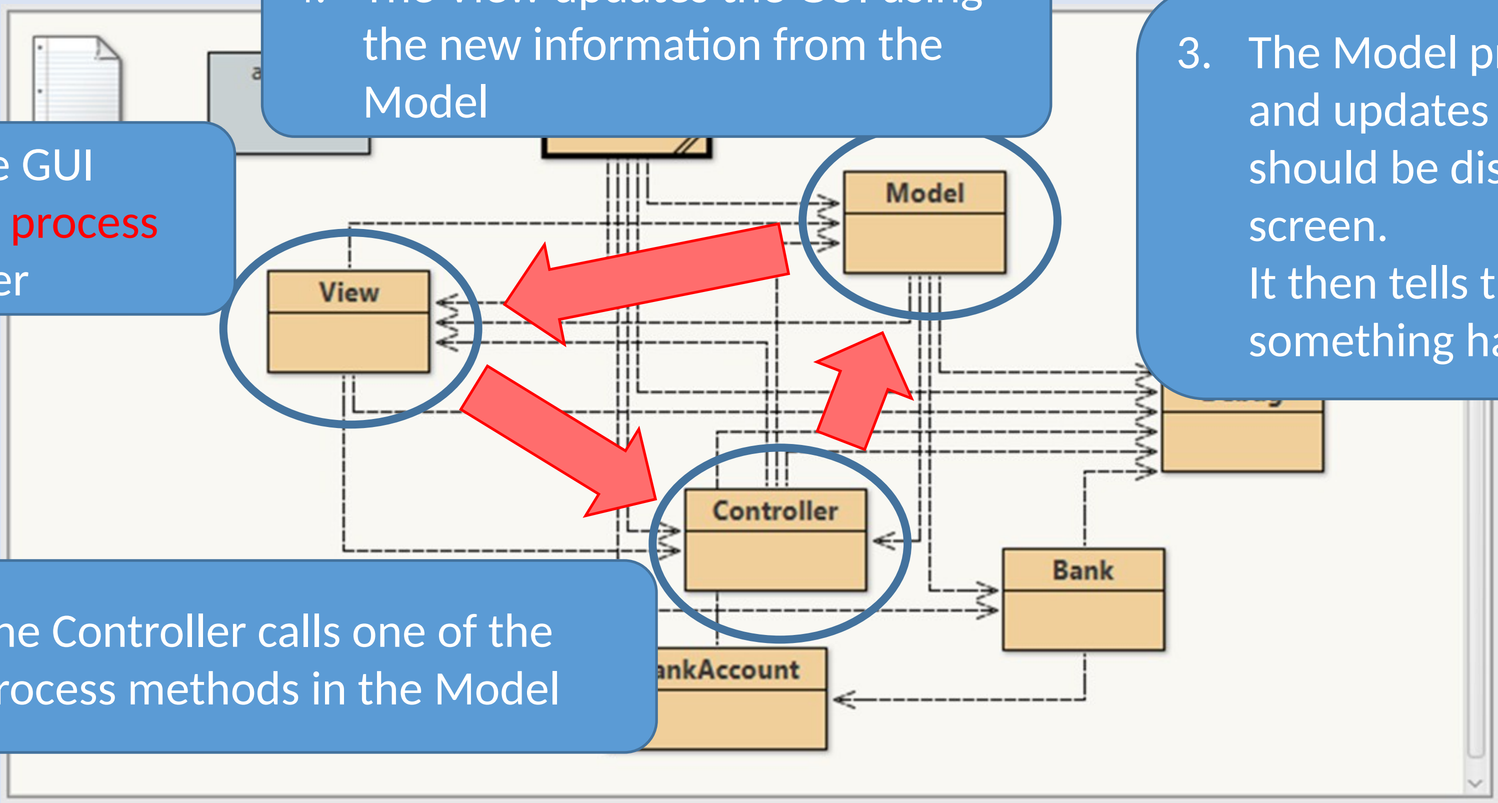
The Controller class

- As with Breakout, the controller is simplest of the three classes
- This provides the event handler function that the View uses
- Whenever the user presses a key, the controller decides what the game should do and tells the Model to do it
- It does this simply on the basis of the labels on each of the buttons. For example
  - When the user clicks a digit, it tells the Model that ‘a digit’ has been clicked (and of course, which one was clicked)
  - Each of the other functions generates a unique message to the Model (in other words, calls a specific method in the Model class)
- So it basically provides a mapping from View buttons to Model functions (Which you can change)

Other classes

- The Main class – starts the program, creates the Model, View, Controller and Bank objects and ‘joins them together’
- The Bank and BankAccount classes – the main data classes representing the banking information that the program manages
- The Debug class – prints out messages about what is going on to help you debug the program

How the components communicate



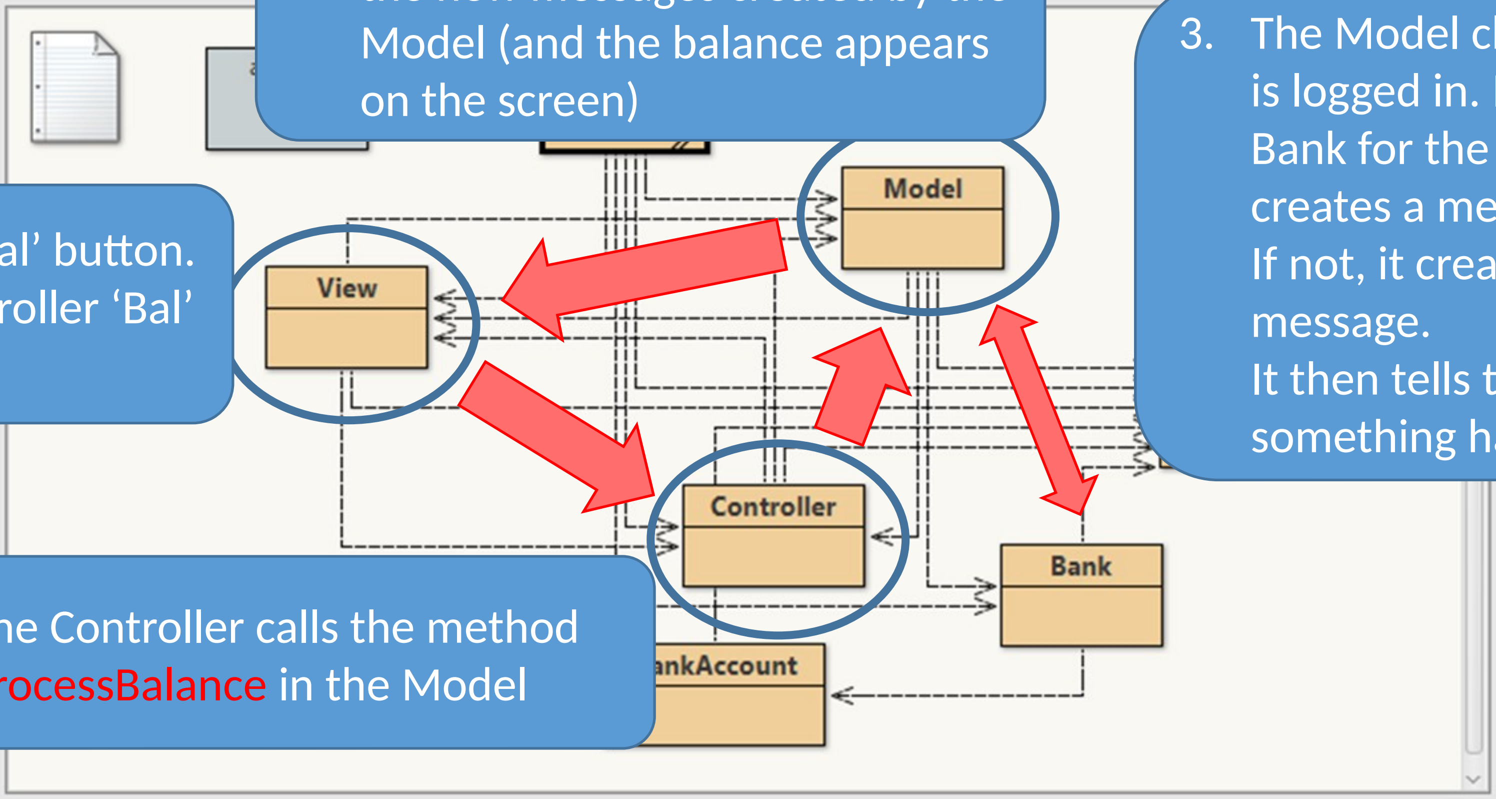
1. Click on a button on the GUI
   The View class calls the process method in the Controller

2. The Controller calls one of the process methods in the Model

3. The Model processes the action and updates what it thinks should be displayed on the screen.
   It then tells the View that something has changed

4. The View updates the GUI using the new information from the Model

ATM example – getting the balance



1. The user presses the ‘Bal’ button. The View tells the Controller ‘Bal’ has been pressed.

2. The Controller calls the method processBalance in the Model

3. The Model checks that the user is logged in. If so it asks the Bank for the balance and creates a message to display it. If not, it creates an error message.
   It then tells the View that something has changed

4. The View updates the GUI with the new messages created by the Model (and the balance appears on the screen)

Lab exercises
Week 2.02

Week 2.02 lab work

- Because the projects are starting up, we are giving you several things to do at the same time
- Don’t worry! – nothing needs to be finished straight away
- This week you will find
  - An inheritance lab, looking at the ideas covered in the lecture
  - An ATM lab, so you can see what the ATM project is like (we will also cover this in the seminars)
- You also have the Breakout lab from last week, and we are providing some additional tips to help you finish it
- You have a couple of weeks more to do the project labs and choose your project

Week 2.02 lab work - ATM

- Run the ATM project and explore the code a little
- You can try and login using account number 10001 and password 11111. These credentials should work, but they don’t because there is code missing in the Bank class (in the login method)
- Once you have fixed that and can log in, you will find that none of the banking functions work. This is because you also need to add code in the BankAccount class (in the withdraw, deposit, and balance methods).
- Remember, we can help you with this lab work. Once the basic ATM is working, you are ready to try things on your own if you want to use the ATM for your project.