Object-Oriented Programming

Inheritance

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

October 9, 2018

Table of Contents

Learning outcomes

After this lecture and the related practical students should...

- understand the concept of inheritance in programming
- be able to extend classes and alter inherited functionality
- understand the concept of polymorphism, and the use of the `instanceof` keyword with inheritance
- understand the concept of abstract classes and method

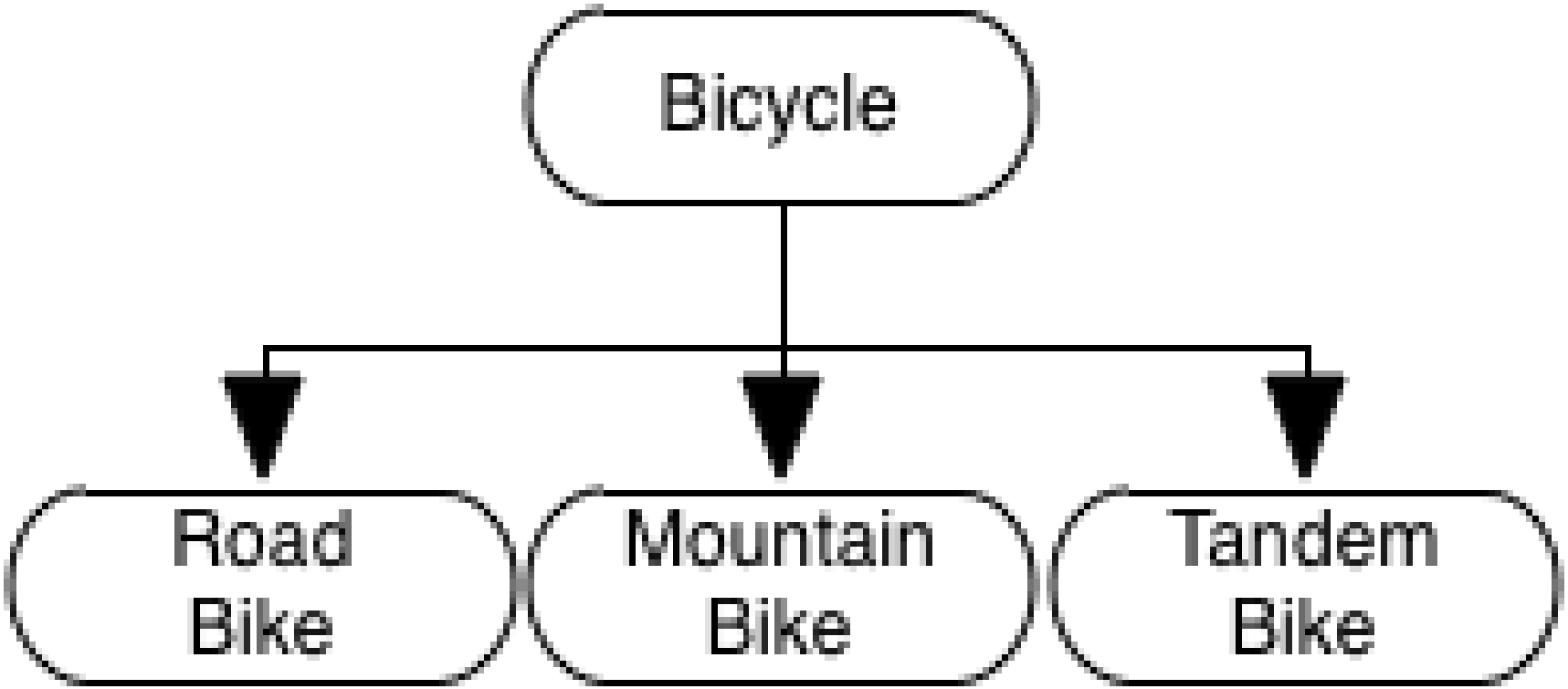
Code Reuse

- We can use interfaces to reuse code in different applications and situations
- Often classes will share common functionality
- Lets look at an example with bicycles
  - Mountain bikes, road bikes and tandem bikes all share certain characteristics of bicycles
  - But these bicycles also have additional features that make them different from each other

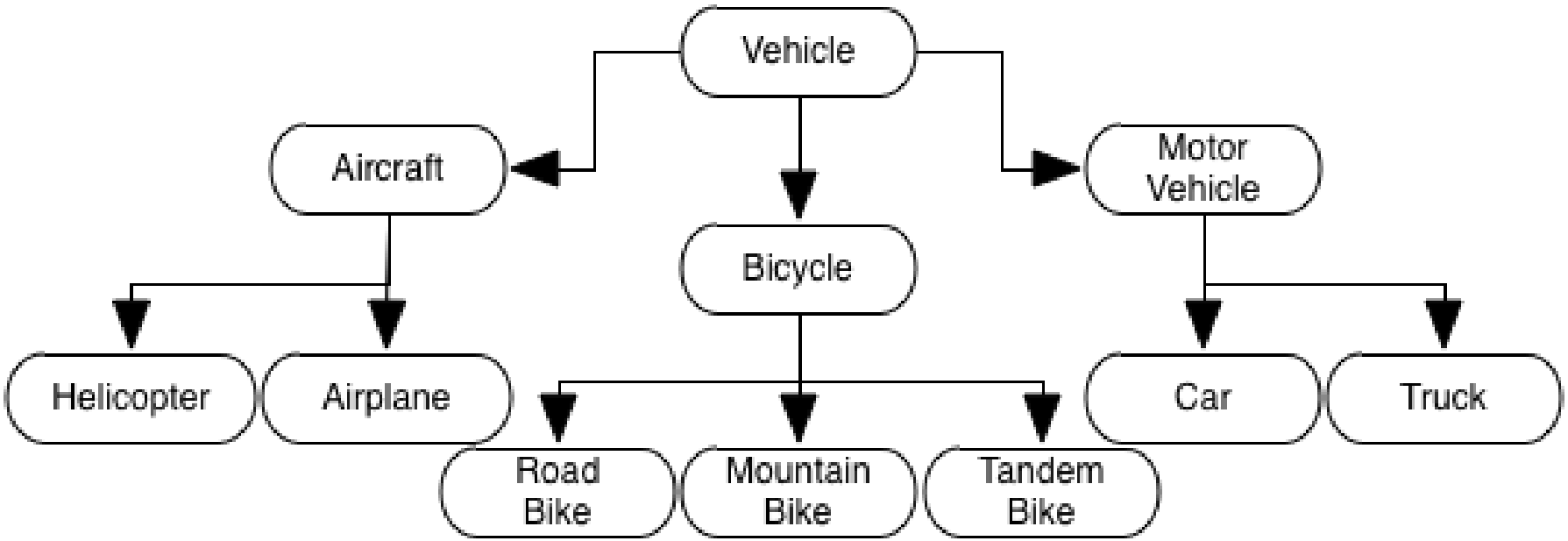
Hierarchies

- We associate things to each other using a hierarchies
- My bike is a mountain bike, but it could be described as a bike, or as a vehicle
- All of these descriptions are true, it performs all of the functions of a bike, so a mountain bike "is a" bike, it also performs all of the functions of a vehicle, so a mountain bike "is a" vehicle

Hierarchies



Hierarchies



Properties of Hierarchies

- The higher $\rightarrow$ more general
- The lower down $\rightarrow$ more specific
- One parent
- Many children

Hierarchies in OOP

- A class gets all functionality of classes directly above it in the hierarchy
- `MountainBike` contains the functionality of `Bicycle` and `Vehicle` classes
- If the `Bicycle` class has a method `getGear` and the `Vehicle` class has `getSpeed`, then the `MountainBike` class also contains both methods
- This is called inheritance

Inheritance in OOP

- Inheritance involves relationships between classes
- This relationship can be defined as “is a”
- The classes in these relationships are called the subclass and the superclass
- A bicycle “is a” vehicle - a vehicle “is not a” bicycle

Superclass and Subclass

- Each class can have only **one** superclass
- The class lower in the hierarchy is the subclass
- A class can have **many** subclasses
- This relationship is defined by the subclass

Syntax of Inheritance

- To define the "is a", we use the `extends` keyword

Example

```
1 public class MountainBike extends Bicycle {
2     // new fields and methods defining
3     // a mountain bike would go here
4 }
```

- `MountainBike` gets the variables and methods from `Bicycle`
- Focused on mountain bike

Extending

- The `MountainBike` class will extend the functionality of the `Bicycle` class
- Any code that we add to the `MountainBike` class increases the functionality of the class
- This can not change the `Bicycle` class

Inheritance Example

Superclass

```
1 public class A {
2   public void sayHi(){
3     System.out.println("Hello!");
4   }
5 }
```

Subclass

```
1 public class B extends A {
2   public void sayBye(){
3     System.out.println("Goodbye");
4   }
5 }
```

Usage

```
1 public static void main(String[] args){
2   A a = new A();
3   B b = new B();
4
5   a.sayHi();
6   b.sayHi();
7
8   b.sayBye();
9 }
```

Access Levels

- Instance variables are inherited from superclass
- If it is declared `private` it cannot be accessed
- The variable still exists, but we cannot access or change its value

Access Levels

- If we want to use this variable, we have to choose one of the other access modifiers
  - protected allows the variable to be used in the class and in the subclass
  - default allows the variable to be used anywhere within the same package
  - public allows the variable to be used anywhere
- If we wish to keep to the ideal of encapsulation, we should use protected

Inheritance Example

- We will develop a simple `BankAccount` class
  - Be able to deposit money
  - Be able to withdraw money
  - Be able to query the balance

BankAccount Class

```
1  public class BankAccount {
2    private double balance;
3
4    public void deposit(double amount){
5      balance += amount;
6    }
7    public void withdraw(double amount){
8      balance -= amount;
9    }
10   public double getBalance(){
11     return balance;
12   }
13 }
```

Inheritance Example

- There are many different types of bank accounts
  - A savings account will earn interest
  - A checking account will charge fees
- The functionality to withdraw and deposit money is still required

- We will create a subclass of the `BankAccount` class for each of these types of account

Checking Account

- The checking account adds the basic functionality to deduct fees from the customers account

Checking Account Class

```
1 public class CheckingAccount extends
    BankAccount {
2   public void deductFee(){
3     withdraw(1);
4   }
5 }
```

Savings Account

- The savings account adds the basic functionality to add interest to the customers account

Savings Account Class

```
1 public class SavingAccount extends BankAccount
    {
2   public void addInterest(){
3     double interest = getBalance() * 0.04;
4     deposit(interest);
5   }
6 }
```

Inheritance from Object

- Every class has one superclass
- `Object` is the top (root) of the hierarchy
- If you do not specify a superclass, then it is automatically the `Object` class

Methods from Object

- There are many methods which every class inherits from the object class, here are the most common
  - `Object clone()`
  - `boolean equals(Object obj)`
  - `int hashCode()`
  - `String toString()`
  - `void wait()`
  - `void notify()`
- Any class in Java can make use of these methods

Polymorphism

- We can use polymorphism with inheritance
- if C extends D, then C objects can be stored in a D type variable

Example

A `CheckingAccount` object can be stored in a variable of the following types;

- `CheckingAccount`
- `BankAccount`
- `Object`

Polymorphism Limitations

- Only type methods
- If a `CheckingAccount` object is stored in a `CheckingAccount` variable, we can use all its methods
- If a `CheckingAccount` object is stored in a `BankAccount` variable, we can use only the methods of the `BankAccount` class
- If a `CheckingAccount` object is stored in a `Object` variable, we can access only the methods of the `Object` class

Types of subclasses

- When we extend a class, we inherit the functionality of the original class
- The new class "is a" version of the original class
- The `instanceof` keyword works like this
- If we have a `CheckingAccount` object named b, what is the result of the expressions?
  - `(b instanceof CheckingAccount)`
  - `(b instanceof BankAccount)`
  - `(b instanceof Object)`

Types of Subclasses

- `instanceof` returns true for any of the following;
  - The class of the object itself
  - The class of the objects superclass
  - The class of the superclass's superclass, etc.
  - Any interface implemented by the class of the object
  - Any interface implemented by any of the objects superclass's

Changing Functionality

- When defining subclasses, we may want to change the inherited behaviour
- This means to change the functionality of the methods declared in the superclass
- This can usually be achieved by writing the same method again, but changing the functionality
- This process is called overriding a method

Overriding Methods

- The process of overriding a method the same as writing a new method, except:
  1. The name of the method must be the same
  2. The return type of the method must be the same
  3. The visibility of the method must be the same
  4. The parameters of the method must be the same

Overriding Example

- It is common for a checking account to charge a transaction fee
- We will change our `CheckingAccount` class so that every time withdraw or deposit is called we also called the `deductFee` method

CheckingAccount

```
1  public class CheckingAccount extends
       BankAccount {
2    public void deductFee(){
3      withdraw(1);
4    }
5    public void deposit(double amt){
6      balance += amt;
7      deductFee();
8    }
9    public void withdraw(double amt){
10     balance -= amt;
11     deductFee();
12   }
13 }
```

Calling the Wrong Method

- `dedutFees` uses `withdraw`
- `withdraw` uses `dedutFees`
- What we really want is to specify that we want to use the method that is defined in the superclass (`BankAccount`)
- This can be achieved using the keyword `super`

Super

- The keyword super refers to the direct superclass
  - The class in the declaration
- There are three situations where is can be used
  - To access an instance variable declared in the superclass
  - To call methods declared in the superclass
  - During construction of a subclass to pass parameters to the superclass constructor

Super for instance variables

- Only required if you use the same variable name
- It is bad programming practice to use the same name for instance variable in the subclass
- This can cause much confusion and the behaviour of methods may not be what you expect
- DO NOT DO THIS!!!

Super for overridden Methods

- The syntax is like `this`.

Syntax

`super.methodName();` will call `methodName` in the superclass

Example

```
super.withdraw(amt);
```

CheckingAccount Updated

```
1  public class CheckingAccount extends
       BankAccount {
2    public void deductFee(){
3      super.withdraw(1);
4    }
5    public void deposit(double amt){
6      super.deposit(amt);
7      deductFee();
8    }
9    public void withdraw(double amt){
10     super.withdraw(amt);
11     deductFee();
12   }
13 }
```

Superclass Method

- We can use the super keyword to choose the implementation of the method we are calling
- But we can only choose between our overridden method and the version in our superclass
  - Calling `withdraw(1);` will call our overridden method
  - Calling `super.withdraw(1);` will call the withdraw method in the superclass
- This means that we can call a version of the method which does not deduct fees from the account

Superclass Constructor

- All objects must be constructed
- If a class requires parameters for construction, we must supply them
- When we extend a class we must call its constructor and pass it the required parameters
- If a superclass requires parameters, we must pass these from the constructor in the subclass

Super Constructors

- In the constructor of a subclass, the first thing we must do is called the constructor of the superclass
- The only keyword we need is `super` and in brackets we supply all of the required variables
- The parameters supplied must match one of the constructors in the superclass

Super Constructor Syntax

```
1 public ClassName(parameters) {
2   super(parameters);
3   ...
4 }
```

Example

Lets look at the `BankAccount` Example

BankAccount Constructor

```
1 public BankAccount(double initial){
2   balance = initial;
3 }
```

Subclass Constructors

```
1 public CheckingAccount(double b){
2   super(b);
3 }
```

```
1 public SavingAccount(){
2   super(0);
3 }
```

Method Overloading

- In Java, it is possible to have many methods with the same name
- However, each must have different parameters
- Java will choose the correct one to execute based on the parameters we supply
- An excellent example of this is the `PrintStream` class
- This is the class that `System.out` belongs to

Method Overloading in PrintStream

- In the PrintStream class there are many methods with the name print and println
- Each method requires a different type of parameter
  - One will accept an integer
  - One will accept a String
  - One will accept an object
- This is why we are able to print the values of variables without any format specifiers

Examples

```
System.out.println(123);
System.out.println("Hi");
System.out.println(b);
```

Overriding Object

- These methods inherited from the Object class are very useful because they can be called on any object
- The most useful of these methods are
  - `boolean equals(Object obj)`
  - `int hashCode()`
  - `String toString()`
- Each method has a default implementation, but they are usually not very good
  - `toString` simply prints the name of the class and the memory address of the object

Overriding toString

- The toString method is called any time we print the value of an object

Example

```
1 BankAccount b = new BankAccount(1000);
2 ...
3 System.out.println(b);
```

- But it would be better if the message printed was something to do with the actual account and not the address of the object

BankAccount toString

```java
public class BankAccount {
  private double balance;
  private long accountNumber;
  private String accountName;
  public BankAccount(long ac, String n){
    accountNumber = ac;
    accountName = n;
    balance = 0;
  }
```

```java
public String toString(){
  String message = "Bank Account of " + accountName +
   "\n";
  message += "Account Number: "+ accountNumber + "\n";
  message += "Account Balance : " + balance + "\n";
  return message;
}
```

Overriding toString

- After overriding `toString`, we can perform the same steps as before

Example

```
1 BankAccount b = new BankAccount(12387587L, "Sean");
2 ...
3 System.out.println(b);
```

- This time we get a nicer output including the name of the account holder, the account number and the current balance

```
1 Bank Account of Sean
2 Account Number: 12387587
3 Account Balance: 0.0
```

Overriding equals

- Perhaps one of the most useful methods to use is the equals method
- The equals method should tell us if two objects are the same
- The default implementation only compares the memory addresses of the two objects

Example

```
1 BankAccount a = new BankAccount(1354654L, "Sean");
2 BankAccount b = new BankAccount(1354654L, "Sean");
3
4 if(a.equals(b)){
5   System.out.println("The Accounts are the same");
6 }
```

Overriding equals

- For our own classes we should override this method so that it compares the objects based on their instance variables
- However, the parameter of the method must always be the type `Object`
- This means that we must check that the object belongs to the correct class before comparing it
- Once converted, we only need to compare the account number (this uniquely identifies the account)
- For other classes we may need to compare multiple instance variables

BankAccount equals Method

Implementation

```
1 public boolean equals(Object obj){
2   if(obj instanceof BankAccount){
3     BankAccount b = (BankAccount)obj;
4     return b.accountNumber ==
    this.accountNumber;
5   } else {
6     return false;
7   }
8 }
```

BankAccount equals Method

- Line 2 checks to see if the parameter is a `BankAccount` class
- Line 3 creates a `BankAccount` reference and performs a typecast so we can access the instance variables of the parameter
- Line 4 performs the actual comparison and returns the result
- Line 5 returns false if the object is not based on the `BankAccount` class

Inheritance in Interfaces

- We can also use inheritance with interfaces
- Inheritance in interfaces is much simpler because interfaces cannot contain implemented methods
- When one interface inherits from another, it is simply adding further requirements for any class that implements the subinterface

```
1 public interface A {
2   void methodA();
3 }
```

```
1 public interface B extends A {
2   void methodB();
3 }
```

Abstract

- The term abstract means an idea or concept
- Something abstract does not exist in the real world
- For example, the concept of a phone is abstract
  - The idea is something that only exists in thought
  - But many devices are constructed, that are different implementations of the abstract phone concept

Abstract in Java

- The abstract keyword in Java means something that is not implemented
- It can be applied to classes and methods

Abstract Classes

- An abstract class is similar to an interface in some ways
- For example you cannot create an object based on an abstract class
- Attempting to construct an abstract class will cause a compiler error
- However you can extend an abstract class and inherit functionality from it
- An abstract class can contain abstract methods, which do not have an implementation

Differences between Abstract classes and Interfaces

- Abstract classes can contain functional methods
- Abstract classes can contain instance variables
- If a method has no implementation, it must be declared abstract
- A class that contains an abstract method must be declared abstract
- If you extend an abstract class you must either provide an implementation for all abstract methods inherited or also be declared abstract

Abstract Syntax

Declaring an Abstract class

To declare a class as abstract, place the keyword abstract before class

```
1 public abstract class AbstractShape {
2     ...
3 }
```

Declaring an Abstract Method

To declare a method as abstract, place the keyword abstract before the return type

```
1 public abstract void move(int x, int y);
```

Abstract Example

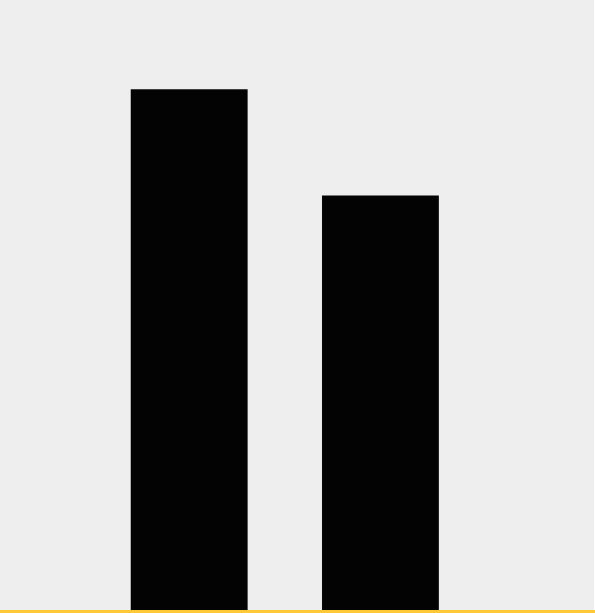
```
1  public abstract class Abstract2DDrawing {
2    protected int x, y;
3    public Abstract2DDrawing(int x, int y){
4      this.x = x;
5      this.y = y;
6    }
7    public void move(int x, int y){
8      this.x += x; this.y += y;
9    }
10   public abstract void draw(Graphics g);
11 }
```

Extending an Abstract Class

```
1  public class Building extends
       Abstract2DDrawing{
2    int w, h;
3    public Building(int x, int y, int w, int h) {
4      super(x, y);
5      this.w = w;
6      this.h = h;
7    }
8    public void draw(Graphics g) {
9      g.fillRect(x, y, w, h);
10   }
11   public void resize(int x, int y){
12     w += x; h += y;
13   }
14 }
```

Using the extended class

- `Building b = new Building(70, 50, 55, 350);`
- `b.draw(g);`
- `Building c = new Building(160, 50, 55, 300);`
- `c.draw(g);`



The Final keyword

- In Lecture 5 we saw that the keyword final used to make the value of a variable permanent
- The same keyword is used to change inheritance
- It can be used for both classes and methods

Final Classes

- A final class is one that cannot be extended
- This is used when we create a class for others to use, but we do not want them to be able to change the functionality

Final Class Syntax

```
1 public final class BankAccount{
2     ...
3 }
```

- Once the final keyword is used in this way, we define code exactly the same as before, however we know that this code can never be overridden

Why Final Class?

- A typical example of a final class would be a class representing a bank account
- Once we have written the code that defines how money is deposited or interest is applied, we do not want the user of the class to be able to change how we have implemented this
- Without the class being declared final, they could simply extend the class and override a method designed to apply a fee to the account, they could then replace this with code that adds money to the account instead

Final Methods

- If we want to allow the programmers using our code to create subclasses, but we still want to prevent them from changing some of the methods, we can define these methods as final
- This means that when the class is extended, the final methods cannot be overridden
- A method can be declared final by placing the keyword final before the return type of the method

Final Method Syntax

```
1 public final boolean
     checkPassword(String p){
2    ...
  }
```

Calling Methods in Constructor

- In Java, it is perfectly acceptable to call other methods from any code within a class, this is also true for the constructor of a class
- However, any method that you call from a constructor should be declared as final
- This is because when the class is extended, the functionality of this classes might be overridden
- We will not know what effect this might have on the constructor
- Declaring the methods as final will prevent this problem