Object Oriented Programming

Introduction to OOP

- **Object-Oriented Programming (OOP)** is the term used to describe a programming approach based on **objects** and **classes**.
- The object-oriented paradigm allows us to organise software as a collection of objects that consist of both **data** and **behaviour**.
- This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

- Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features.
- It is widely accepted that object-oriented programming is the most important and powerful way of creating software.
- The object-oriented programming approach encourages:
  - **Modularisation**: where the application can be decomposed into modules.
  - **Software re-use**: where an application can be composed from existing and new modules.

- An object-oriented programming language generally supports five main features:
  - Classes
  - Objects
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

- If we think of a real-world object, such as a television, it will have several features and properties:
  - We do not have to open the case to use it.
  - We have some controls to use it (buttons on the box, or a remote control).
  - We can still understand the concept of a television, even if it is connected to a DVD player.
  - It is complete when we purchase it, with any external requirements well documented.
  - The TV will not crash!

This compares very well to the notion of a class

- A class should:
  - Provide a well-defined interface - such as the remote control of the television.
  - Represent a clear concept - such as the concept of a television.
  - Be complete and well-documented - the television should have a plug and should have a manual that documents all features.
  - The code should be robust - it should not crash, like the television.

- Classes allow us a way to represent complex structures within a programming language. They have two components:
  - **States** - (or data) are the values that the object has.
  - **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

Example **States**

Example **Methods**

Television

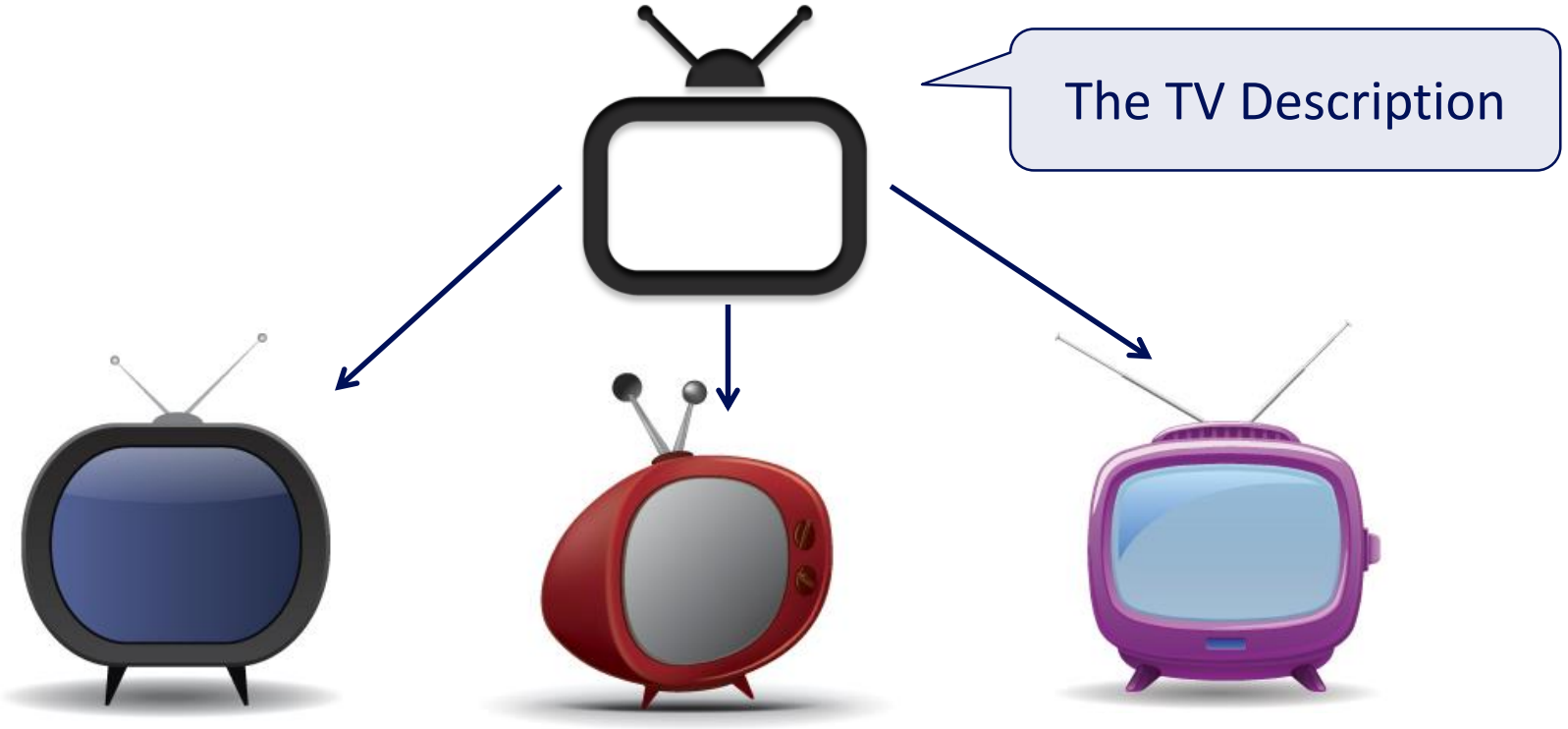
- channelNumber : int
- onOff : boolean
- volumeLevel : int

+ **changeChannel**(int channel) : void
+ **changeVolume**(int level) : void
+ **muteVolume**(boolean isMute) : void
+ **powerOff**() : void
+ **powerOn**() : void

Unified Modelling Language (**UML**) representation of the Television class

- An **instance of a class** is called an **object**.
- You could think of a class as the description of a concept, and an object as the realisation of this description to create an independent distinguishable entity.
- For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realisation of these plans into a real-world physical television.
- There would be one set of plans (the class), but there could be thousands of real-world televisions (objects).
- Objects are concrete (a real-world object, a file on a computer).

Object Example



An example where the Television class description is realised into several television objects. These objects should have their own identity and are independent from each other. For example, if the channel is changed on one television it will not change on the other televisions.

The syntax for declaring a class

ClassName.java

```
class <class_name> {

  // Declare the list of attributes
  <data_type> <attribute_name>;
  ...

  // Declare the list of methods
  public <return_value_name> <method_name>(<parameter_type> <parameter_name>, ...) {
      ...
  }
  ...
}
```

Circle.java

```java
public class Circle {    // class name
  double radius;        // variables
  String color;

  // methods
  double getRadius() {
      ......
  }

  double getArea() {
      ......
  }
}
```

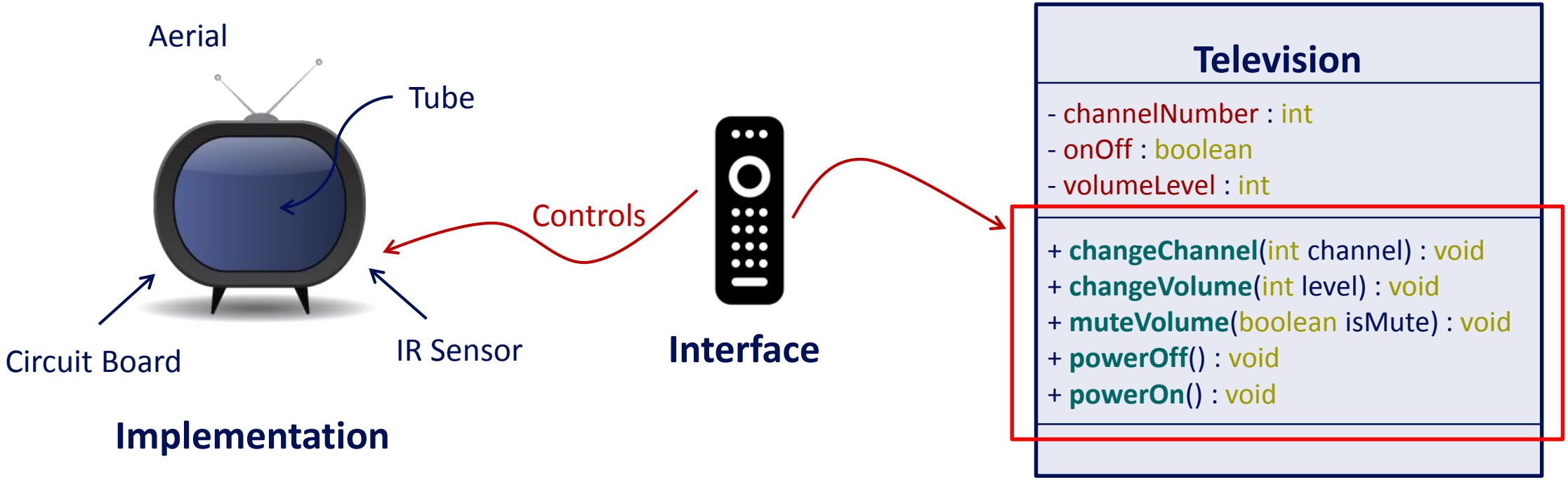
TestCircle.java

```java
  // Suppose that the class Circle has variables
  // radius and color, and methods getArea() and
  // getRadius().
  // Declare and construct instances circle1 and
  // circle2 of the class Circle
  Circle circle1 = new Circle();
  Circle circle2 = new Circle();

  // Invoke member methods for the instance c1
  // via dot operator
  System.out.println(circle1.getArea());
  System.out.println(circle1.getRadius());

  // Reference member variables for instance
  // circle2 via dot operator
  circle2.radius = 5.0;
  circle2.color = "blue";
```

Encapsulation

- The object-oriented paradigm encourages **encapsulation**.
- Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be **hidden**, so that we don't need to understand how the object works.
- All we need to understand is the **interface** that is provided for us.
- You can think of this in the case of the Television class:
  - The functionality of the television is hidden from us, but we are provided with a remote control, or set of controls for interacting with the television, providing a high level of **abstraction**.
  - There is no requirement to understand how the signal is decoded from the aerial and converted into a picture to be displayed on the screen before you can use the television.

- There is a sub-set of functionality that the user is allowed to call, termed the **interface**. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television..
- The full implemenation of a class is the sum of the **public interface** plus the **private implementation**.

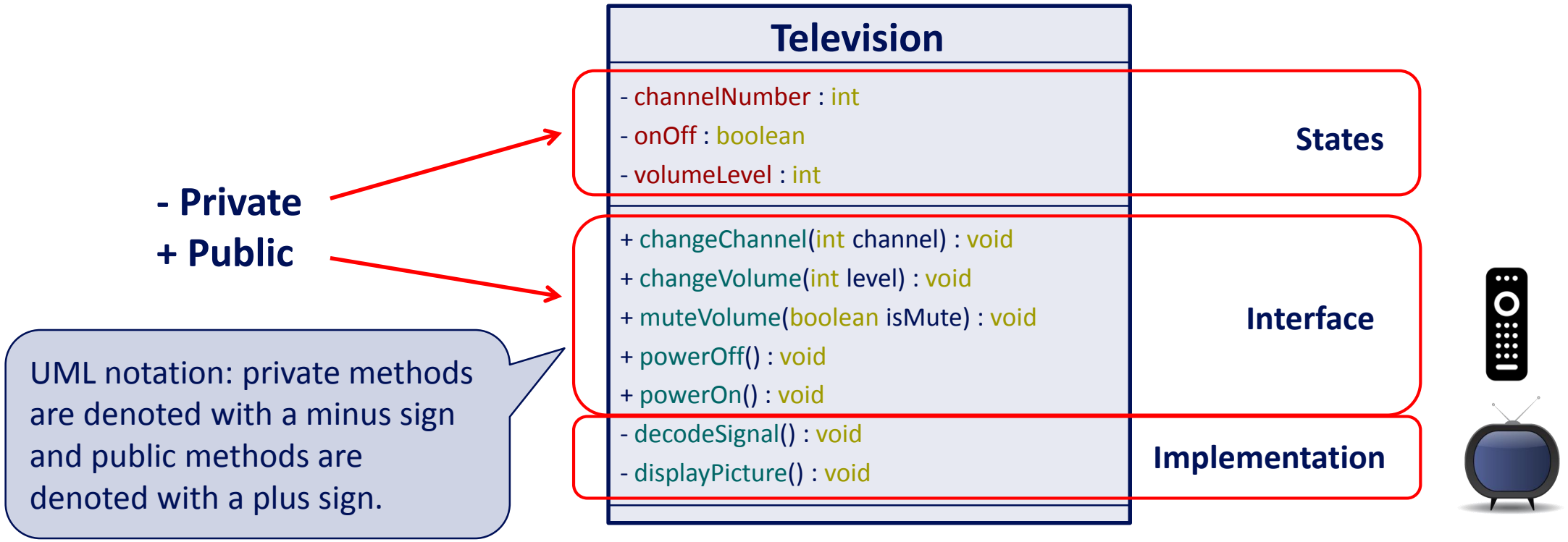


- **Encapsulation** is the term used to describe the way that the interface is separated from the implementation. You can think of encapsulation as "**data-hiding**", allowing certain parts of an object to be visible, while other parts remain hidden. This has advantages for both the user and the programmer.

  - For the **user** (who could be another programmer):
    - The user need only understand the interface.
    - The user need not understand how the implementation works or was created.
  - For the **programmer**:
    - The programmer can change the implementation, but need not notify the user.

- So, providing the programmer does not change the interface in any way, the user will be unaware of any changes, except maybe a minor change in the actual functionality of the application.

- We can identify a level of 'hiding' of particular methods or states within a class using the public, private and protected keywords:
  - **public methods** - describe the interface.
  - **private methods** - describe the implementation.



- The private methods would be methods written that are part of the inner workings of the television, but need not be understood by the user.
- For example, the user would need to call the powerOn() method but the private displayPicture() method would also be called, but internally as required, not directly by the user. This method is therefore not added to the interface, but hidden internally in the implementation by using the private keyword.

Encapsulation example

Circle.java

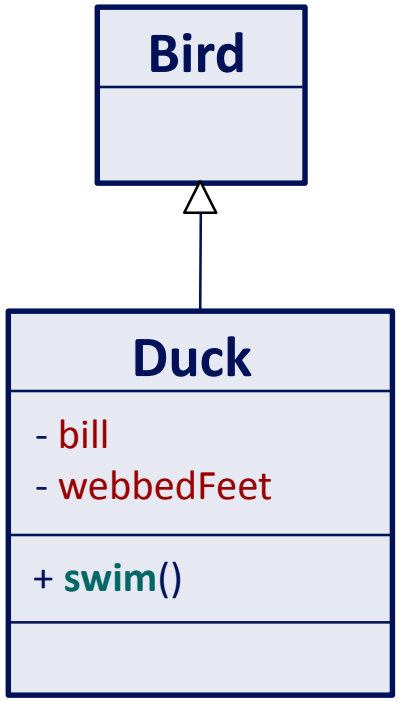
```java
public class Circle {
  // Private instance variables
  private double radius;
  private String color;

  // Constructors
  public Circle() {    // Default constructor
    radius = 1.0;
    color = "red";
  }

  public double getRadius() {  // Getter for radius
    return radius;
  }

  public String getColor() {   // Getter for color
    return color;
  }

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public void setColor(String newColor) { // Setter for color
     color = newColor;
  }

  public void setRadius(double newRadius) { // Setter for radius
     radius = newRadius;
  }
}
```

TestCircle.java

```java
public class TestCircle {
  public static void main(String[] args) { // Program entry point
    // Declare and construct an instance of the Circle class
    // called cicle
    Circle circle = new Circle(); // Use 1st constructor
    System.out.println("The radius is: " + circle.getRadius());
    // The radius is: 1.0

    System.out.println("The color is: " + circle.getColor());
    // The color is: red

    System.out.printf("The area is: %.2f%n", circle.getArea());
    // The area is: 3.14

    circle.setRadius(2.0);
    circle.setColor("blue");

    System.out.println("The radius is: " + circle.getRadius());
    // The radius is: 2.0

    System.out.println("The color is: " + circle.getColor());
    // The color is: blue

    System.out.printf("The area is: %.2f%n", circle.getArea());
    // The area is: 12.57
  }
}
```

Inheritance

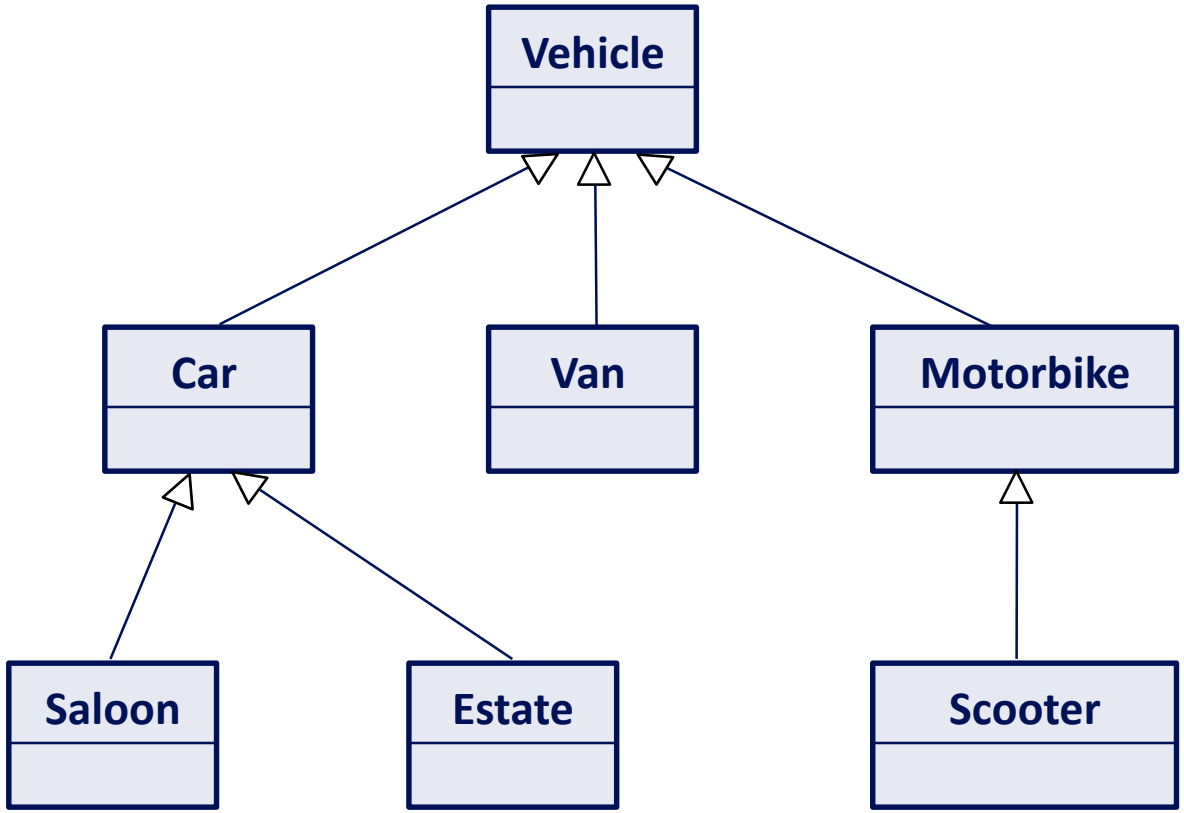
- If we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions.
- The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.
- Humans use this concept in categorising objects and descriptions. For example, you may have answered the question - "What is a duck?", with "a bird that swims", or even more accurately, "a bird that swims, with webbed feet, and a bill instead of a beak". So we could say that a Duck is a Bird that swims.
- The figure illustrates the inheritance relationship between a Duck and a Bird. In effect we can say that a Duck **is-a** special type of Bird.



The figure illustrates the inheritance relationship between a Duck and a Bird.

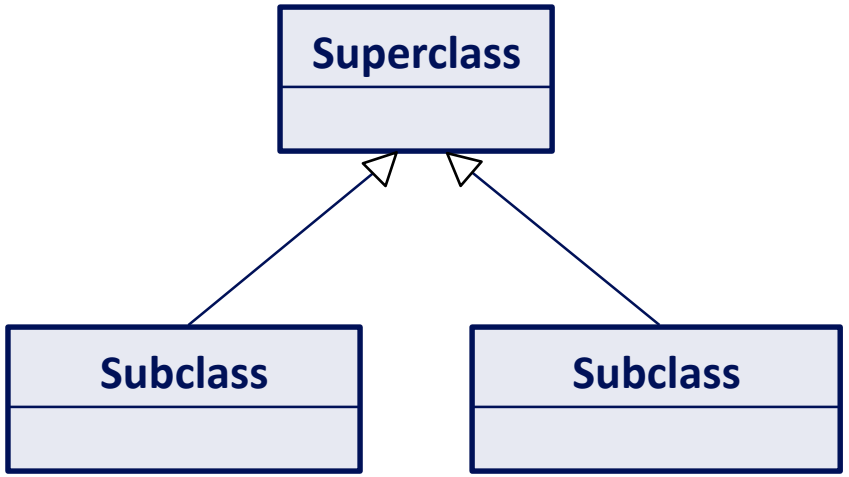
- For example: if were to be given an unstructured group of descriptions such as **Car**, **Saloon**, **Estate**, **Van**, **Vehicle**, **Motorbike** and **Scooter**, and asked to organise these descriptions by their differences.

- You might say that a **Saloon** car **is-a** **Car** but has a long boot, whereas an Estate car **is-a** car with a very large boot.



An example of how we may organise these descriptions using inheritance.

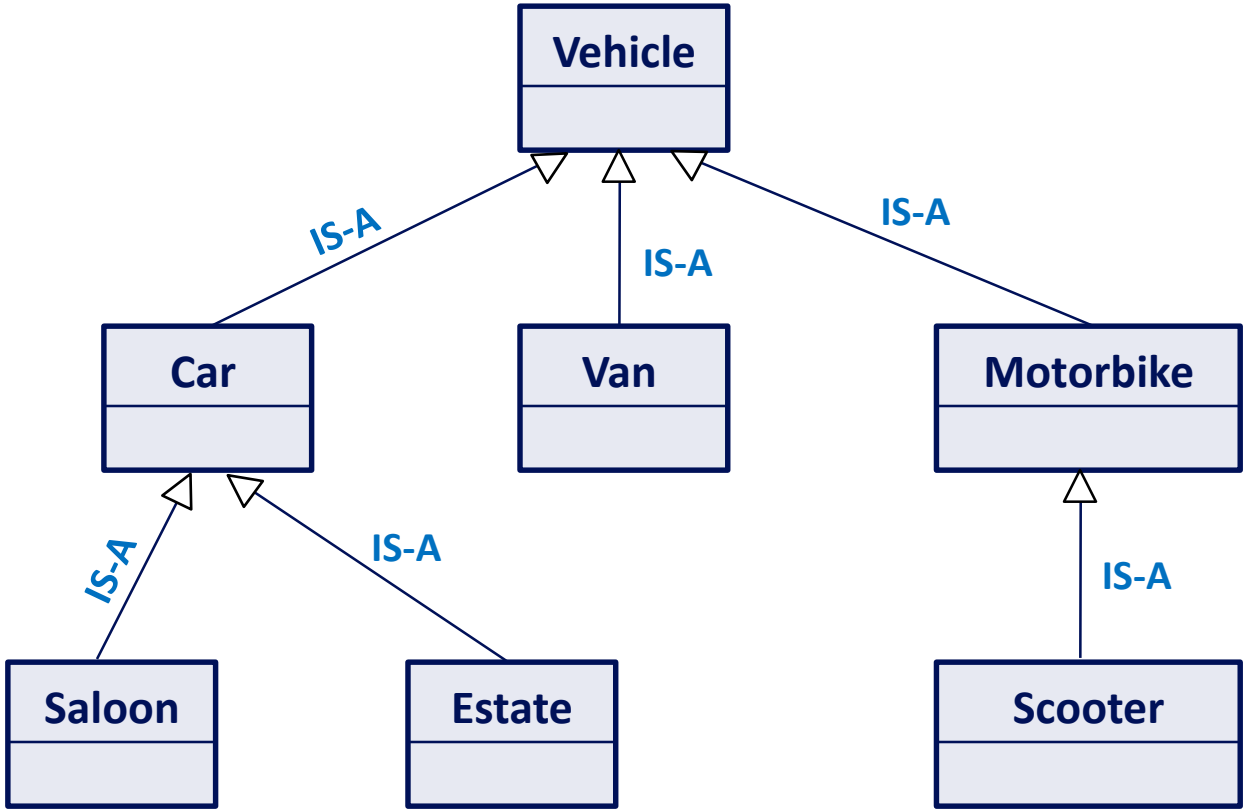
- So we can describe this relationship as a child/parent relationship.

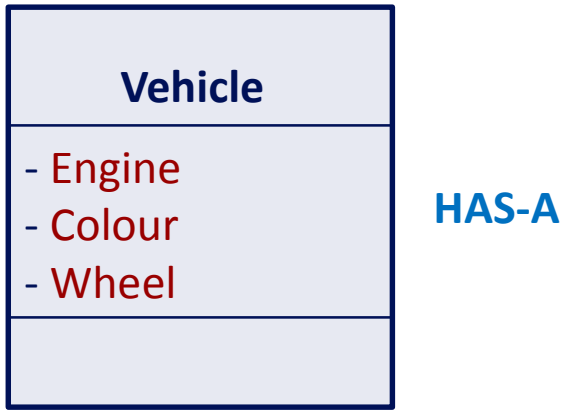


The Car class is a child of the Vehicle class, so Car inherits from Vehicle.

Illustrates the relationship between a **superclass** and a **subclass**. A subclass inherits from a superclass

- One way to determine that you have organised your classes correctly is to check them using the "**IS-A**" and "**HAS-A**" relationship checks. It is easy to confuse objects within a class and children of classes when you first begin programming with an OOP methodology.



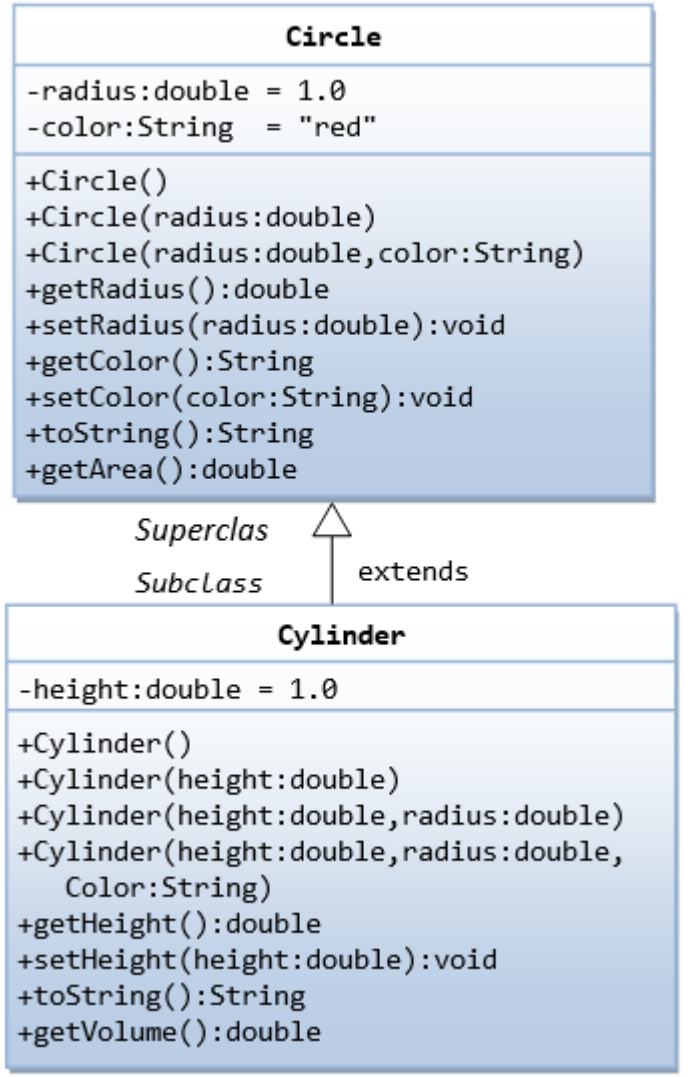


HAS-A

- The **IS-A** relationship describes the **inheritance**.
- We can say, "A Car **IS-A** Vehicle" and "A SaloonCar **IS-A** Car", so all relationships are correct.
- The **HAS-A** relationship describes the **composition** (or **aggregation**) of a class. So we can say "An Vehicle **HAS-A** a Engine", or "An Engine, Colour and Wheels **IS-A-PART-OF** a Vehicle". This is the case even though an Engine is also a class! where there could be many different descriptions of an Engine - Petrol, Diesel, etc.
- So, using inheritance the programmer can:
  - **Inherit** a behaviour and **add** further specialised behaviour - for example, a Car **IS-A** Vehicle with the addition of four Wheel objects, Seats, etc.
  - **Inherit** a behaviour and **replace** it - for example, the SaloonCar class will inherit from Car and provide a new "boot" implementation.
  - Cut down on the amount of code that needs to be written and debugged - for example, in this case only the differences are detailed, a SaloonCar is essentially identical to the Car, with only the differences requiring description.

Inheritance example

- We derive a subclass called Cylinder from the superclass Circle.
- It is important to note that we reuse the class Circle. Reusability is one of the most important properties of OOP. (Why reinvent the wheels?)
- The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle.
- The class Cylinder further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors.



Cylinder.java

```java
public class Cylinder extends Circle {
  private double height;

  // Constructors
  public Cylinder() {
    super(); // invoke superclass' constructor Circle()
    this.height = 1.0;
  }

  // Getter and Setter
  public double getHeight() {
    return this.height;
  }

  public void setHeight(double height) {
    this.height = height;
  }

  // Returns the volume of this Cylinder
  public double getVolume() {
    return getArea() * height; // Use Circle's getArea()
  }

  // Returns a self-descriptive String
  public String toString() {
    return "This is a Cylinder"; // to be refined later
  }
}
```

TestCylinder.java

```java
public class TestCylinder {
  public static void main(String[] args) {
    Cylinder cylinder = new Cylinder();
    // Construced a Circle with Circle()
    // Constructed a Cylinder with Cylinder()

    System.out.println("Radius is " + cylinder.getRadius()
        + ", Height is " + cylinder.getHeight() + ", Color is  "
        + cylinder.getColor() + ", Base area is "
        + cylinder.getArea() + ", Volume is " + cylinder.getVolume());
    // Radius is 1.0, Height is 1.0, Color is red,
    // Base area is 3.141592653589793, Volume is 3.141592653589793

    sylinder.setHeight(5.0);
    sylinder.setRadius(2.0);

    System.out.println("Radius is " + cylinder.getRadius()
        + ", Height is “ + cylinder.getHeight() + ", Color is  "
        + cylinder.getColor() + ", Base area is " + cylinder.getArea()
        + ", Volume is " + cylinder.getVolume());
    // Radius is 2.0, Height is 5.0, Color is red,
    // Base area is 12.566370614359172, Volume is 62.83185307179586
  }
}
```

Polymorphism

- When a class inherits from another class it inherits both the **states** and **methods** of that class
- In the case of the Car class inheriting from the Vehicle class, the Car class inherits the methods of the Vehicle class, such as engineStart(), gearChange(), lightsOn(), etc.
- The Car class will also inherit the **states** of the Vehicle class, such as isEngineOn, isLightsOn, numberWheels, etc.

- **Polymorphism** means "multiple forms". It comes from Greek word "poly" (means many) and "morphos" (means form).
- In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in **different classes**, or the same method name can be used in the same class with slightly **different paramaters**.
- There are two forms of polymorphism, **overriding** and **overloading**.

- Overloading is the second form of polymorphism. The **same method name** can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler.
- For example
  - Two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality.

Overloading: the types of parameters differ

```
add(int x, int y);
add(String x, String y);
```

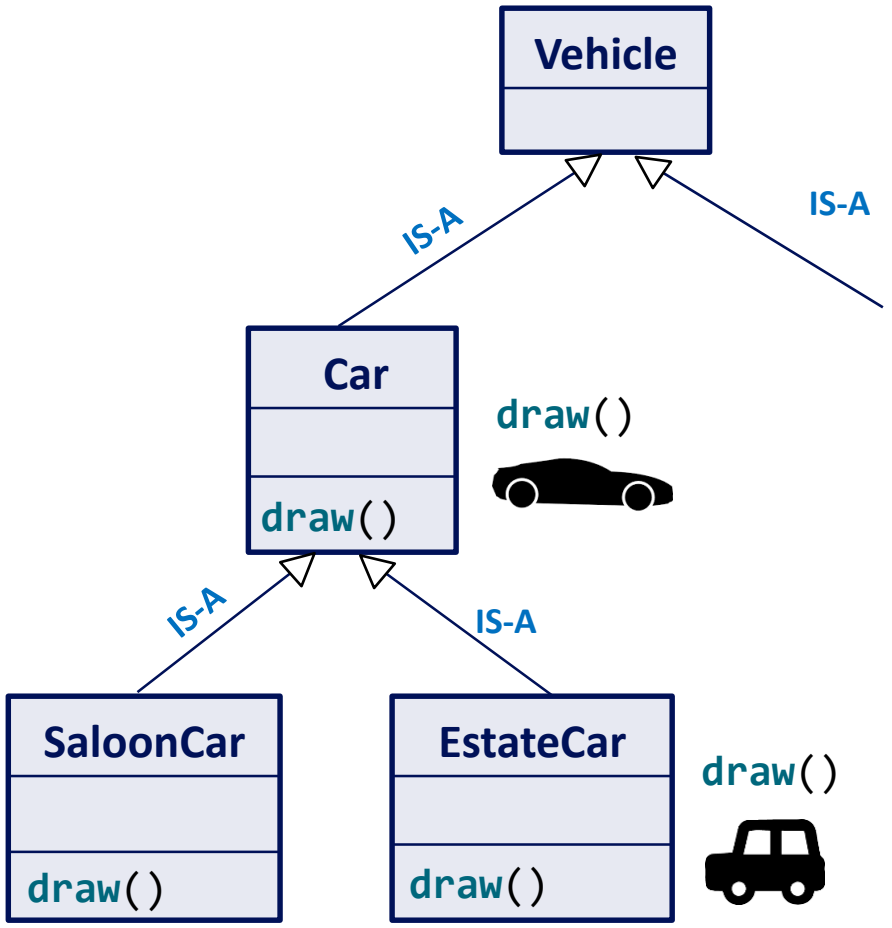
  - The number of arguments can also determine which method should be run. The first method may simply display the current channel number, but the second method will set the channel number to the number passed.

Overloading: number of parameters differ

```
channel();
channel(int x);
```

- A derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation.
- **Overriding** is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.
- Overriding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.

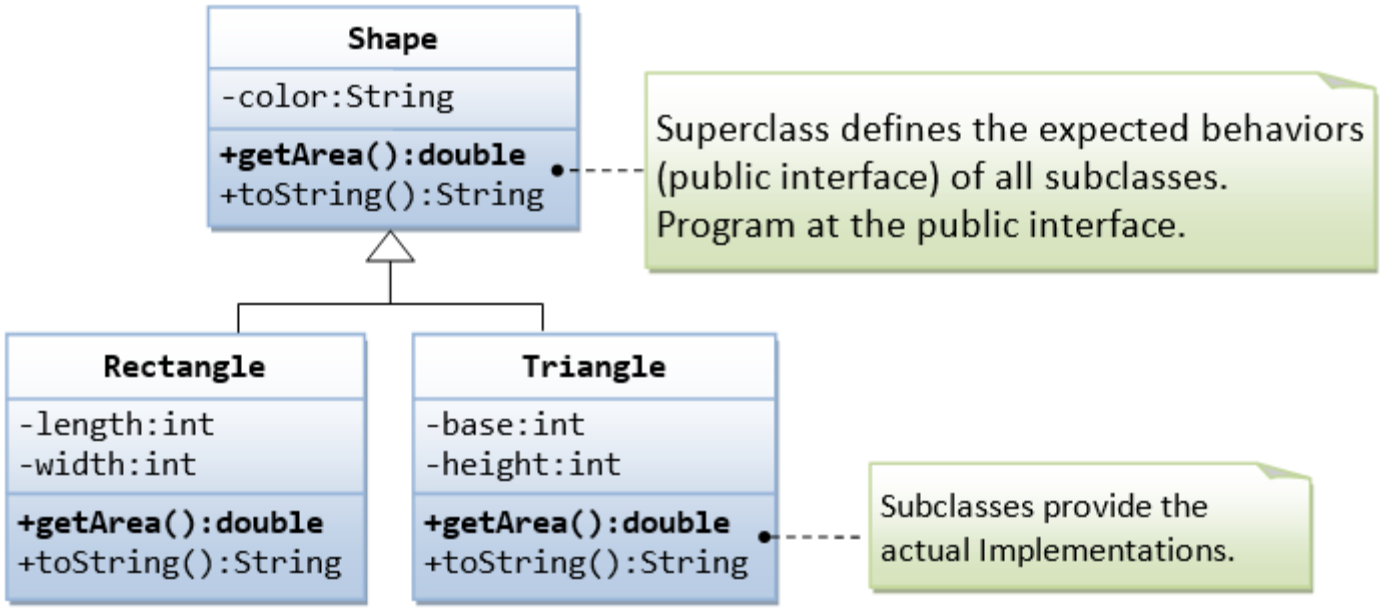
- **Car** inherits from **Vehicle** and from this class **Car** there are further derived classes **SaloonCar** and **EstateCar**.
- If a draw() method is added to the **Car** class, that is required to draw a picture of a generic vehicle. This method will not adequately draw an estate car, or other child classes.
- Overriding allows us to write a specialised draw() method for the EstateCar class - There is no need to write a new draw() method for the **SaloonCar** class as the **Car** class provides a suitable enough draw() method.
- All we have to do is write a new draw() method in the **EstateCar** class with the exact same method name.
- So, Overriding allows:
  - A more straightforward API where we can call methods the same name, even thought these methods have slightly different functionality.
  - A better level of abstraction, in that the implementation mechanics remain hidden.



The overridden draw() method.

Polymorphism example

- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes.
- For example, we would like all the shapes to have a method called getArea(), which returns the area of that particular shape.



Shape.java

```java
public class Shape {
  // Private member variable
  private String color;

  /** Constructs a Shape instance with the given color */
  public Shape(String color) {
    this.color = color;
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Shape[color = " + color + "]";
  }

  /** All shapes must provide a method called getArea() */
  public double getArea() {
    // We have a problem here!
    // We need to return some value to compile the program.
    System.err.println("Shape unknown! Cannot compute area!");
    return 0;
  }
}
```

Rectangle.java

```java
public class Rectangle extends Shape {
  // Private member variables
  private int length;
  private int width;

  // Constructs a Rectangle instance with the given color,
  // length and width
  public Rectangle(String color, int length, int width) {
    super(color);
    this.length = length;
    this.width = width;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Rectangle[length = " + length + ", width = "
           + width + "," + super.toString() + "]";
  }

  // Override the inherited getArea() to provide the proper
  // implementation for rectangle
  @Override
  public double getArea() {
    return length * width;
  }
}
```

Triangle.java

```java
public class Triangle extends Shape {
  // Private member variables
  private int base;
  private int  height;

  // Constructs a Triangle instance with the given color,
  // base and height
  public Triangle(String color, int base, int height) {
    super(color);
    this.base = base;
    this.height = height;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Triangle[base = " + base + ", height = "
           + height + ", " + super.toString() + "]";
  }

  // Override the inherited getArea() to provide the proper
  // implementation for triangle
  @Override
  public double getArea() {
    return 0.5 * base * height;
  }
}
```

TestShape.java

```
1  /**
2   * A test driver for Shape and its subclasses
3   */
4  public class TestShape {
5    public static void main(String[] args) {
6      Shape shape1 = new Shape(“yellow");
7      System.out.println(shape1); // Run Shape's toString()
8      // Shape[color = red]
9
10     Shape shape2 = new Rectangle("red", 4, 5); // Upcast
11     System.out.println(shape2); // Run Rectangle's toString()
12     // Rectangle[length = 4, width = 5, Shape[color = red]]
13     System.out.println("Area is " + shape2.getArea()); // Run Rectangle's getArea()
14     // Area is 20.0
15
16     Shape shape3 = new Triangle("blue", 4, 5); // Upcast
17     System.out.println(shape3); // Run Triangle's toString()
18     // Triangle[base = 4, height = 5, Shape[color = blue]]
19     System.out.println("Area is " + shape3.getArea()); // Run Triangle's getArea()
20     // Area is 10.0
21   }
22 }
```

Abstraction

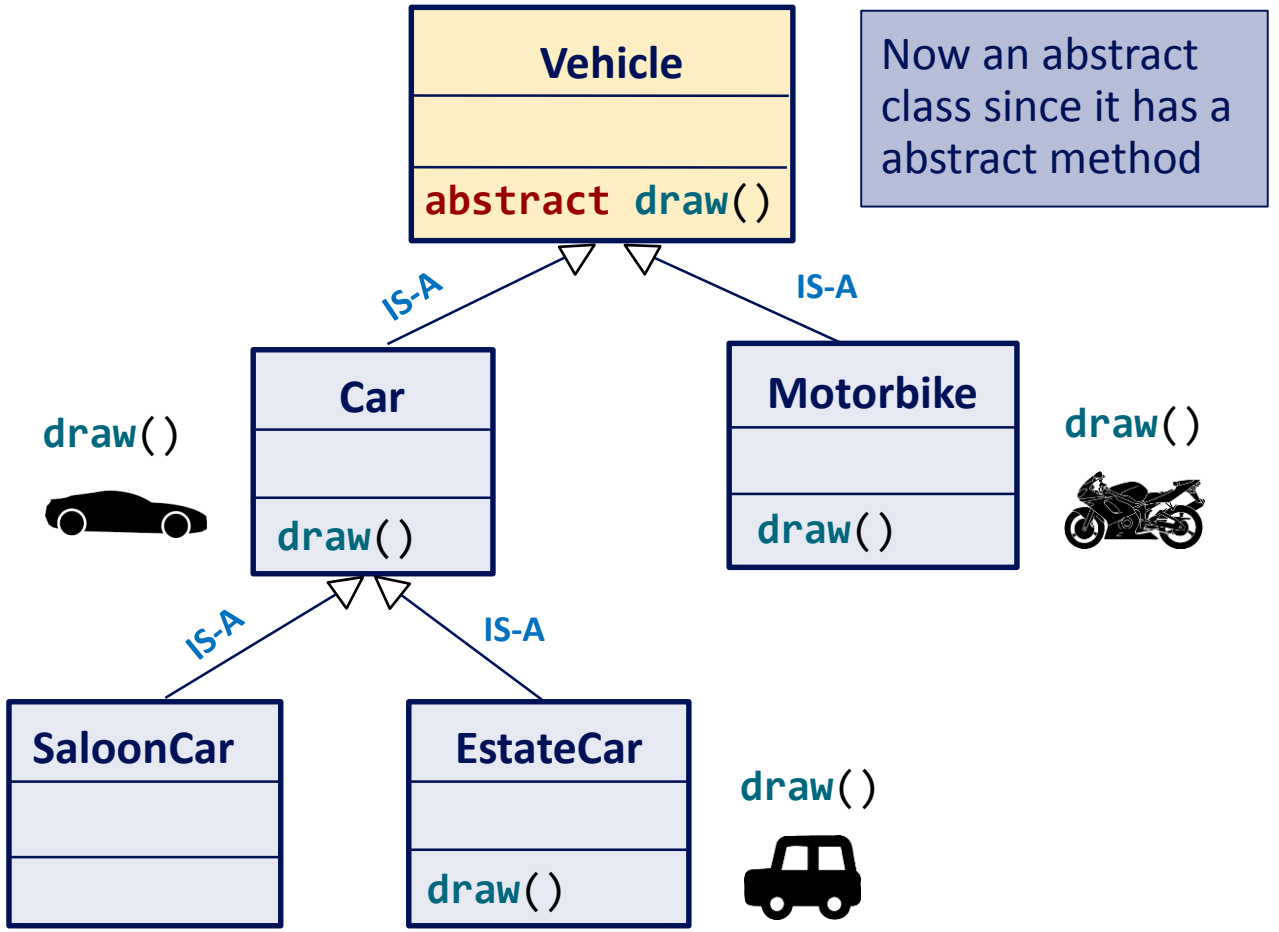
- Abstraction is the concept of object-oriented programming that shows only essential attributes and hides unnecessary information.
- The main purpose of abstraction is hiding the unnecessary details from the users.
- Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts.
- A real-life example of abstraction can be using an electric circuit board. We use the buttons to turn on and off the electrical devices through the circuit board. We know a lot of things happen inside the board. Here we are using the functionality provided and we aren't interested in the actual working of the circuit board.
- How to achieve Abstraction? Abstraction can be achieved in two different way:
  - Using abstract classes
  - Using interfaces

- An **Abstract Class** is a class that is **incomplete**, in that it describes a set of operations, but is missing the actual implementation of these operations.
- Abstract Classes:
  - Cannot be instantiated.
  - So, can only be used through inheritance.
- For example: In the Vehicle class example previously the draw() method may be defined as **abstract** as it is not really possible to draw a generic vehicle. By doing this we are forcing all derived classes to write a draw() method if they are to be instantiated.
- A class is like a set of plans from which you can create objects. In relation to this analogy, an Abstract Class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.

- The draw() has been written in all of the classes and has some functionality.

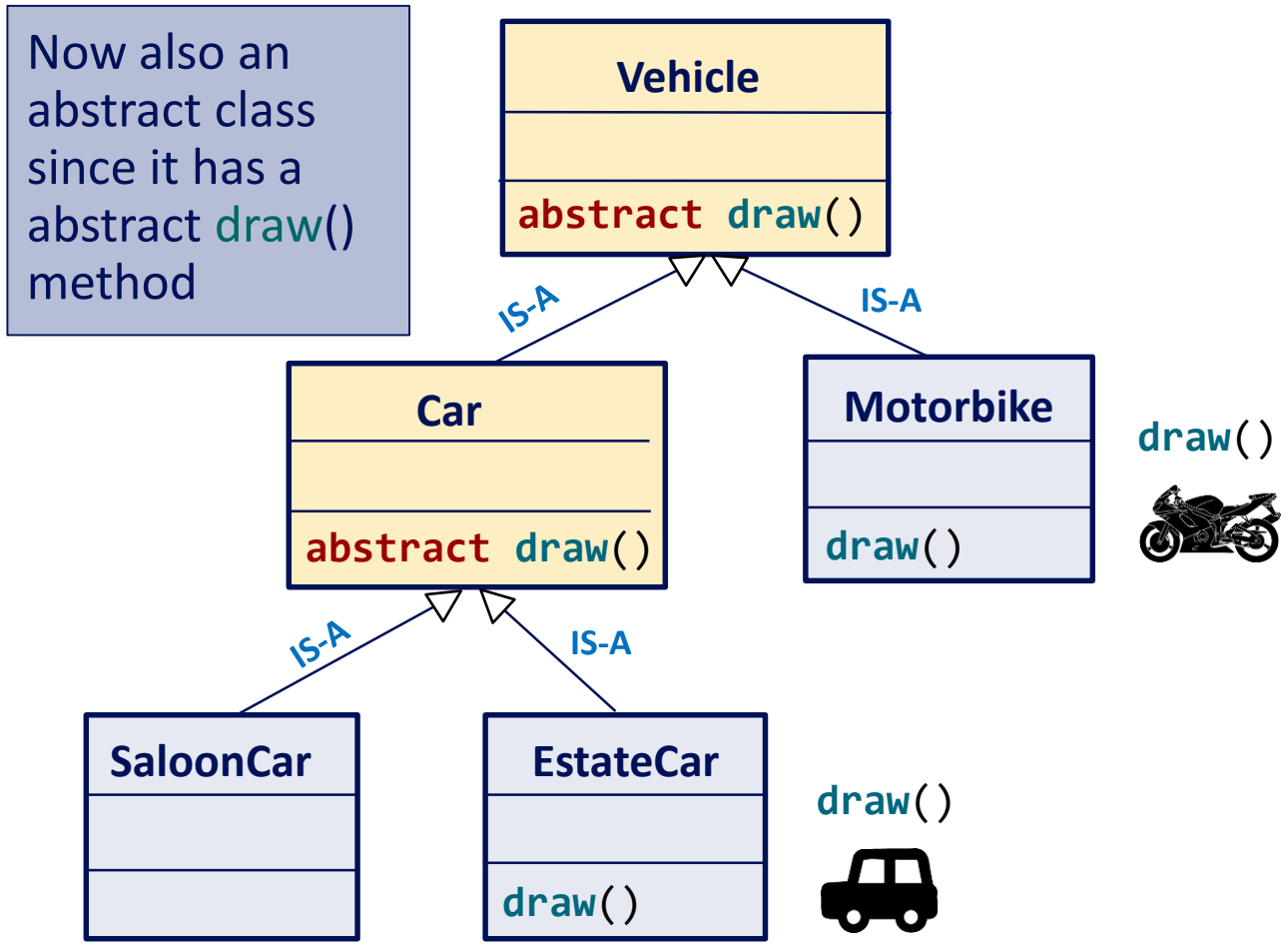
- The draw() in the **Vehicle** has been tagged as abstract and so this class cannot be instantiated - i.e. we cannot create an object of the **Vehicle** class, as it is incomplete.

- The SaloonCar has no draw() method, but it does inherit a draw() method from the parent **Car** class. Therefore, it is possible to create objects of **SaloonCar**.

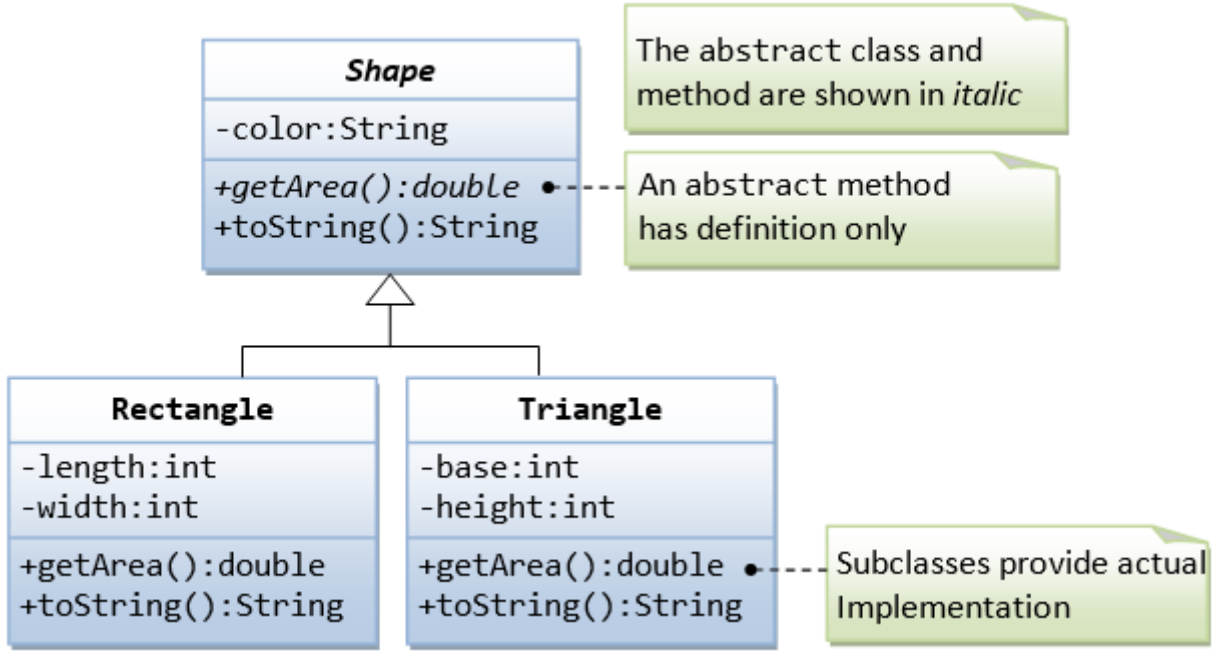


- If we required we could also tag the draw() method as **abstract** in a derived class, for example we could also have tagged the draw() as abstract in the **Car** class.

- This would mean that you could not create an object of the **Car** class and would pass on responsibility for implementing the draw() method to its children.



- An abstract Method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword **abstract** to declare an abstract method.
- A class containing one or more abstract methods is called an Abstract Class. An Abstract Class must be declared with a class-modifier abstract. An Abstract Class CANNOT be instantiated, as its definition is not complete.



Abstract Class example

Shape.java

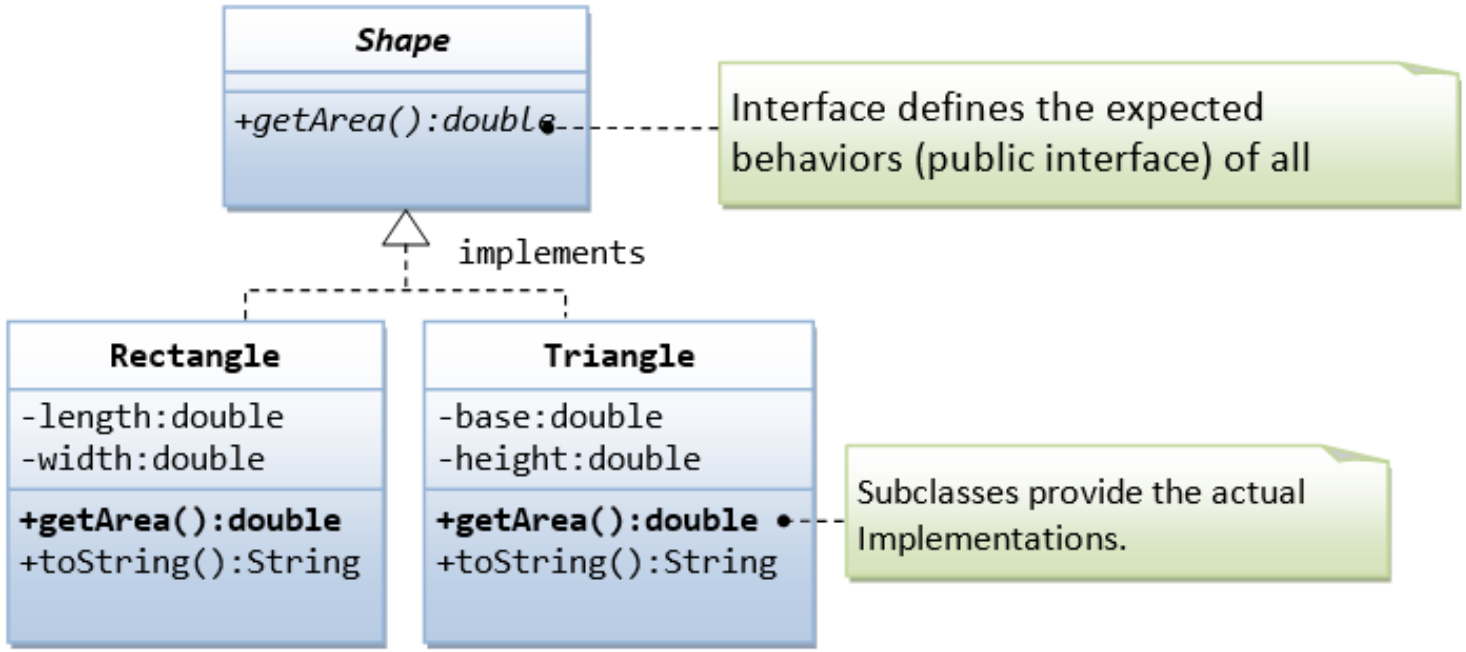
```java
/**
 * This abstract superclass Shape contains an abstract
 * method getArea(), to be implemented by its subclasses.
 */
abstract public class Shape {
  // Private member variable
  private String color;

  // Constructs a Shape instance with the given color
  public Shape(String color) {
    this.color = color;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Shape[color = " + color + "]";
  }

  // All Shape's concrete subclasses must implement a
  // method called getArea()
  abstract public double getArea();
}
```

TestShape.java

```java
public class TestShape {
  public static void main(String[] args) {
    Shape shape1 = new Rectangle("red", 4, 5);
    System.out.println(shape1);
    System.out.println("Area is " + shape1.getArea());

    Shape shape2 = new Triangle("blue", 4, 5);
    System.out.println(shape2);
    System.out.println("Area is " + shape2.getArea());

    // Cannot create instance of an abstract class
    Shape shape3 = new Shape("green");
    // compilation error: Shape is abstract;
    //cannot be instantiated
  }
}
```

- A **Interface** is a 100% Abstract Superclass which define a set of methods its subclasses must support.
- An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables).
- We have to use the keyword "**interface**" to define an interface (instead of keyword "class" for normal classes). The keyword public and abstract are not needed for its abstract methods as they are mandatory.
- Similar to an abstract superclass, an interface cannot be instantiated. You have to create a "subclass" that implements an interface, and provide the actual implementation of all the abstract methods.
- Unlike a normal class, where you use the keyword "extends" to derive a subclass. For interface, we use the keyword "**implements**" to derive a subclass.
- An interface is a contract for what the classes can do. It, however, does not specify how the classes should do it.
- An interface provides a form, a protocol, a standard, a contract, a specification, a set of rules, an interface, for all objects that implement it. It is a specification and rules that any object implementing it agrees to follow.

- We re-write the abstract superclass Shape into an interface, containing only abstract methods.



Shape.java

```java
/**
 * The interface Shape specifies the behaviors of this implementations
 * subclasses.
 */
public interface Shape { // Use keyword "interface" instead of "class"
  // List of public abstract methods to be implemented by its subclasses
  // All methods in interface are "public abstract".
  // "protected", "private" and "package" methods are NOT allowed.
  double getArea();
}
```

Rectangle.java

```java
/**
 * The subclass Rectangle needs to implement all the abstract methods
 * in Shape
 */
public class Rectangle implements Shape { // using keyword "implements"
instead of "extends"
  // Private member variables
  private int length;
  private int width;

  // Constructs a Rectangle instance with the given length and width
  public Rectangle(int length, int width) {
    this.length = length;
    this.width = width;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Rectangle[length = " + length + ", width = " + width + "]";
  }

  // Need to implement all the abstract methods defined in the interface
  // Returns the area of this rectangle
  @Override
  public double getArea() {
    return length * width;
  }
}
```

Triangle.java

```java
/**
 * The subclass Triangle need to implement all the abstract methods
 * in Shape
 */
public class Triangle implements Shape {
  // Private member variables
  private int base;
  private int height;

  // Constructs a Triangle instance with the given base and height
  public Triangle(int base, int height) {
    this.base = base;
    this.height = height;
  }

  // Returns a self-descriptive string
  @Override
  public String toString() {
    return "Triangle[base = " + base + ", height = " + height + "]";
  }

  // Need to implement all the abstract methods defined in the interface
  // Returns the area of this triangle
  @Override
  public double getArea() {
    return 0.5 * base * height;
  }
}
```

TestShape.java

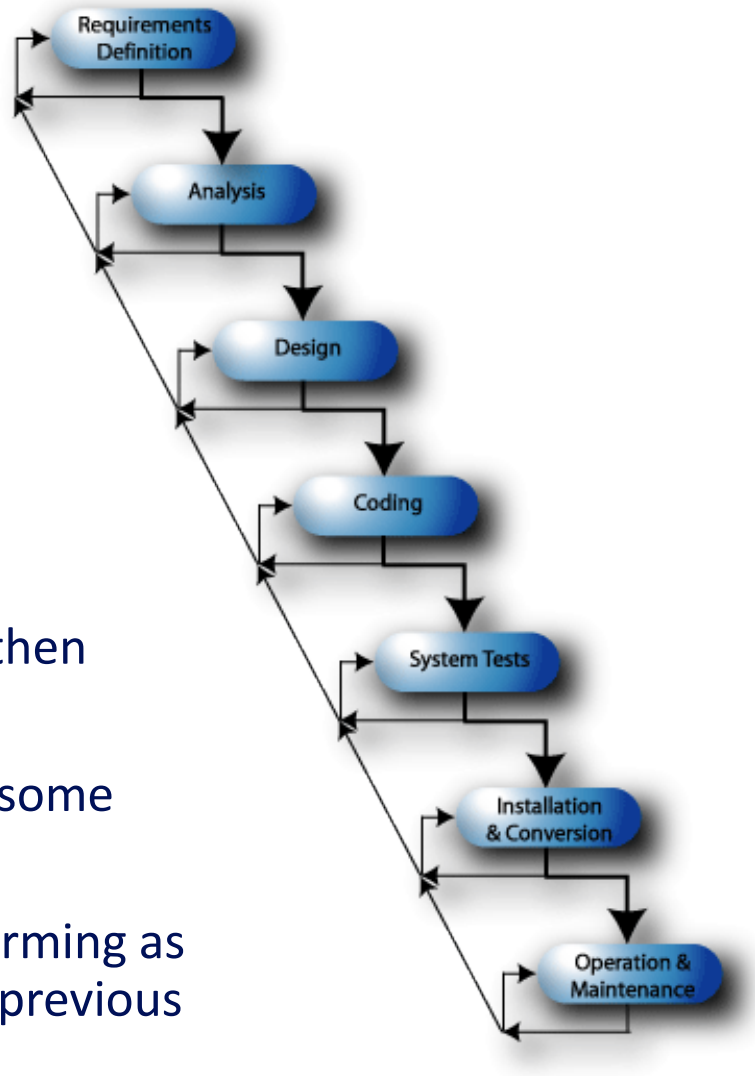
```
1  public class TestShape {
2    public static void main(String[] args) {
3      Shape shape1 = new Rectangle(1, 2); // upcast
4      System.out.println(shape1);
5      // Rectangle[length = 1, width = 2]
6      System.out.println("Area is " + shape1.getArea());
7      // Area is 2.0
8
9      Shape shape2 = new Triangle(3, 4); // upcast
10     System.out.println(shape2);
11     // Triangle[base = 3, height = 4]
12     System.out.println("Area is " + shape2.getArea());
13     // Area is 6.0
14
15     // Cannot create instance of an interface
16     Shape shape3 = new Shape("green");
17     // compilation error: Shape is abstract; cannot be instantiated
18   }
19 }
```

- Abstract Class and Interface are used to separate the public interface of a class from its implementation so as to allow the programmer to program at the interface instead of the various implementation.
- When we talk about **Abstract Classes** we are defining the characteristics of an object type and specify **what an object is**.
- When we talk about an **Interface**, we define the capabilities that we promise to provide. We are talking about establishing a contract about **what an object can do**.

Object-Oriented Analysis and Design

- Object-oriented programming has been around since the 1990s. Formal design processes when using objects involves many complex stages and are the debate of much research and development.
- **Why use the object-oriented approach?** Consider the general cycle that a programmer goes through to solve a programming problem:
  - **Formulate the problem** - The programmer must completely understand the problem.
  - **Analyse the problem** - The programmer must find the important concepts of the problem.
  - **Design** - The programmer must design a solution based on the analysis.
  - **Code** - Finally the programmer writes the code to implement the design.

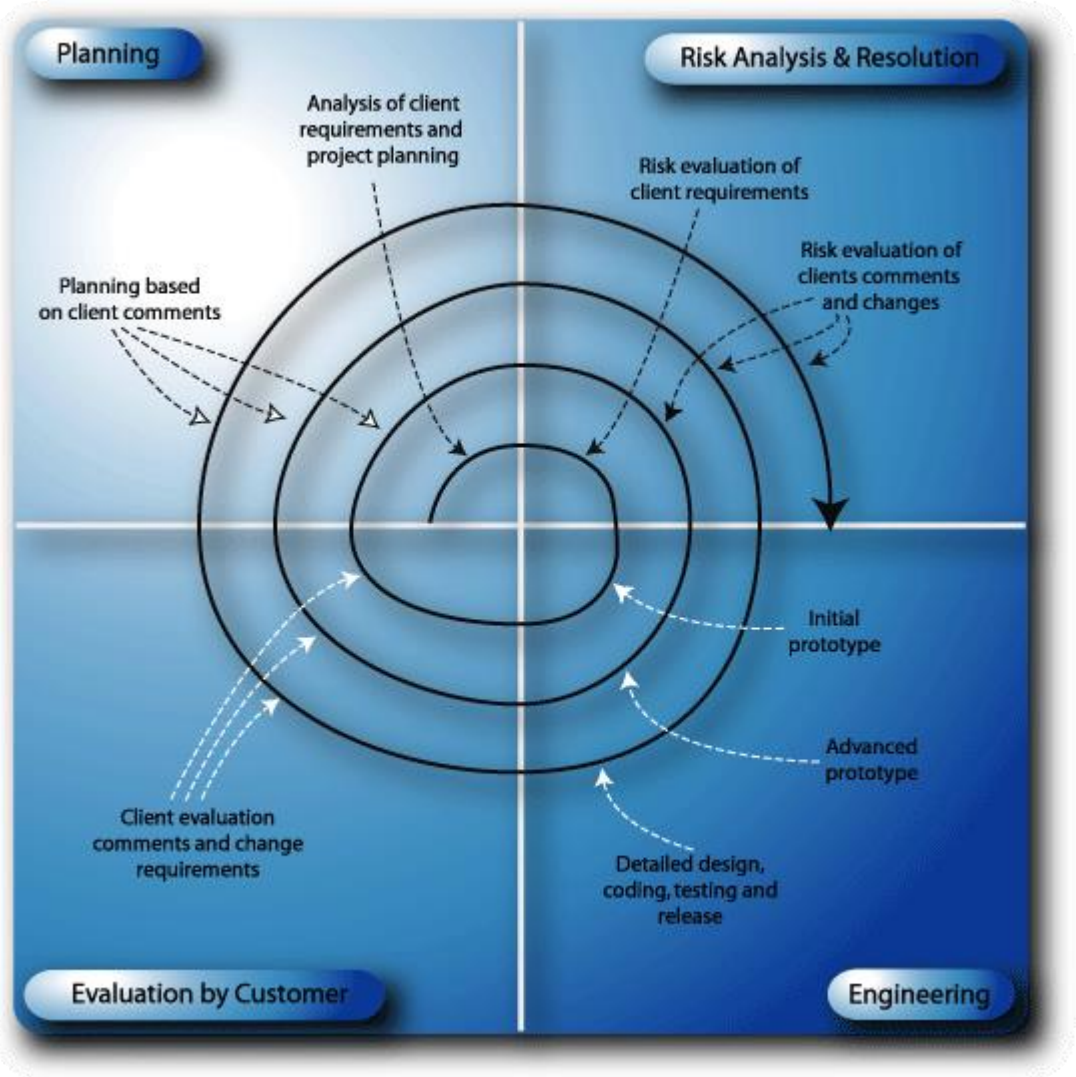
- The Waterfall Model is a linear sequential model that begins with definition and ends with system operation and maintenance. It is the most common software development life cycle model and is particularly useful when specifying overview project plans, as it fits neatly into a Gantt chart format.
- The Waterfall Model is a general model, where in small projects some of the phases can be dropped.
- In large scale software development projects, some of these phases may be split into further phases.
- At the end of each phase the outcome is evaluated and if it is approved then development can progress to the next phase.
- If the evaluation is rejected then the last phase must be revisited and in some cases earlier phases may need to be examined.
- In the Figure, the thicker line shows the likely path if all phases are performing as planned. The thinner lines show a retrace of steps to the same phase or previous phases.



- **Requirements Definition**: The customer must define the requirements to allow the developer to understand what is required of the software system. If this development is part of a larger system then other development teams must communicate to develop system interfaces.
- **Analysis**: The requirements must be analysed to form the initial software system model.
- **Design**: The design stage involves the detailed definition of inputs, outputs and processing required of the components of the software system model.
- **Coding**: The design is now coded, requiring quality assurance of inspection, unit testing and integration testing.
- **System Tests**: Once the coding phase is complete, system tests are performed to locate as many software errors as possible. This is carried out by developer before the software is passed to the client. The client may carry out further tests, or carry out joint tests with the developer.
- **Installation and Conversion**: The software system is installed. As part of a larger system, it may be an upgrade; in which case, further testing may be required to ensure that the conversion to the upgrade does not effect the regular corporate activity.
- **Operation and Maintenance**: Software operation begins once it is installed on the client site. Maintenance will be required over the life of the software system once it is installed. This maintenance could be repair, to fix a fault identified by the client, adaptive to use the current system features to fulfill new requirements, or perfective to add new features to improve performance and/or functionality.

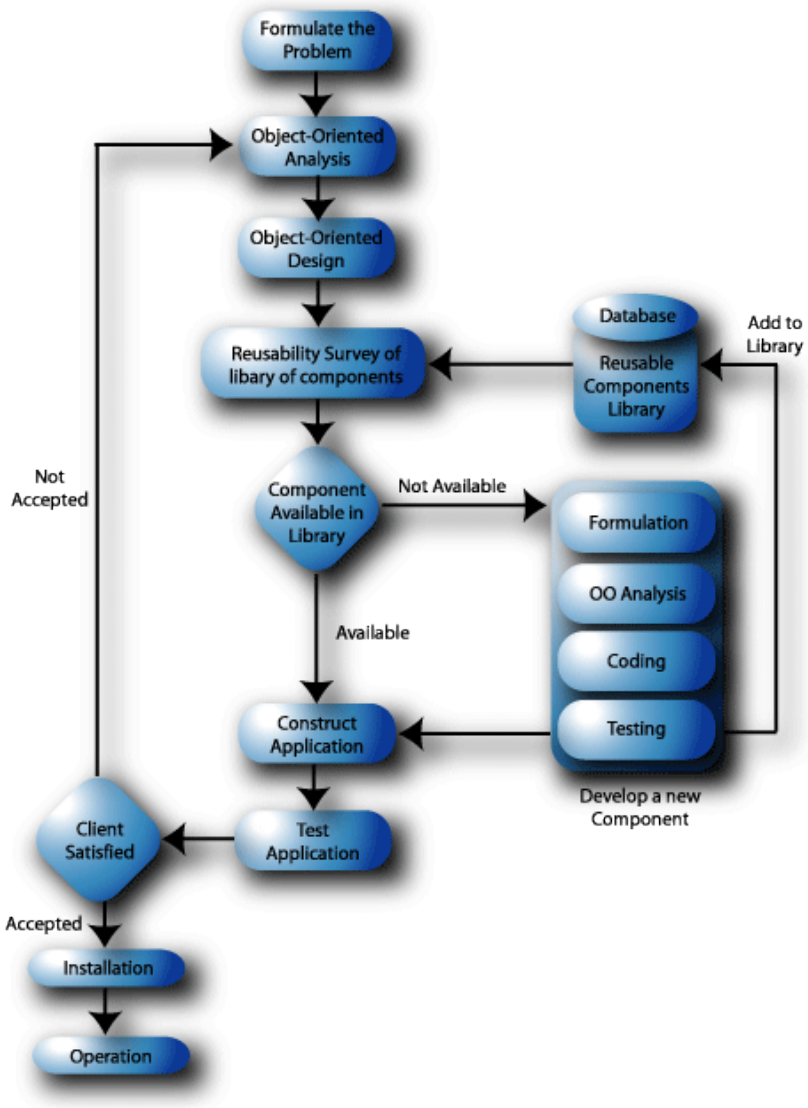
The Spiral Model

- The Spiral Model was suggested by Boehm (1988) as a methodology for overseeing large scale software development projects that show high prospects for failure. It is an iterative model that builds in risk analysis and formal client participation into prototype development.
- The spiral, as shown in figure of development is iterative, with each iteration involving planning, risk analysis, engineering (from design, to coding, testing, installation and then release) and customer evaluation (including comments, changes and further requirements). More advanced forms of this model are available for dealing with further communication with the client.
- The spiral model is particularly suited to large scale software development projects and needs constant review. For smaller projects an agile development model is more suitable.



The Object-Oriented Design Model

- One object-oriented methodology is based around the re-use of development modules and components. As such, a new development model is required that takes this re-use into account.
- The object-oriented model as shown in the figure builds integration of existing software modules into the system development. A database of reusable components supplies the components for re-use. The object-oriented model starts with the formulation and analysis of the problem. The design phase is followed by a survey of the component library to see if any of the components can be re-used in the system development. If the component is not available in the library then a new component must be developed, involving formulation, analysis, coding and testing of the module. The new component is added to the library and used to construct the new application.
- This model aims to reduce costs by integrating existing modules into development. These modules are usually of a higher quality as they have been tested in the field by other clients and should have been debugged. The development time using this model should be lower as there is less code to write.
- The object-oriented model should provide advantages over the other models, especially as the library of components that is developed grows over time.



- **Task**: If we were given the problem; “Write a program to implement a simple savings account”… The account should allow deposits, withdrawals, interest and fees.
- **Solution**:
  - The problem produces many concepts, such as bank account, deposit, withdrawal, balance etc.. that are important to understand. An OO language allows the programmer to bring these concepts right through to the coding step.
  - The savings account may be built with the properties of an account number and balance and with the methods of deposit and withdrawal, in keeping with the concept of the bank account.
  - This allows an almost direct mapping between the design and the coding stages, allowing code that is easy to read and understand (reducing maintenance and development costs).

- So after discussion with the client, the following formulation could be achieved - Design a banking system that contains both teller and ATM interaction with the rules:
  - The cashiers and ATMs dispense cash.
  - The network is shared by several banks.
  - Each transaction involves an account and documentation.
  - There are different types of bank accounts.
  - There are different kinds of transactions.
  - All banks use the same currency.
  - Foreign currency transactions are permitted.
  - ATMs and tellers require a cash card.

Step 1. Identify Possible Classes

- ATM, cashier, cashier station, software, customer, cash.
- Banking network, bank.
- Transaction, transaction record.
- Account, deposit account, long term savings account, current account.
- Withdrawal, lodgement, cheque.
- Currency.
- Foreign currency, euro cheque.
- Cash card, computer system.

Step 2. Remove Vague Classes

- Software, computer system, cash.

- **Step 3. Add New classes that arise!**
  - Currency converter
- **Step 4. Create Associations**
  - Banking Network (includes cashiers and ATMs)
  - Banks (holds accounts)
  - Account (has a balance, a currency, a log of transactions)
  - Transaction (requires a cash card)
  - Lodgement (has an account number, an amount)
  - Withdrawal (has an account number, an amount)
  - Cheque (is a withdrawal, has a payee, an amount)
  - Eurocheque (is a cheque, has a currency)
  - ATMs (accept cashcards, dispense cash)

Step 5. Refine the Classes

Bank:
- has a name
- has accounts
- has a base currency
- has a sort code

Account:
- has an owner
- has a balance
- has an account number
- has a log of transactions

Deposit Account:
- is an account
- has a shared interest rate

Current Account:
- is an account
- has an overdraft limit

Transaction:
- has an account
- has a date
- has a value
- has a bank
- has an account Number
- has a number

Withdrawal:
- is a transaction

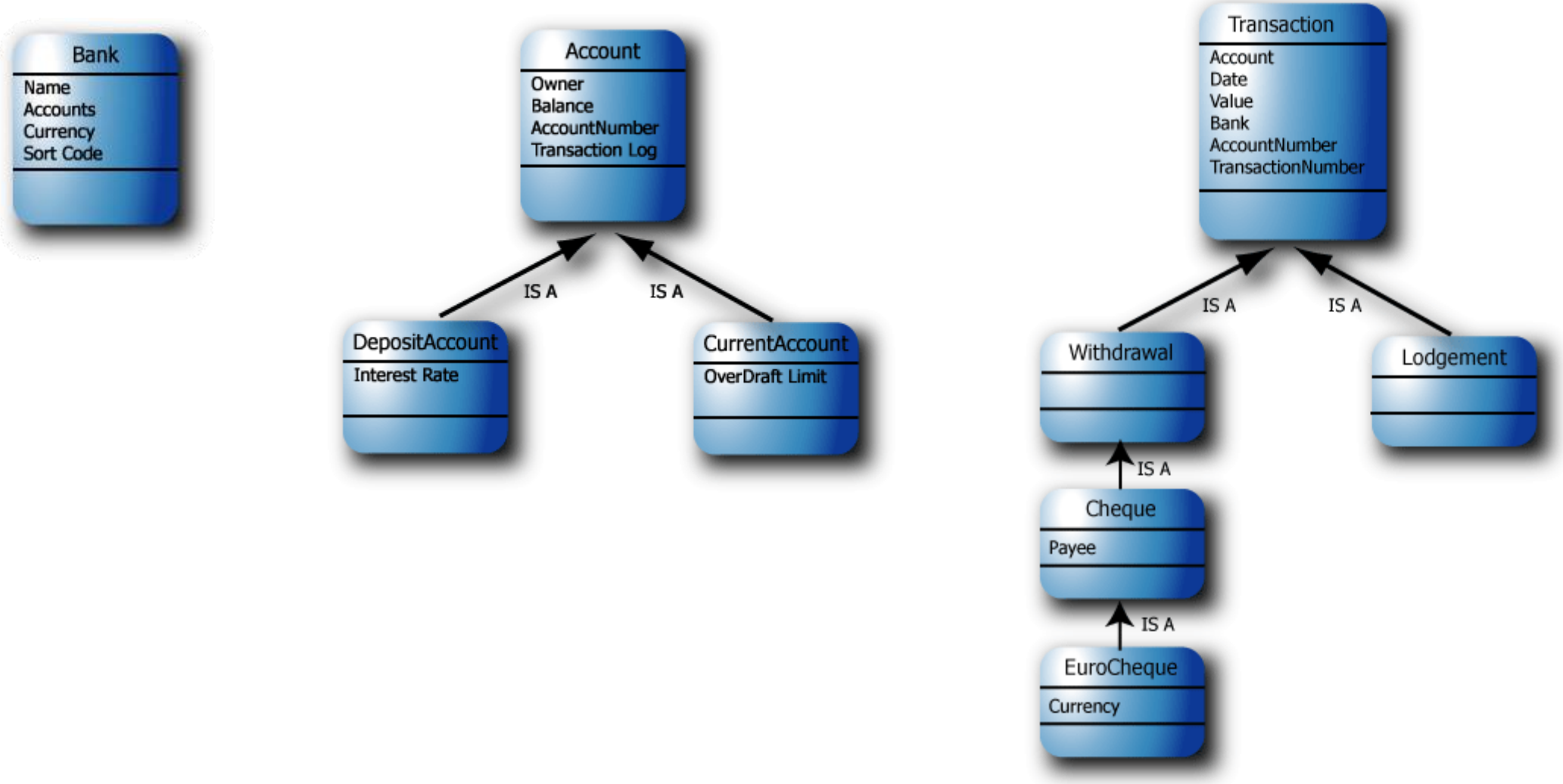
Lodgement:
- is a transaction

Cheque:
- is a withdrawal
- has a payee

EuroCheque:
- is a cheque
- has a currency

CurrencyConverter:

- **Step 6. Visual Representation of the Classes**



Thank you!