CS 162FZ: Introduction to Computer Science II

Lecture 11

Class and Object

Dr. Chun-Yang Zhang

Introduction

- Throughout the course we have seen different inbuilt classes. These classes include the `Scanner class` and the `String class.`
- We have also seen classes of some of the primitive data types, `Integer` and `Double`, eg. `Integer.parseInt()` and `Double.parseDouble().`

Primitive types

- The primitive data types in Java are: `int`, `byte`, `short`, `float`, `double`, `Boolean`, `char` and `long`.
- When we declare a variable of one of these types, a fixed size of memory is allocated to hold the value to be stored.
- The fixed size of memory allocated is different for each type.
- For example if we declare a integer, 32 bytes of space is allocated (discussed in Chapter 4 of Introduction to Computer Science).
- Primitive data types are limited in both their size and their flexibility.

Arrays

- Arrays are a powerful way to extend a primitive data type.
- They allow us to collect variables of the same type under a single variable name.
- We can also make arrays of any size length
- `char word [] = new char[5]; // creates an array of 5 elements`
- `int results[][] = new int[7][6]; // creates an array of 7 rows, 6 columns`
- `int data[][] = new int [4][5]; //this declares a 4x5 matrix of integers.`
- But even arrays are limited in what they can and cannot do for us.

Strings

- A `String` is an example of a reference type. When it is declared no memory is allocated until it is initialized. The `String` type is more powerful than even arrays as it can both hold any sized value, and it provides functionality to interact with that value.
- i.e. `String.length(), String.toUpperCase(), String.toLowerCase() String.charAt(), String.substring(), String.replace() etc.`

- These (`Scanner, String, Integer`) are some classes that are built into java. We can also create java classes to perform a particular user defined task.

User Defined Class

- A **class** can be considered like a template.
- This template can be the blueprint of something more general. Let us consider an example of representing a car in a class.
- All cars have certain features (attributes) in common.
- They all have a motor type of a particular type:
  - Electric,
  - ICE (Internal Combustion Engine)
  - Fuel Cell
- They have a colour
- They have a number of doors
- There are many more attributes that we could list that all cars could share. Can you think of some more?

User Defined Class

In addition to attributes, every class can also perform certain tasks:

They can go forward.

They can go in reverse.

Their doors can be opened and closed.

Their indicators can be switched on and off etc.

User Defined Class

- It would be helpful if we could both **encapsulate** the **description** of the car and encode **functionality** relating to the car.
- The code we could write to capture both attributes and functionality of the car is called a **class.**
- A class is a definition for a category of objects that all have the **same behaviour** and **same type** of **attributes**.
- In Java we can use **data types (like int, double, String etc.) to store attributes** and we can **use methods to encode functionality (behaviour).**
- Note: We can also use **other objects as attributes**!

Components of a Class

- There are a number of components that most classes share in common. These are:

  1. A class name.

- This name should be meaningful and represent the function/purpose of the class. For example – if you are creating a class to represent a car then the name of the class should be `Car.` The normal naming convention is that class names should start with an **uppercase letter and be a noun.**

  2. Attributes, also known as **instance variables.**

- These are variables that will be used inside the class to hold values such as the make/model of the car, how many doors it has, etc.

Components of a Class

3. Constructor(s).

- There should always be a default constructor. This type of constructor has zero input parameters. If there is none provided the JVM will automatically provide one.
- There will usually be another constructor with 1 or more with parameters.

4. Getters methods (also known as Accessors).

- These are methods defined by the user to return the values of the attributes in a class.

5. Setters (also known as Mutators).

- These are methods which change the value of attributes in a class

6. Functionality in the form of methods.

Maynooth University
National University of Ireland Maynooth

Objects

- An object is an **instance** of a class. It can be thought of being like any variable. When using the Scanner class you created an object of that class by writing

```
Scanner sc = new Scanner(System.in)
```

- This is creating an object of the Scanner class. If we create another object of the Scanner class as follows

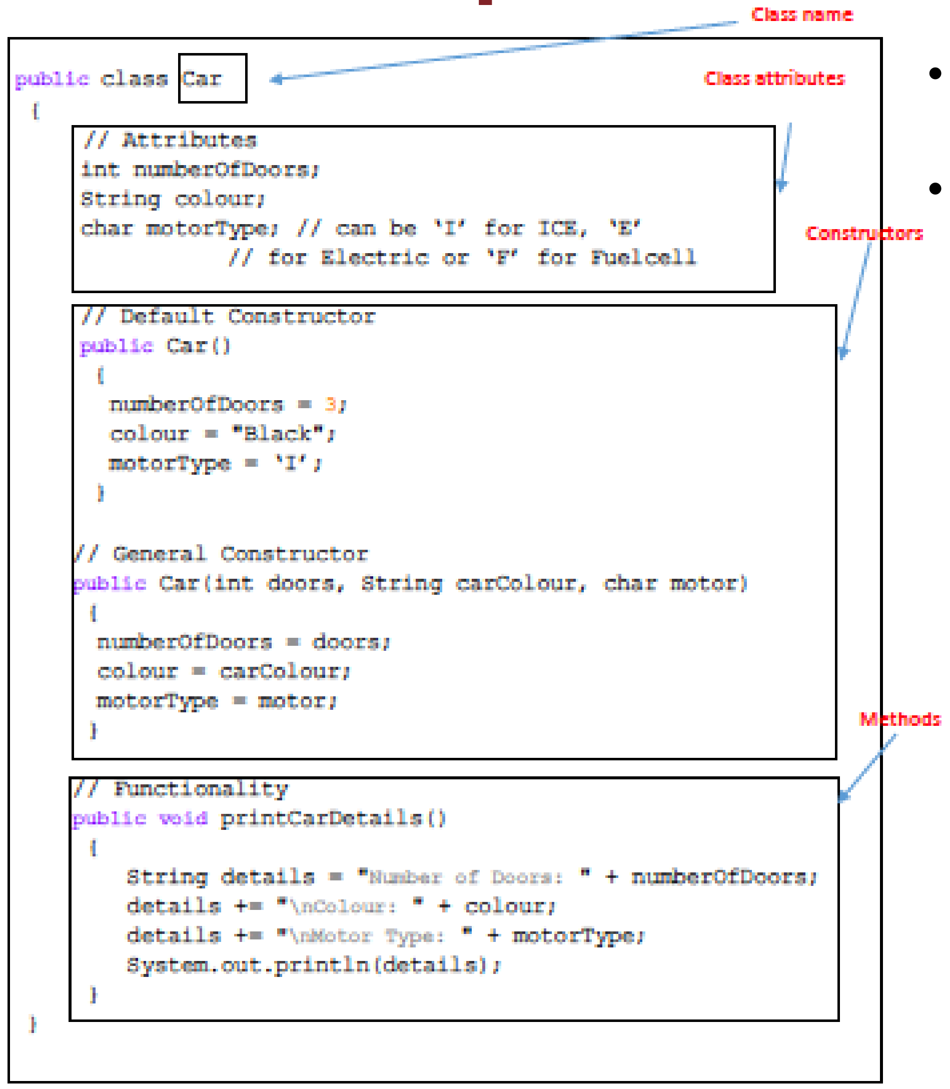
```
Scanner sc1 = new Scanner(System.in)
```

- We now have two instances of the class in our program.

Class Example

- Let us start looking at an example of a java class to represent a `Car`.
- Our class, called `Car` will have the following attributes: `numberOfDoors`, `motorType` and `colour`.
- We might want to have a method called `printCarDetails()` that prints out these attributes.

Class Example

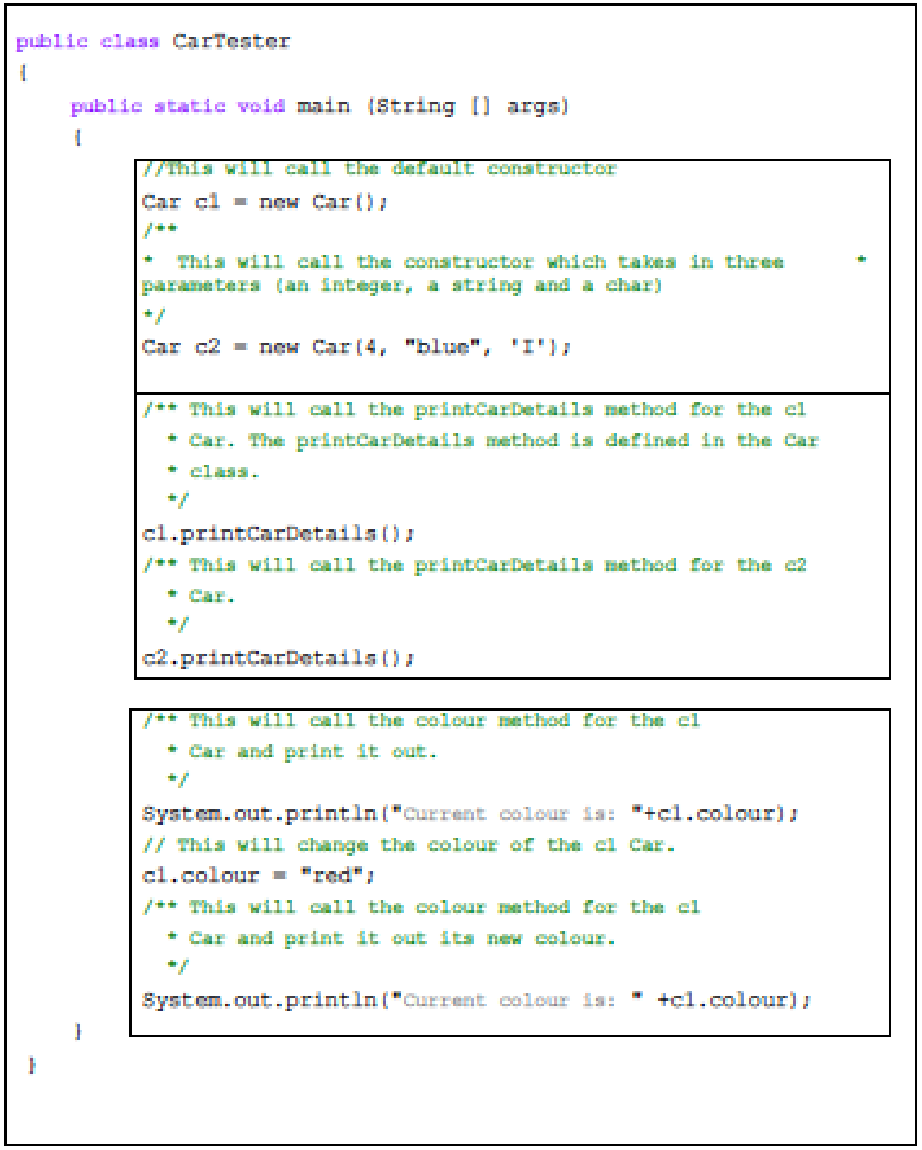


```
public class Car
{
    // Attributes
    int numberOfDoors;
    String colour;
    char motorType; // can be 'I' for ICE, 'E'
                    // for Electric or 'F' for Fuelcell

    // Default Constructor
    public Car()
    {
        numberOfDoors = 3;
        colour = "Black";
        motorType = 'I';
    }

    // General Constructor
    public Car(int doors, String carColour, char motor)
    {
        numberOfDoors = doors;
        colour = carColour;
        motorType = motor;
    }

    // Functionality
    public void printCarDetails()
    {
        String details = "Number of Doors: " + numberOfDoors;
        details += "\nColour: " + colour;
        details += "\nMotor Type: " + motorType;
        System.out.println(details);
    }
}
```

- It is important to note we have no main() method in our class.
- In our main() method (contained in another file), we can create objects of the Car class in a similar way we create objects of the Scanner Class.

Class Example

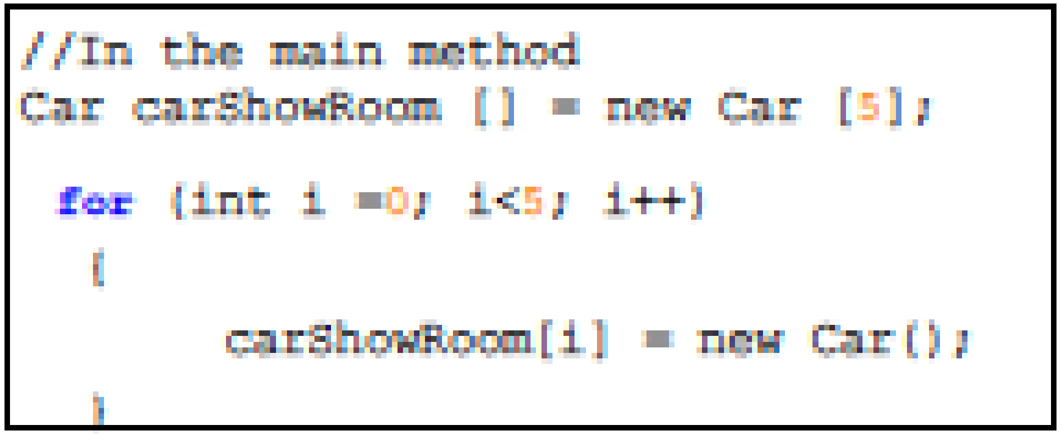
- In our main() method (contained in another file), we can create objects of the `Car class` in a similar way we create objects of the `Scanner Class.`
- For example:



```
public class CarTester
{
    public static void main (String [] args)
    {
        //This will call the default constructor
        Car c1 = new Car();
        /**
        *  This will call the constructor which takes in three      *
        parameters (an integer, a string and a char)
        */
        Car c2 = new Car(4, "blue", 'I');

        /** This will call the printCarDetails method for the c1
          * Car. The printCarDetails method is defined in the Car
          * class.
          */
        c1.printCarDetails();
        /** This will call the printCarDetails method for the c2
          * Car.
          */
        c2.printCarDetails();

        /** This will call the colour method for the c1
          * Car and print it out.
          */
        System.out.println("Current colour is: "+c1.colour);
        // This will change the colour of the c1 Car.
        c1.colour = "red";
        /** This will call the colour method for the c1
          * Car and print it out its new colour.
          */
        System.out.println("Current colour is: " +c1.colour);
    }
}
```

Arrays of Objects

- We may wish to create many cars in our program. In the previous example we created two objects of Cars called c1 and c2. However, if we wanted to create more it can get very messy. Using an array to store objects of the same type can help make it easier. As an example, let us create five Cars.



```
//In the main method
Car carShowRoom [] = new Car [5];

  for (int i =0; i<5; i++)
   {
          carShowRoom[i] = new Car();
   }
```

Naming Conventions for Classes and Objects

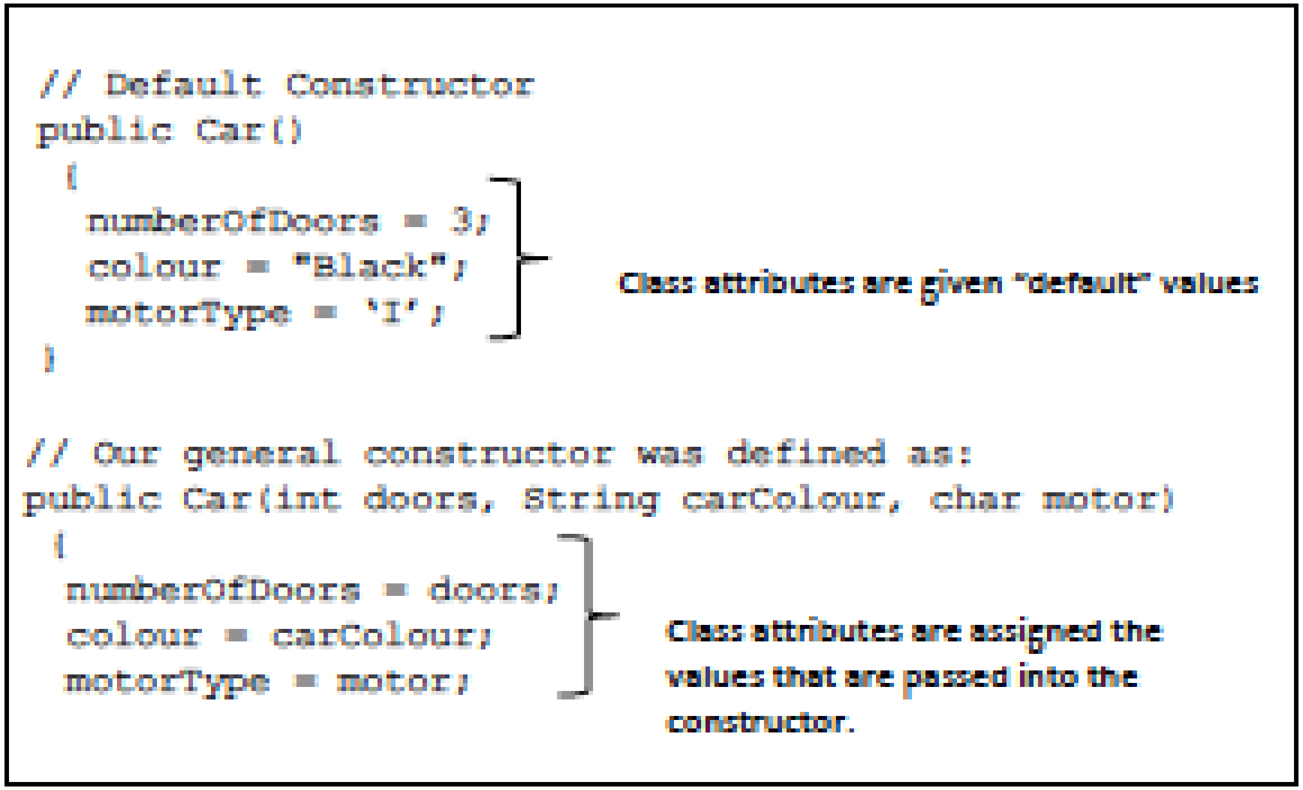
- When creating a class, the class name should always start with an upper case letter
- e.g. `public class Car`
- When creating an object of a class, the object name should always start with a lower case letter with any subsequent word starting with an uppercase letter. For example:
- `Car myCar = new Car();`
- `Table woodenTable = new Table();`
- `Chair threeLeggedStool = new Chair();`

Constructors

- A constructor is a **special type** of method that is invoked when the `new` keyword is used with the object.
- They have **no return types** (**not even void**) and must have the **same name as the class**.
- We usually have **a default constructor** (i.e. one with no parameters)
- and **at least one other constructor** with 1 or more parameters.

Constructors

In the Car class example, the default constructor is:



```
// Default Constructor
public Car()
 {
  numberOfDoors = 3;
  colour = "Black";
  motorType = 'I';
}
```

Class attributes are given "default" values

```
// Our general constructor was defined as:
public Car(int doors, String carColour, char motor)
 {
  numberOfDoors = doors;
  colour = carColour;
  motorType = motor;
```

Class attributes are assigned the values that are passed into the constructor.

Accessor and Mutator Methods

- **Accessor** methods are used to **return (get)** an attributes value.
- They are also referred to as **getter methods**.
- **Mutator** methods are used to **change (set)** an attributes value.
- They are also referred to as **setter methods**.
- Accessor and mutator methods define a precise way for the user to interact with these class variables.

Accessor and Mutator Methods

- Let us look at an example of an accessor method to get the colour of a car.

```
 public String getColour()
 {
 return colour;
 }
```

- It should be noted that an accessor method takes zero input parameters and has a return type (i.e. not void).

- The example method above gets the value in the variable colour and returns it to the calling method.

- As the method is public, any class will be able to call it.

Accessor and Mutator Methods

- Let us look at an example of a mutator method to set the colour of a car.

```
 public void setColour(String
 carColour)
 {
 colour = carColour;
 }
```

- It should be noted that a mutator method takes one or more parameters and always has a void return type.

- This example method sets the value of the variable `colour` to the value passed in to the method in the variable `carColour`.

Naming Conventions for Accessor and Mutator methods

- When naming accessor and mutator methods, the following accepted naming convention should be used.
- The accepted convention is that the first letter of the first word of the method name is lower case (and will be either `get or set`) and all others are upper case. For example:

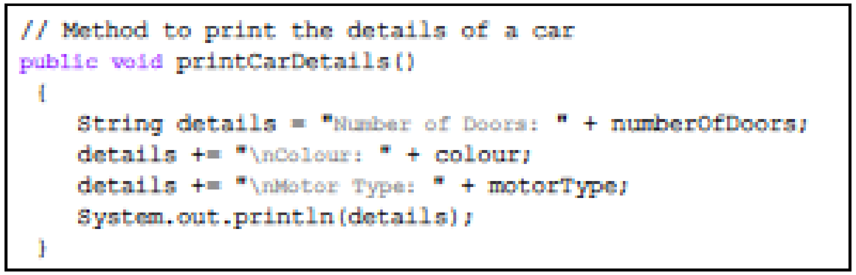
- Accessor: `get<AttributeName>()` e.g. for the attribute `carReg` we use `getCarReg()`

- Mutator: `set<AttributeName>()` e.g for the attribute `motorType` we use `setMotorType()`

Maynooth University
National University of Ireland Maynooth

Static Keyword with Classes

- Notice that when we created `printCarDetails()` we did not use the keyword `static`.
- If we use `static` with a method (or a variable) then that method belongs exclusively to the class definition and will never be copied to an instance of the class (i.e. an object).
- If we leave the keyword `static` out, then every instance of the class will get its own copy of the method (or variable).



```
// Method to print the details of a car
public void printCarDetails()
 {
    String details = "Number of Doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    System.out.println(details);
 }
```

Another user defined class example

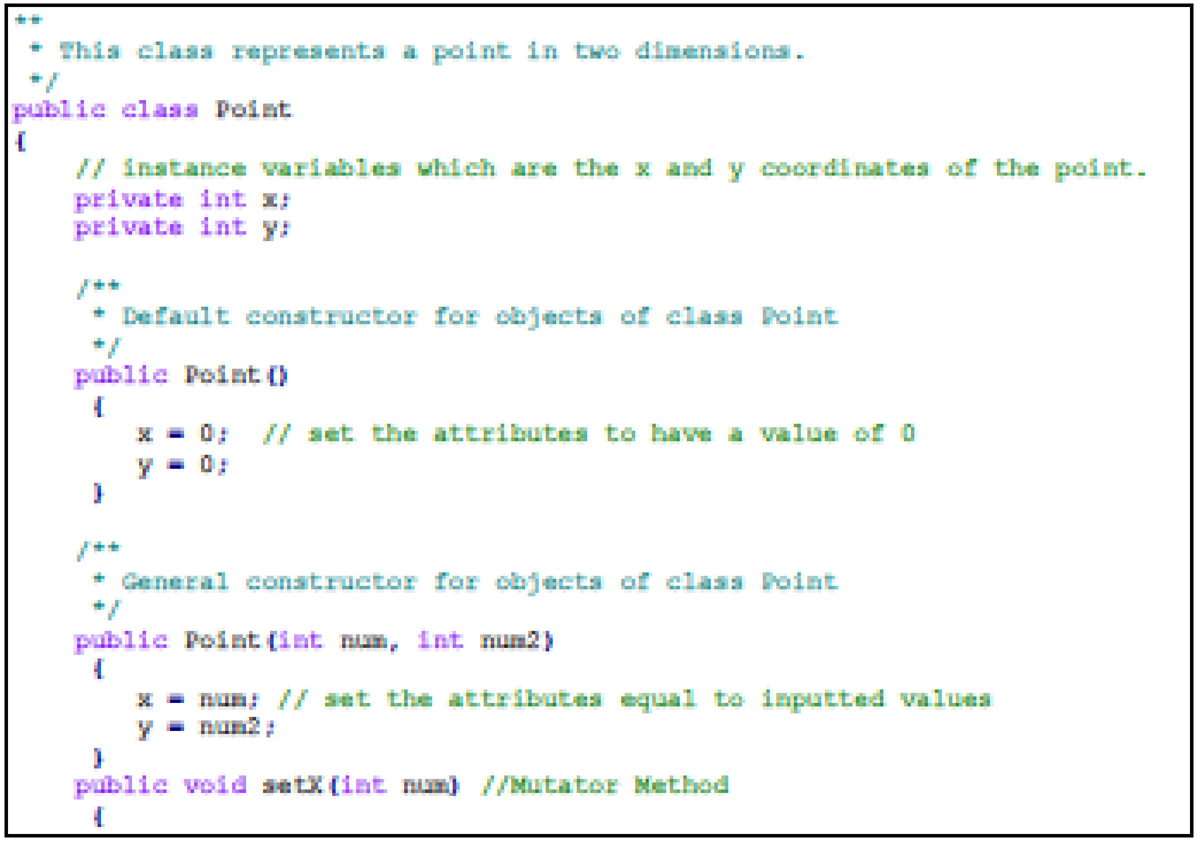
- The following class describes a Point in 2D space.
- A Point has attributes storing an X position and a Y position
- It describes a `printDetails()` method that prints both of the attribute values to the screen.

- The class should have:
  a) a constructor that takes no parameters and sets both attribute values to 0
  b) a constructor that takes values for each attribute and sets the attribute to the values passed in

Maynooth University
National University of Ireland Maynooth

Another user defined class example

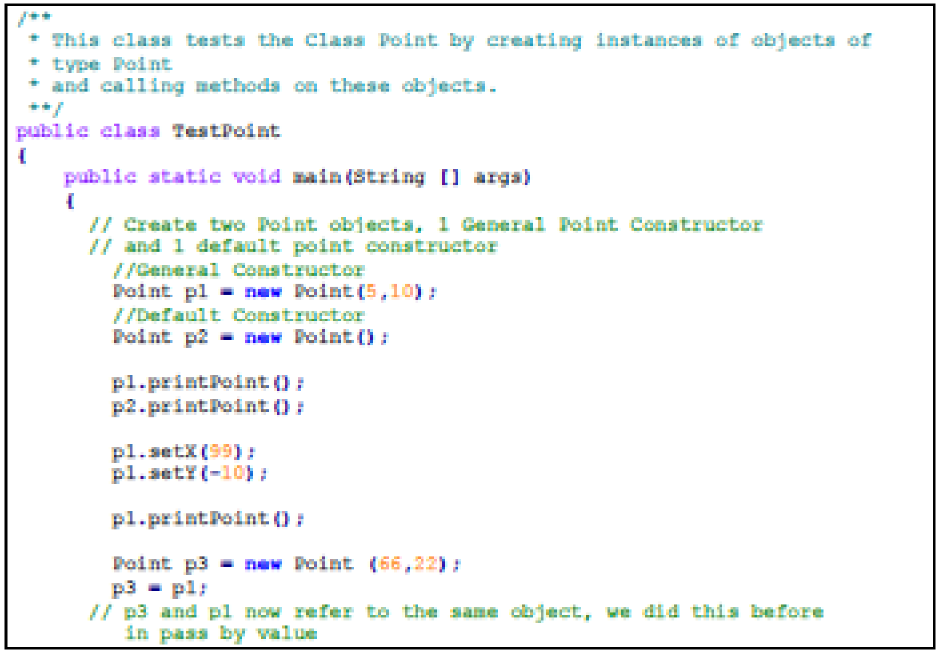
An example of the class to represent this scenario is as foll



```
**
 * This class represents a point in two dimensions.
 */
public class Point
{
    // instance variables which are the x and y coordinates of the point.
    private int x;
    private int y;

    /**
     * Default constructor for objects of class Point
     */
    public Point()
     {
         x = 0;  // set the attributes to have a value of 0
         y = 0;
     }

    /**
     * General constructor for objects of class Point
     */
    public Point(int num, int num2)
     {
         x = num; // set the attributes equal to inputted values
         y = num2;
     }
    public void setX(int num) //Mutator Method
     {
        x = num ;
       // set the class instance variable to the value in the formal
       // parameter num.
     }
                                                                     PTO.
```

```
    Public int getX()   //Mutator Method
     {
        return x; // return the value in the class instance variable x
     }

    public void setY(int num)
     {
        y = num ;
       // set the class instance variable to the value in the formal
       // parameter num.
     }

    public int getY()
     {
        return y; // return the value in the class instance variable y
     }
    public void printPoint()
     {
        System.out.println("Point  coordinates are (" + getX() + "," +
        getY()+")");
     }
}
```

- The previous example code represents a Point in 2D space.
- To complete the class implementation, we need to create a test class that will create new instances of the class and call the associated methods.



```
/**
 * This class tests the Class Point by creating instances of objects of
 * type Point
 * and calling methods on these objects.
 **/
public class TestPoint
{
    public static void main(String [] args)
    {
      // Create two Point objects, 1 General Point Constructor
      // and 1 default point constructor
        //General Constructor
        Point p1 = new Point(5,10);
        //Default Constructor
        Point p2 = new Point();

        p1.printPoint();
        p2.printPoint();

        p1.setX(99);
        p1.setY(-10);

        p1.printPoint();

        Point p3 = new Point (66,22);
        p3 = p1;
      // p3 and p1 now refer to the same object, we did this before
         in pass by value
```

Another user defined class example

```java
/**
 * This class tests the Class Point by creating instances of objects of
 * type Point
 * and calling methods on these objects.
 **/
public class TestPoint
{
    public static void main(String [] args)
    {
      // Create two Point objects, 1 General Point Constructor
      // and 1 default point constructor
        //General Constructor
        Point p1 = new Point(5,10);
        //Default Constructor
        Point p2 = new Point();

        p1.printPoint();
        p2.printPoint();

        p1.setX(99);
        p1.setY(-10);

        p1.printPoint();

        Point p3 = new Point (66,22);
        p3 = p1;
      // p3 and p1 now refer to the same object, we did this before
         in pass by value
```

```java
        p3.printPoint();
        p1.printPoint();

    }
}
```

Maynooth University
National University of Ireland Maynooth

Modifiers in Java

- Modifiers change the meaning of the variables, methods and classes they are placed in front of.

- There are two types of modifiers:

1. Access Modifiers – sets levels of access to a variable, method, class or package

2. Non-Access Modifiers – alters the variable, method, class or package in some other (non-access related) way

- The keyword `static` is an example of a modifier.

Access Modifiers

There are **four types** of access modifiers:

1. `Public` makes the variable, method, class or package accessible to the world
2. `private` makes the variable or method accessible only within the class in which it is defined
3. `Protected` makes the variable, method or class accessible only within the package or derived subclasses
4. `<none>` - default makes the variable, method or class accessible only within the package (but not accessible to subclasses)

Maynooth University
National University of Ireland Maynooth

Access Modifiers

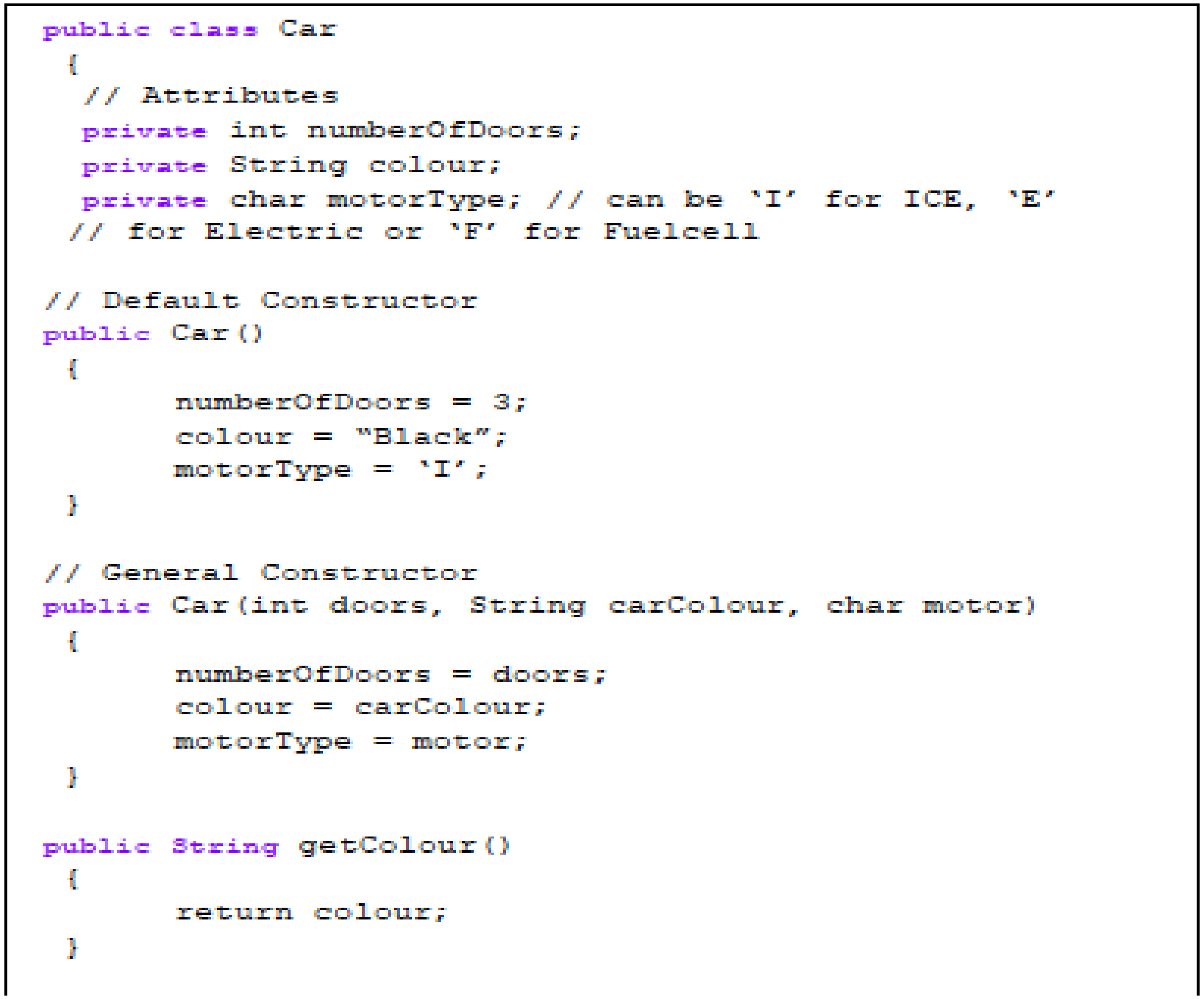
- Knowing when to use these modifiers will depend on the requirements of the class, but as a rule of thumb following the following:
  a) *Most classes are public.*
  b) *Most attributes are private.*
  c) *Methods are generally public, sometimes protected and occasionally private.*

Making Attributes Private

- In our `Car` class we should alter our attributes as follows (but only if we add in accessor and mutator methods that will allow us to access the attribute values):
  - `private int numberOfDoors;`
  - `private String colour;`
  - `private char motorType;`
- We do this as we generally don't want users to directly interact with our class attributes.
- We will have to give them a different way to access them (if at all).

Making Attributes Private

Our new class will therefore be:

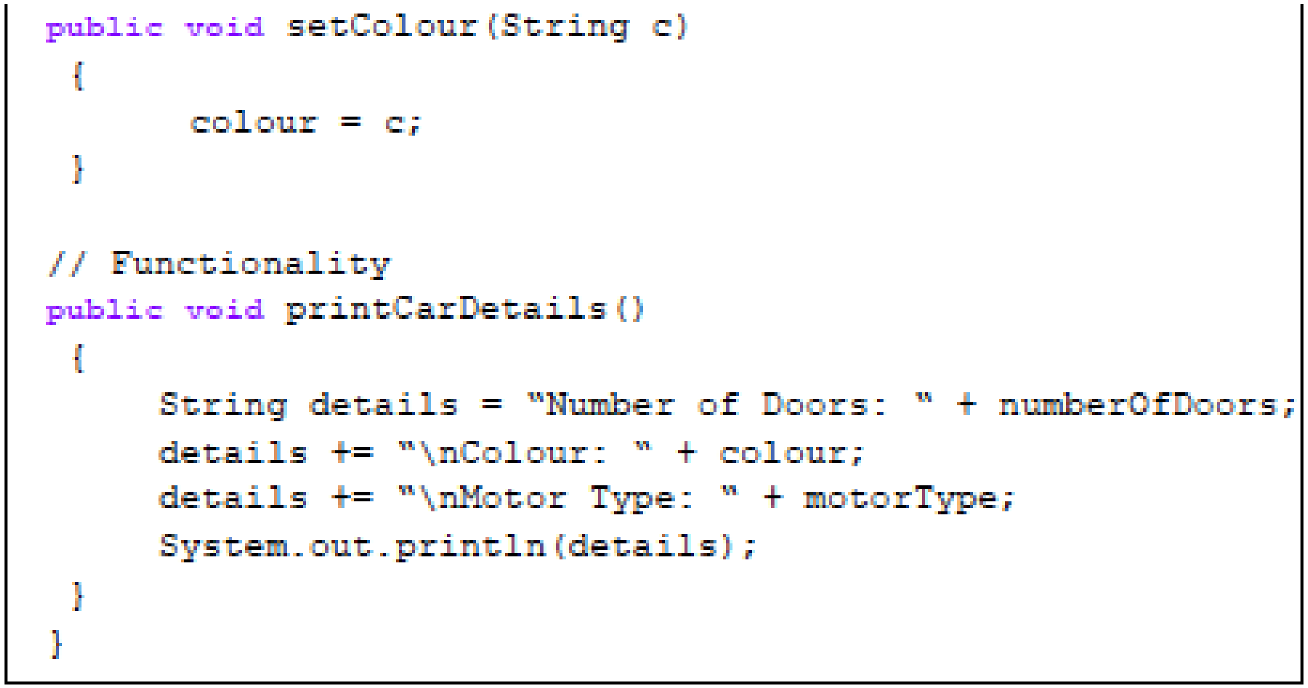


```
public class Car
 {
  // Attributes
  private int numberOfDoors;
  private String colour;
  private char motorType; // can be 'I' for ICE, 'E'
 // for Electric or 'F' for Fuelcell

// Default Constructor
public Car()
 {
        numberOfDoors = 3;
        colour = "Black";
        motorType = 'I';
 }

// General Constructor
public Car(int doors, String carColour, char motor)
 {
        numberOfDoors = doors;
        colour = carColour;
        motorType = motor;
 }

public String getColour()
 {
        return colour;
 }
```

Making Attributes Private

Our new class

will therefore be:



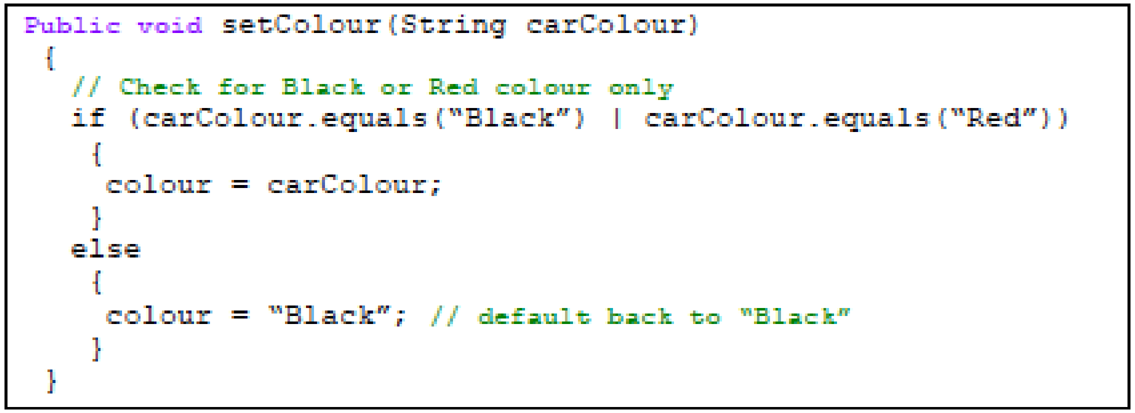
```
public void setColour(String c)
 {
        colour = c;
 }

// Functionality
public void printCarDetails()
 {
      String details = "Number of Doors: " + numberOfDoors;
      details += "\nColour: " + colour;
      details += "\nMotor Type: " + motorType;
      System.out.println(details);
 }
}
```

Controlling Access

- Quite often we want to precisely control how users access our class attributes.
- For example, there are many times when making a variable read-only makes sense.
- Maybe we don’t want to allow a user to change the car registration number of a particular car.
- We could, therefore write only an accessor method and not add any mutator method for `carReg`.
- Another example of controlling access is deciding how to allow a user to set an attribute.
- For example, maybe we want to limit the colours a user can use for a `Car` object.

Further Restrictions

- Let us restrict the colour a car can be to either black or red.
- The following code ensures that the user can’t just store any value in the attribute (e.g. setting the String variable `colour` to “raincoat” or “Foo Fighters”).



```
Public void setColour(String carColour)
 {
   // Check for Black or Red colour only
   if (carColour.equals("Black") | carColour.equals("Red"))
    {
     colour = carColour;
    }
   else
    {
     colour = "Black"; // default back to "Black"
    }
 }
```

Non-Access Modifiers

Examples of Non-Access Modifiers:

- `static` - makes the variable or method a member of the class and not a member of an instance of the class (i.e. a class variable, not an object variable).
- `final` makes the variable constant and **cannot be changed.**
- Others include `abstract` and `synchronized`.

Using Non-Access Modifiers

- Knowing when to use these modifiers will again depend on the requirements of the class.
- We use `final` when making constant variables that are used again and again in our code.
- We use `static` when we are sure we only want one copy of a variable or method to exist, or when we want the class to control that variable or method (rather than an object controlling it)

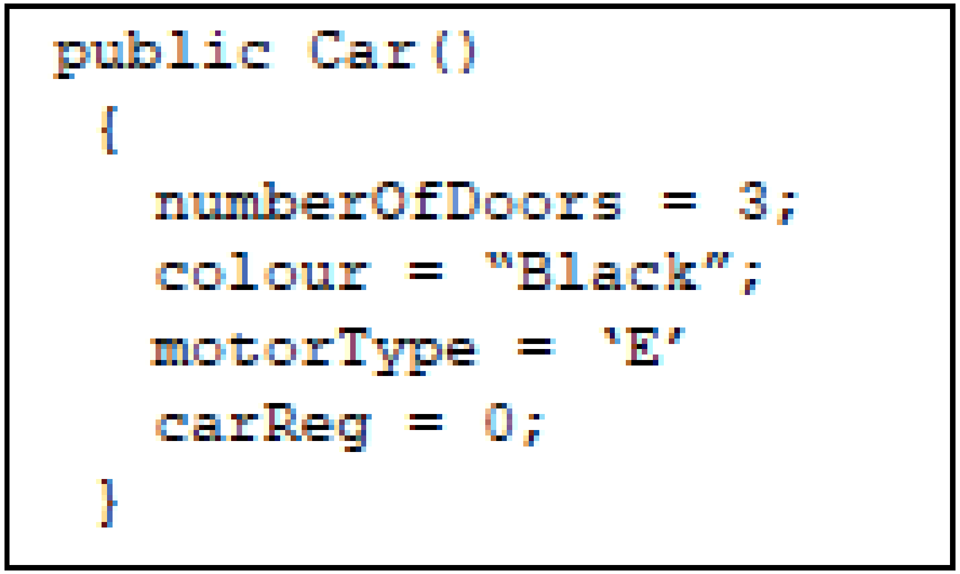
Using Non-Access Modifiers

Examples

- **Example #1** `public static void main(String[] args) { }`
- We use `static` here because we only want ONE `main()` method. Using `static` guarantees that that will be the case no matter what the user does with the class that contains `main()`.
- **Example #2** Let’s say we add a new attribute to our `Car` class. We will call it `carReg` and it is of type `int`.
  `private int carReg;`

Using Non-Access Modifiers

We must change our constructors and methods to take this new attribute into account. Our default constructor may change to:



```
public Car()
  {
    numberOfDoors = 3;
    colour = "Black";
    motorType = 'E'
    carReg = 0;
  }
```

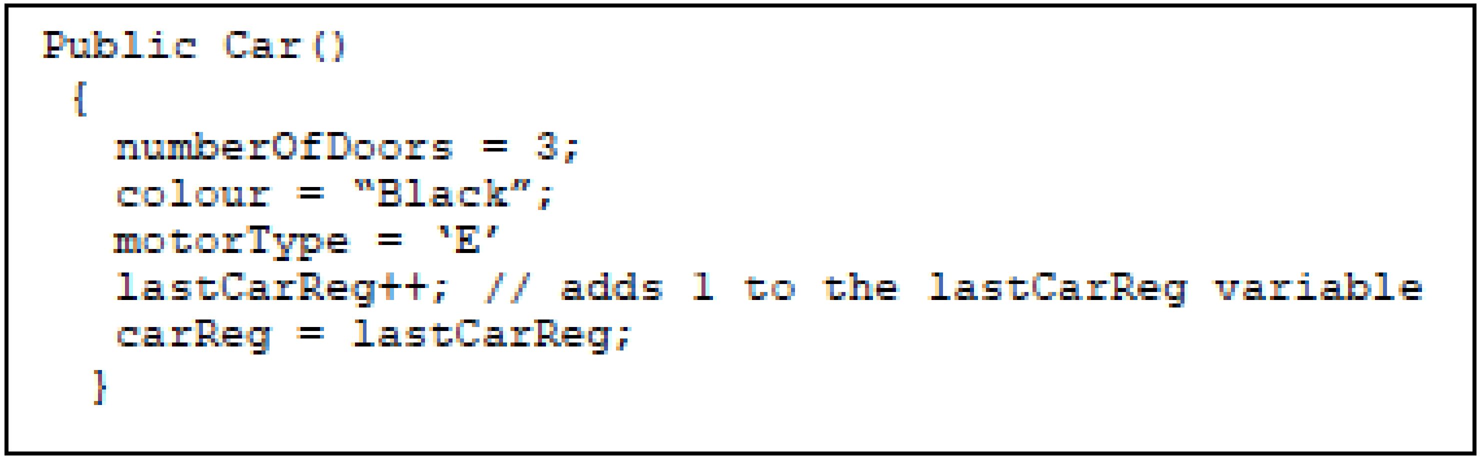
Using Non-Access Modifiers

- This means, every time we create a `Car` object using this constructor, each object will have the same value for `carReg` (i.e. 0).
- We know car registrations should be **unique**. We can alter our class code by adding a `static` variable that remembers the last `car reg` number used by the class.
- So we have

```
private static int lastCarReg = 1000;
```

Using Non-Access Modifiers

- Now we can update our constructor to add 1 to this `static` variable and then assign the new registration number to our carReg variable.



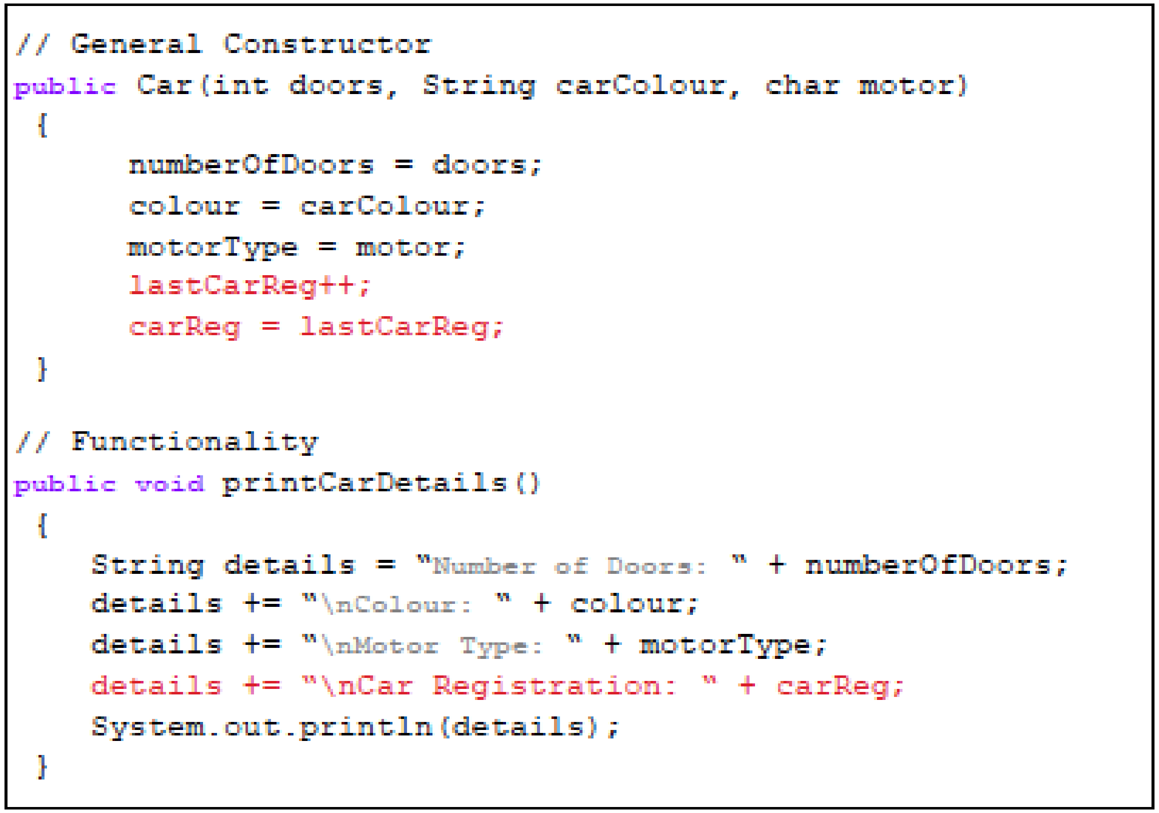
```
Public Car()
 {
   numberOfDoors = 3;
   colour = "Black";
   motorType = 'E'
   lastCarReg++; // adds 1 to the lastCarReg variable
   carReg = lastCarReg;
  }
```

Using Non-Access Modifiers

- This means that each new `Car` object created with this constructor will get a unique ID, starting with 1001 and going up by 1 each time this constructor is called.
- We should make sure to change our other constructors to do the same thing, so that the **behaviour is consistent** irrespective of the type of constructor used.

Using Non-Access Modifiers

The structure of the general constructor and the print details method are as follows:



```java
// General Constructor
public Car(int doors, String carColour, char motor)
 {
      numberOfDoors = doors;
      colour = carColour;
      motorType = motor;
      lastCarReg++;
      carReg = lastCarReg;
 }

// Functionality
public void printCarDetails()
 {
    String details = "Number of Doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    details += "\nCar Registration: " + carReg;
    System.out.println(details);
 }
```

Note on `Static` and Instance Variables

- Let us consider creating a class to represent a birth record for a child.

- A birth record would contain lots of details (attributes) e.g. the child’s name, mothers name, fathers name, hospital of birth and their PPS number.

- Each birth record that is created has its own copy of these attributes.

Note on `Static` and Instance Variables

- These are called **instance attributes** (variables), as in they belong to an instance (object – in this case a birth record) of the class.

- Every PPS number has to be unique and therefore when a new child is born they should get a new unique PPS number in their birth record.

- We need a `static` variable that represents the current/next PPS number to be given out.

Note on Static and Instance Variables

- Every time a new PPS number is needed we use the previous value plus one as the new PPS number `(currentPPS = previousPPS + 1).`
- This variable must be `static`, because we only ever want one copy of it.
- It doesn’t belong to any of the instances of the class (objects), it belongs to the birth record class but all of the objects of the class can access and use it.

Note on Static and Instance Variables

- Every time a new PPS number is needed we use the previous value plus one as the new PPS number `(currentPPS = previousPPS + 1).`
- This variable must be `static`, because we only ever want one copy of it.
- It doesn’t belong to any of the instances of the class (objects), it belongs to the birth record class but all of the objects of the class can access and use it.

`this` operator

When we created attributes for our `Car` class, we picked names that were meaningful and descriptive:

```
private int numberOfDoors;
private String colour;
private char motorType;
private int carReg;
```

- When it came to creating constructors, accessors and mutators, we had to pick different names for our parameters to describe the same attributes.
- This can be confusing:

```
public Car (int doors, String carColour,
char motor)
```

`this` operator

- We would prefer to be able to continue to use the same variable names.
- However, if we chose the same names for our parameters as we chose for our attributes, we will get a compiler error.
- However, if we use the special `this` keyword in Java then we can differentiate between variables that are passed as parameters and variables that belong to an instance of a class.
- For example, if we want to set the motor type of an instance of `Car` using the `motorType` mutator method, we could do the following:

`this` operator

- For example, if we want to set the motor type of an instance of `Car` using the `motorType` mutator method, we could do the following:

```
public void setMotorType(char motorType)
{
this.motorType = motorType;
}
```

`this` operator

- Using the this keyword indicated that the variable `motorType` on the left hand side of the assignment is the instance attribute variable and the variable
- `motorType` on the right hand side is the parameter variable belonging exclusively to the method `setMotorType()`
- This method takes the value from the method parameter `motorType` and saves it into the instance attribute variable `motorType`

this operator

The default constructor can be modified as follows:

```
public Car(int numberOfDoors, String colour,
char motorType)
{
this.numberOfDoors = numberOfDoors;
this.colour = colour;
this.motorType = motorType;
}
```

`Object class`

- In Java, every class is actually a sub class of a very important class.
- It is called the class `Object`.
- The `Object` class has a number of methods that are subsequently made available to all other classes.

Overriding the `toString()` Method

- In the **Object** Application Programming Interface (API), `toString()` is described as a method that “Returns a string representation of this Object”.

- It is a good practice to override this method and provide a string description of whatever object our class is encapsulating.

- All it means is that you return a String that contains all/some of the attributes values.

Overriding the `toString()` Method

- All it means is that you return a String that contains all/some of the attributes values.

- The calling method could then print these attributes. It is like the methods you have seen before for gathering/printing the attribute values of a class.

- Earlier we wrote a method `printCarDetails()` which printed the details of the car.

Overriding the `toString()` Method

```
Public void printCarDetails()
 {
    String details = "Number of Doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    System.out.println(details);
}
```

Let us now examine how to override the `toString()` method to perform the same functionality.

```
Public String toString()
 {
    String details = "Number of Doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    return details;
 }
```

Overriding the `toString()` Method

Let us look at our Car class again and the final product. The following two classes demonstrate:

(1) a definition for a car `(Car.java),`

(2) a method to interact/test this class `(testCar.java).`

- Let us assume that every car can be described by four attributes:
  - Number of doors
  - Colour
  - Motor type
  - Car registration

- The Car class has two constructors – one which takes no values (default) and one which takes three values, one for each attribute except `carReg`.

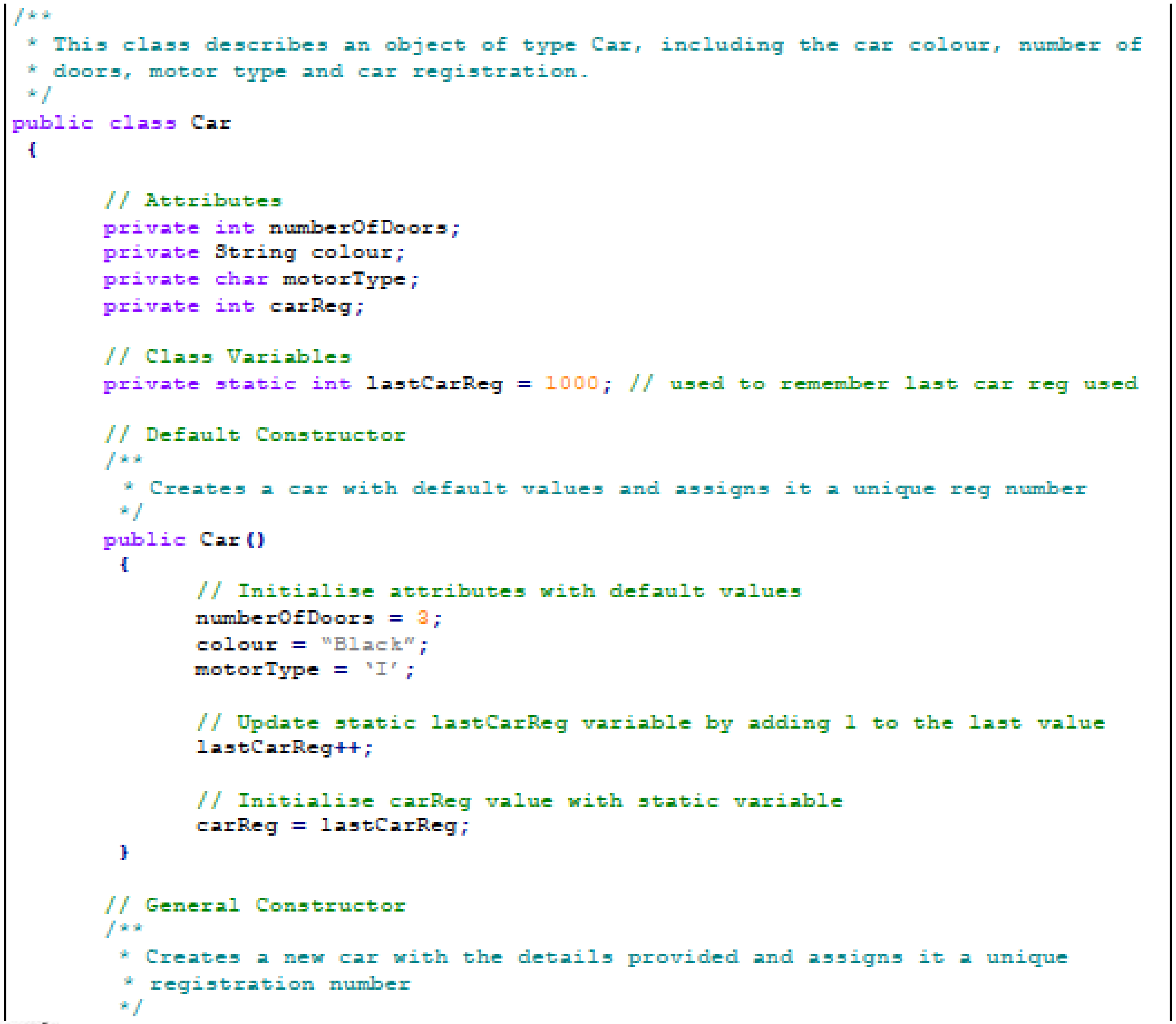
Maynooth University
National University of Ireland Maynooth

Overriding the `toString()` Method

- A car has methods to get and set the colour of the car, to get and set the type of motor get the number of doors and get the registration number of the car.

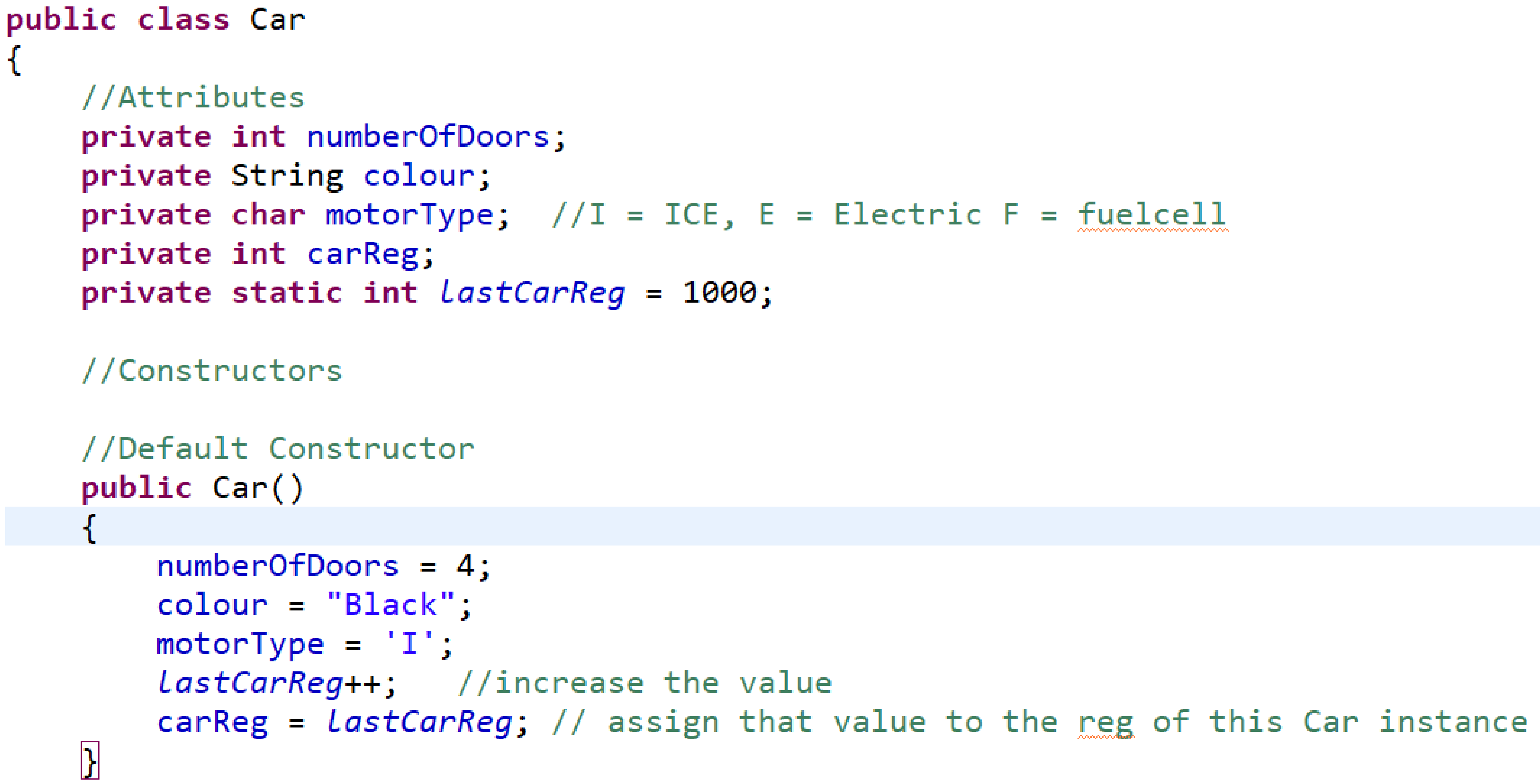
- It also contains a method to print the details of the car.

Car.java()



```
/**
 * This class describes an object of type Car, including the car colour, number of
 * doors, motor type and car registration.
 */
public class Car
 {

      // Attributes
      private int numberOfDoors;
      private String colour;
      private char motorType;
      private int carReg;

      // Class Variables
      private static int lastCarReg = 1000; // used to remember last car reg used

      // Default Constructor
      /**
       * Creates a car with default values and assigns it a unique reg number
       */
      public Car()
       {
            // Initialise attributes with default values
            numberOfDoors = 3;
            colour = "Black";
            motorType = 'I';

            // Update static lastCarReg variable by adding 1 to the last value
            lastCarReg++;

            // Initialise carReg value with static variable
            carReg = lastCarReg;
       }

      // General Constructor
      /**
       * Creates a new car with the details provided and assigns it a unique
       * registration number
       */
```

Car.java()

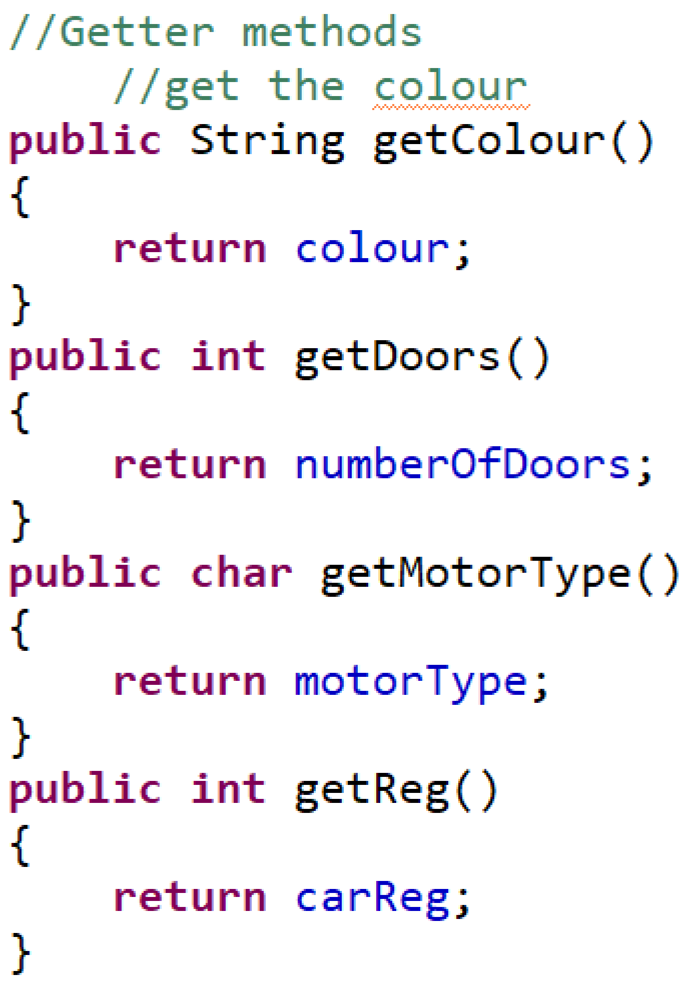


```java
public class Car
{
    //Attributes
    private int numberOfDoors;
    private String colour;
    private char motorType;  //I = ICE, E = Electric F = fuelcell
    private int carReg;
    private static int lastCarReg = 1000;

    //Constructors

    //Default Constructor
    public Car()
    {
        numberOfDoors = 4;
        colour = "Black";
        motorType = 'I';
        lastCarReg++;   //increase the value
        carReg = lastCarReg; // assign that value to the reg of this Car instance
    }
```

Car.java()

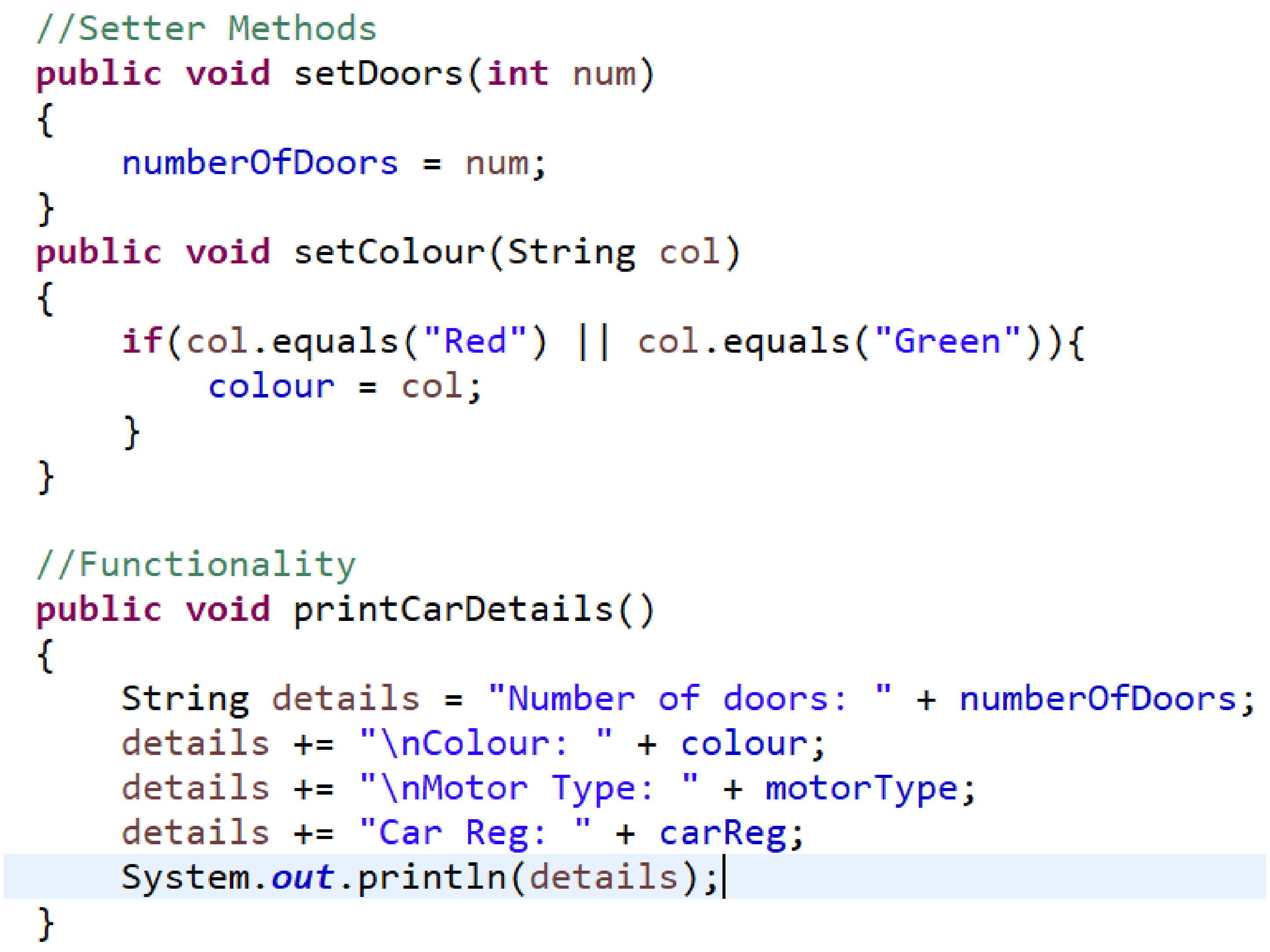
```
public Car(int numberOfDoors, String colour, char motorType)
{
    this.numberOfDoors = numberOfDoors;
    this.colour = colour;
    this.motorType = motorType;
    lastCarReg++;
    carReg = lastCarReg;
}

public Car(int doors)
{
    numberOfDoors = doors;
    colour = "Black";
    motorType = 'I';
}
```

Car.java()



```java
//Getter methods
    //get the colour
public String getColour()
{
    return colour;
}
public int getDoors()
{
    return numberOfDoors;
}
public char getMotorType()
{
    return motorType;
}
public int getReg()
{
    return carReg;
}
```

Car.java()



```
//Setter Methods
public void setDoors(int num)
{
    numberOfDoors = num;
}
public void setColour(String col)
{
    if(col.equals("Red") || col.equals("Green")){
        colour = col;
    }
}

//Functionality
public void printCarDetails()
{
    String details = "Number of doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    details += "Car Reg: " + carReg;
    System.out.println(details);
}
```

Maynooth University
National University of Ireland Maynooth

Car.java()

```java
public String toString()
{
    String details = "Number of doors: " + numberOfDoors;
    details += "\nColour: " + colour;
    details += "\nMotor Type: " + motorType;
    details += "Car Reg: " + carReg;

    return details;
}
```

testCar.java()

```java
public class TestCar{
    public static void main(String args[]){
        //Create a Car instance by calling the default constructor
        Car c1 = new Car();

        //Create another Car instance by calling our general constructor with 3 parameters
        Car c2 = new Car(5,"red",'E');
        Car c3 = new Car(16);

        //Call printCarDetails method on each car
        c1.printCarDetails();
        c2.printCarDetails();
        c3.printCarDetails();

        System.out.println("Colours:");
        System.out.println("The colour of Car C1 : " + c1.getColour());
        System.out.println("The colour of Car C2 : " + c2.getColour());
        System.out.println("The colour of Car C3 : " + c3.getColour());
        c1.setColour("Green");
        c3.setDoors(4);

        c1.printCarDetails();
        c3.printCarDetails();
        System.out.println(c1.toString());

    }
}
```

Classes and Objects : Summary

- A class can be considered like a template. This template can be the blueprint of something more general.
- The main components of a class are
  - A class Name
  - Attributes also known as instance variables
  - Constructor(s)
  - Accessor methods (also known as Getter methods)
  - Mutator methods (also known as Setter methods)
  - Functionality in the form of methods

Classes and Objects : Summary

- Every class should have one Default constructor and a General constructor
- When creating objects of the class we can use **both** the default and general constructor

```
Car c1 = new Car();
Car c2 = new Car(4, "blue", 'I');
```

- **Accessors** and **mutator** methods are used to **get and set** different attributes within the class.
- There are **two types** of modifiers
    1. Access Modifiers - sets levels of access to a variable, method, class or package
    2. Non-Access Modifiers - alters the variable, method, class or package in some other (non-access related) way

Classes and Objects: Summary

- `this` keyword in Java can be used to differentiate between variables that are passed as parameters and variables that belong to an instance of a class.
- It is good practice to override the `toString()` method in all classes to print out all information relating to the class.