CSC8012 Software Development Techniques and Tools — Part 1

Dr Konrad Dabrowski
konrad.dabrowski@newcastle.ac.uk

School of Computing
Newcastle University

Link to AVD:
https://rdweb.wvd.microsoft.com/arm/webclient/index.html

Assessment

- All module information including assessment can be found at Canvas: `https://canvas.ncl.ac.uk`
- Programming Coursework: (40%)
  - One piece of assessed programming coursework
    - Coursework should be submitted using NESS:
    - `https://ness.ncl.ac.uk`
- One PC-based exam (60%)

Practicals

- Week 8 Practicals (work on formative coursework):
  - 11:30-13:30 Monday USB.3.015 (B)
  - 14:30-16:30 Tuesday USB.3.015 (B)
  - 09:30-11:30 Wednesday USB.3.015 (B)
- Week 9 Practicals (work on assessed coursework):
  - 11:30-13:30 Monday USB.3.015 (B)
  - 14:30-16:30 Tuesday USB.3.015 (B)
  - 09:30-11:30 Wednesday USB.3.015 (B)
- Week 10: No practicals (work on assessed coursework!)
- Week 11 Practicals (work on assessed coursework):
  - 11:30-13:30 Monday USB.3.015 (B)
  - 14:30-16:30 Tuesday USB.3.015 (B)
  - 09:30-11:30 Wednesday USB.3.015 (B)
  - **12:30-14:30 Wednesday USB.3.015 (B)**

Lectures

- Week 8 Lectures:
  - 09:30-11:30 Monday USB.1.006
  - 09:30-11:30 Tuesday USB.1.006
  - 10:30-12:30 Thursday USB.1.006
  - **09:30-11:30 Friday NUBS.1.03**
- Week 9 Lectures:
  - 09:30-11:30 Monday USB.1.006
  - 09:30-11:30 Tuesday USB.1.006
  - 10:30-12:30 Thursday USB.1.006
- Week 10: No lectures (work on your coursework!)
- Week 11 Lectures:
  - 09:30-11:30 Monday USB.1.006
  - 09:30-11:30 Tuesday USB.1.006
  - **09:30-11:30 Thursday FDC.G.56**

Office Hour and CSC8012 Team

- I will hold an office hour Wednesdays 11:35-12:25 on Zoom: https://newcastleuniversity.zoom.us/j/85157196413?pwd=czQySnVEcUdMNkxkM0l6TEpnYWV0QT09
  - Meeting ID: 851 5719 6413
  - Passcode: 302082
  - If you have a specific question, please email me in advance if possible: konrad.dabrowski@newcastle.ac.uk
- You should join the CSC8012 team by following the instructions below:
  1. Go to https://teams.microsoft.com/_#/discover and if prompted sign in with your University login in this format c1234567@newcastle.ac.uk
  2. In the Join or create a team section select "Join a team with a code"
  3. Enter the following code: 3jofil7

Recommended Books

1. *Objects First with Java: A practical Introduction Using BlueJ*, 5th Edition, by David Barnes and Michael Kölling, Pearson Education International, 2012
   - Good introduction to object-oriented programming in Java and the BlueJ programming environment
2. *Big Java*, 4th Edition, by Cay Horstmann, Wiley, 2010
   - Comprehensive book for Java programming covering almost every aspect of Java
3. *Java Programming: From Problem Analysis to Program Design*, 4th Edition, by D.S. Malik, Course Technology, 2010
   - Good introduction to Java programming providing many well explained examples
4. *Java Programming: Program Design Including Data Structures*, by D.S. Malik, Thomson Course Technology, 2006
   - An extended version of book (3) containing chapters about data structures
5. *The Java Tutorials*
   `https://docs.oracle.com/javase/tutorial/`

Goals

- Extend the knowledge of Java gained in the Introduction to Software Development module (CSC8011)
- Provide a grounding in object-oriented design and implementation in the context of the Java programming language
- Introduce the concepts of inheritance, exception handling and a selection of algorithms
- Final aim: to be able to implement a small software system in Java

Module topics

- Inheritance and polymorphism
- Exceptions and exception handling
- Generic classes and methods
- Searching and sorting algorithms

Integrated Development Environments for Java

- There are many integrated development environments (IDEs) for Java programming: IntelliJ, BlueJ, Eclipse, NetBeans etc.
- They all allow you to:
  - create/edit Java code
  - compile
  - try the code out
- You will be allowed to develop your programs by using IntelliJ, BlueJ, Eclipse or an IDE suggested for the CSC8011 module.

Let's Start. . .

- Next, we will introduce you to the concept of *inheritance*.
- Our first aim is to learn how to improve the structure of Java applications using this concept.

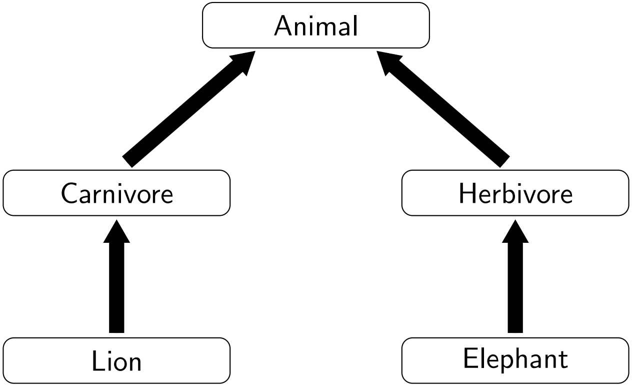
Inheritance and Polymorphism: Topics

- Inheritance
- The `Object` class
- Abstract classes
- Interfaces
- Polymorphism

Inheritance

- *Inheritance* is a mechanism that allows us to define one class as an extension of another.
- It allows us to create two classes that are similar without the need to write the identical part twice.
- A new class derived from an existing class is called a *subclass* (or *child* class) of the existing class.
- The original class from which the other class is derived is called a *superclass* (or *parent* class).
- The subclass inherits the properties of the superclass and can have more fields and methods than its parent. This is why we say that the subclass extends the superclass.
- In Java a class can extend the definition of only one class.
- However, more than one class can inherit from the same superclass.
- Inheritance is transitive: that means that if `a` is a child class of `b`, and `b` is a child class of `c`, then `c` is related by inheritance to `a`.
- Classes can form an inheritance hierarchy.

Inheritance Hierarchy



The Animal Class

```
public class Animal
{
    private String name;

    public Animal()
    {
        name = "";
    }

    public Animal(String n)
    {
        name = n;
    }

   //methods
}
```

The Animal Class – Methods

```
public void setInfo(String n)
{
    name = n;
}

public void printInfo()
{
    System.out.print(name + " ");
}

public String getName()
{
    return name;
}
```

Deriving a Class from an Existing Class

- General syntax:

```
public class ClassName extends ExistingClassName
{
    // class members
}
```

- Suppose we have defined a class called `Animal`. The class `Herbivore` is derived from the `Animal` class, and the class `Elephant` is, in turn, derived from the `Herbivore` class.

```
public class Herbivore extends Animal
{
    ...
}
public class Elephant extends Herbivore
{
    ...
}
```

Superclasses and Subclasses

- The `private` members of the superclass are `private` to the superclass. Hence, the methods of the subclass cannot directly access them.
- The subclass can directly access the `public` members of the superclass.
- The subclass can have additional fields and/or methods.
- The subclass can *overload* the `public` methods of the superclass. That means, in the subclass, you can have a method with the same name as a method in the superclass, but different number and/or types of parameters.
- The subclass can *override* (redefine) the `public` methods of the superclass. That means, in the subclass, you can have a method with the same name, number and types of parameters as a method in the superclass. However, this redefinition applies only to the objects of the subclass, and not to the objects of the superclass.
- All fields of the superclass are also fields of the subclass.
- The methods of the superclass (unless overridden) are also methods of the subclass.

Constructors of the Superclass and Subclass

- When an object is created, the constructor of that object makes sure that all object fields are initialised to some reasonable state.
- When a subclass object is created, it inherits the fields of the superclass, but the subclass object cannot directly access the `private` fields of the superclass. The constructors of the subclass can directly initialise only the fields of the subclass. To initialise the `private` inherited fields we need to call one of the constructors of the superclass.
- To call a constructor of the superclass we use the keyword `super`
  - For a constructor without parameters

    ```
    super();
    ```
  - For a constructor with parameters

    ```
    super(actual parameter list);
    ```
- A call to the constructor of the superclass must be the first statement in the body of a constructor of the subclass.

The Herbivore Class

```
public class Herbivore extends Animal
{
    private int grassNeeded;

    public Herbivore()
    {
        super();
        grassNeeded = 0;
    }

    public Herbivore(String n, int g)
    {
        super(n);
        grassNeeded = g;
    }
    // methods
}
```

The Herbivore Class — Methods

```
public void setInfo(String n, int g)
{
     // example of overloading
     setInfo(n);
     grassNeeded = g;
}

public void printInfo()
{
     // example of overriding
     super.printInfo();
     System.out.print(grassNeeded + " ");
}

public int getGrassWeight()
{
     return grassNeeded;
}
```

The Elephant Class

```
public class Elephant extends Herbivore
{
    private double tuskLength;

    public Elephant(String n, int w, double l)
    {
        super(n, w);
        tuskLength = l;
    }

    public void printCharacteristics()
    {
        System.out.print(getName() + " ");
        System.out.print(getGrassWeight() + " ");
        System.out.println(tuskLength + " ");
    }

    public void printInfo()
    {
        super.printInfo();
        System.out.println(tuskLength + " ");
    }
}
```

Testing the Hierarchy

```
public class TestAnimal
{
    public static void main(String[] args)
    {
        Animal a = new Animal("Lion");
        System.out.println(a.getName());
        Herbivore h = new Herbivore("Rhino", 200);
        System.out.println(h.getName());
        System.out.println(h.getGrassWeight());
        Elephant dumbo = new Elephant("Elephant", 350, 1.2);
        System.out.println(dumbo.getName());
        System.out.println(dumbo.getGrassWeight());
        dumbo.printCharacteristics();
        dumbo.printInfo();
    }
}
```

protected Members of a Class

- The private members of the superclass are private to the superclass. The methods of the subclass cannot directly access them.
- In our last example, we had the following field declarations:

```
private String name; //in the Animal class
private int grassNeeded; //in the Herbivore class
```

- The above private fields can be accessed in the Elephant class using public methods of the Animal and Herbivore classes.

```
public void printCharacteristics()
{
    System.out.print(getName() + " ");
    System.out.print(getGrassWeight() + " ");
    System.out.println(tuskLength + " ");
}
```

- The superclass can give direct access to its private members to the subclasses by declaring them protected instead.
- protected members of the superclass can be accessed directly in the subclasses. They can be also accessed directly in other classes in the same package.

Using protected Members

- Suppose we change declarations of the fields as follows:

```
protected String name; //in the Animal class
protected int grassNeeded; //in the Herbivore class
```

- We can then write the `printCharacteristics` method in the `Elephant` class as follows:

```
public void printCharacteristics()
{
    System.out.print(name + " ");
    System.out.print(grassNeeded + " ");
    System.out.println(tuskLength + " ");
}
```

- We can also write the following statements in the `TestAnimal` class:

```
Elephant dumbo = new Elephant("Elephant", 350, 1.2);
System.out.println(dumbo.name);
System.out.println(dumbo.grassNeeded);
```

Working Within a Hierarchy

- Any object of a subclass can be assigned to a variable of the superclass's type.
- However, the opposite is not generally true. It is only allowed sometimes with an appropriate cast.
- Example:

```
Animal a, aRef, animalWithLongerName;
Herbivore h, hRef;
a = new Animal("Lion");
h = new Herbivore("Rhino",200);
aRef = h;
// hRef = a; //compile-time error
// hRef = (Herbivore) a; // run-time error
                         // (ClassCastException)
hRef = (Herbivore) aRef;
```

- Another example:

```
if (a.getName().length() > h.getName().length())
    animalWithLongerName = a;
else
    animalWithLongerName = h;
```

Dynamic Binding

- When there are several versions of a method in a hierarchy (because it was overridden), the one in the closest subclass to an object is always used.
- So, what happens when an object of the subclass is assigned to a variable of the superclass's type?

```
Animal a, aRef;
Herbivore h;
a = new Animal("Lion");
a.printInfo(); // calls Animal version of printInfo
h = new Herbivore("Rhino",200);
h.printInfo(); // calls Herbivore version of printInfo
aRef = h;
aRef.printInfo(); // calls Herbivore version of printInfo
```

- Even though `aRef` was declared as a variable of the type `Animal`, when the program executes, in the last statement the `printInfo` method of the `Herbivore` class is called. This is called *dynamic binding* — the method to call is determined at execution time rather than at compile time.

The `instanceof` Operator

- Java provides the `instanceof` operator to determine the type of an object pointed to by a reference variable. The `boolean` expression below evaluates to true if `refVar` points to an object of the `ClassName` class; otherwise it evaluates to `false`:
  `refVar instanceof ClassName`
- If `refVar` is `null`, `refVar instanceof ClassName` evaluates to `false` (`null` is not an instance of any class)
- Example:

```
Animal a, aRef;
Herbivore h;
a = new Animal("Lion");
h = new Herbivore("Rhino",200);
aRef = h;
if (a instanceof Herbivore)
    System.out.println("a is an instance of Herbivore");
else
    System.out.println("a is not an instance of Herbivore");
if (aRef instanceof Herbivore)
    System.out.println("aRef is an instance of Herbivore");
else
    System.out.println("aRef is not an instance of Herbivore");
```

- Output:

```
a is not an instance of Herbivore
aRef is an instance of Herbivore
```

The `Object` Class

- The `Object` class is in the `java.lang` package.
- It is the root of the class hierarchy in Java
- Every class in Java has the `Object` class as a superclass (directly or indirectly). If a class does not extend any existing class, then it is considered to be derived from the `Object` class. Hence, for example, the heading of the `Animal` class definition:

  ```
  public class Animal
  ```

  could have been written:

  ```
  public class Animal extends Object
  ```

- The `Object` class provides some useful methods, for example:

  ```
  public boolean equals(Object obj)
  public String toString()
  ```

- By the rules of the inheritance mechanism, every public method of the class `Object` (unless it is overridden) can be invoked on every object of any class type.

The equals Method

- The `equals` method of the `Object` class checks whether the object specified by the parameter is “equal to” `this` object.
- It returns `true` if `this` object and the object specified by the parameter, `obj`, refer to the same memory space. Consider:

  ```
  Animal a1 = new Animal("Lion");
  Animal a2 = new Animal("Lion");
  ```
- Logically, `a1` and `a2` are the same animals, but the two reference variables do not refer to the same object and therefore: `a1.equals(a2)` returns `false`
- If logical equivalence is important, a class should override `equals`.

Overriding equals in the Animal Class

```
public boolean equals(Object obj)
{
    if (this == obj) return true;
    if (!(obj instanceof Animal)) return false;
    Animal a = (Animal) obj;
    if (name == null)
        return a.name == null;
    else
        return name.equals(a.name);
}
```

- Now, the test for equality, `a1.equals(a2)`, returns `true`.

The toString Method

- The `toString` method returns a string representation of the object.
- The `toString` method defined in the `Object` class returns a string consisting of a name of the class of which the object is an instance, followed by the `@` character and the hash code of the object.
- The `print` and `println` methods output the string created by the method `toString`. For example, after the following statements:

  ```
  Animal a = new Animal("Elephant");
  System.out.println(a);
  ```

  the output will be:

  ```
  Animal@65ab7765
  ```
- The `toString` method was designed to be overridden in subclasses.
- If we want the last statement to print `Elephant` instead, we need to override the `toString` method in the `Animal` class as follows:

  ```
      public String toString()
      {
          return name;
      }
  ```
- After including the above method in the `Animal` class, the `printInfo` method becomes redundant and can be removed.

Sports Club Example

- We have been asked to write a program to help a secretary of a sports club in their daily work.
- The program should:
    - keep information about all the members (name, surname, age, information whether the fee was paid);
    - be able to list information about all members;
    - be able to register a new member;
    - be able to accept payments and print receipts and
    - be able to print reminders for all members who are overdue with fee payment.
- We assume that there is a limit on the number of members the club can accept.
- We assume that no two members have the same first name and surname.

The Person Class

```
import java.io.*;
public class Person
{
    private String firstName;
    private String surname;

    public Person()
    {
        firstName = "";
        surname = "";
    }

    public Person(String name1, String name2)
    {
        firstName = name1;
        surname = name2;
    }

    //methods
}
```

The Person Class — Methods

```
public void setName(String name1, String name2)
{
    firstName = name1;
    surname = name2;
}

public String getFirstName()
{
    return firstName;
}

public String getSurname()
{
    return surname;
}

public String toString()
{
    return firstName + " " + surname;
}
```

The Person Class — More Methods

```
public void printName(PrintWriter f)
{
     f.println(firstName + " " + surname);
}

public char initial()
{
     return firstName.charAt(0);
}

public void printShortName(PrintWriter f)
{
     f.print(initial() + ". " + surname);
}

public boolean equals(Person otherPerson)
{
     return (firstName.equals(otherPerson.firstName)
            && surname.equals(otherPerson.surname));
}
```

The ClubMember Class

```
import java.io.*;
public class ClubMember extends Person
{
    private static final int SENIORAGE = 65;
    private static final double NORMALFEE = 10;
    private static final double SENIORFEE = 5;
    private int personAge;
    private boolean feePaid;

    public ClubMember()
    {
        super();
        personAge = 0;
        feePaid = false;
    }

    public ClubMember(String name1, String name2, int age, boolean paid)
    {
        super(name1, name2);
        personAge = age;
        feePaid = paid;
    }

    //methods
}
```

The ClubMember Class — Methods

```
public void setNameAge(String name1, String name2, int age)
{
    setName(name1, name2);
    personAge = age;
}
public int getAge()
{
    return personAge;
}
public boolean checkFeePaid()
{
    return feePaid;
}
public void setAge(int age)
{
    personAge = age;
}
public void setFeePaid(boolean b)
{
    feePaid = b;
}
```

The ClubMember Class — The toString Method

```
public String toString()
{
    String message = " Paid";
    if (!feePaid)
        message = " Not" + message;
    return super.toString() + " " + personAge + message;
}
```

The ClubMember Class — More Methods

```
public boolean isSenior()
{
     return (personAge >= SENIORAGE);
}

public double moneyDue()
{
     if (!isSenior())
         return NORMALFEE;
     else
         return SENIORFEE;
}

public void payAndPrintReceipt(PrintWriter f)
{
    if (!feePaid)
    {
        feePaid = true;
        printName(f);
        f.println("£" + moneyDue() + " received with thanks.");
    }
    else
        System.out.println("You have already paid your fee.");
}
```

The ClubMember Class — The printReminder Method

```
public void printReminder(PrintWriter f, Person manager)
{
    if (!feePaid)
    {
        f.print("Dear ");
        printShortName(f);
        f.println(",");
        f.println("Your fee of " + moneyDue() +
                  " pounds is now due.");
        f.print("Yours sincerely, ");
        manager.printName(f);
    }
}
```

The Club Class

```
import java.io.*;
public class Club
{
    private int maxMembers;
    private Person manager;
    private int numberOfMembers;
    private ClubMember[] members;

    public Club(int maxNumber, Person m)
    {
        maxMembers = maxNumber;
        manager = m;
        members = new ClubMember[maxMembers];
        numberOfMembers = 0;
    }

    // methods
}
```

The `Club` Class — Methods

```
public boolean placeAvailable()
{
    return (numberOfMembers < maxMembers);
}

public void register(ClubMember m)
{
    members[numberOfMembers] = m;
    numberOfMembers++;
}

public void listMembers()
{
     for(int i=0; i<numberOfMembers; i++)
     {
         System.out.println(members[i]);
     }
}
```

The `Club` Class — More Methods

```
public void sendReminders(PrintWriter f)
{
     for(int i=0; i<numberOfMembers; i++)
     {
         if (!members[i].checkFeePaid())
             members[i].printReminder(f,manager);
     }
}

public void resetFeePaid()
{
     for(int i=0; i<numberOfMembers; i++)
     {
         members[i].setFeePaid(false);
     }
}
```

The Club Class — Even More Methods

```
public int memberNumber(Person p)
{
    for(int i=0; i<numberOfMembers; i++)
    {
        Person m = members[i];
        if (m.equals(p))
            return i;
    }
    return -1;
}

public void payFee(Person p, PrintWriter f)
{
    int n = memberNumber(p);
    if (n == -1)
        System.out.println("Sorry, you are not a member.");
    else
        members[n].payAndPrintReceipt(f);
}
```

The SportsClubRunning Class

```
import java.io.*;
import java.util.*;

public class SportsClubRunning
{
  static Scanner k = new Scanner(System.in);

  public static void main(String[] args)
                                   throws FileNotFoundException
  {...}

  private static void printMenu()
  {...}

  private static Person readNames()
  {...}

  private static ClubMember readMemberData(Club club)
  {...}
}
```

The printMenu Method

```
private static void printMenu()
{
    System.out.println("------------------------------");
    System.out.println("MENU");
    System.out.println("f - finish");
    System.out.println("l - list all members");
    System.out.println("n - reset fee information");
    System.out.println("p - accept payment");
    System.out.println("r - register a new member");
    System.out.println("s - send reminders");
    System.out.println("------------------------------");
    System.out.println("Type a letter and press Enter");
}
```

The `main` Method

```
public static void main(String[] args) throws FileNotFoundException
{
    PrintWriter outFile = new PrintWriter("H:\\csc8012\\clubR.txt");
    Person boss = new Person("Tom","Smith");
    Club sportClub = new Club(2, boss);
    printMenu();
    char ch = k.next().charAt(0);
    k.nextLine();
    while (ch != 'f')
    {
        switch(ch)
        {
            //process single request
        }
        printMenu();
        ch = k.next().charAt(0);
        k.nextLine();
    }
    outFile.close();
}
```

The switch Statement from main

```
switch(ch)
{
    case 'l': sportClub.listMembers();
              break;
    case 'n': sportClub.resetFeePaid();
              break;
    case 'p': sportClub.payFee(readNames(),outFile);
              break;
    case 'r': if (sportClub.placeAvailable())
              {
                  ClubMember member = readMemberData(sportClub);
                  if (member != null)
                      sportClub.register(member);
                  else
                      System.out.println("You are already registered as a member.");
              }
              else
              System.out.println("Sorry, we cannot accept any more members.");
              break;
    case 's': sportClub.sendReminders(outFile);
              break;
    default:  System.out.println("Invalid entry, try again");
}
```

The readNames Method

```
private static Person readNames()
{
    System.out.println("Enter person's first name and surname,"
                    + " and press Enter");
    String name1 = k.next();
    String name2 = k.next();
    k.nextLine();
    return new Person(name1, name2);
}
```

The readMemberData Method

```
private static ClubMember readMemberData(Club club)
{
    Person p = readNames();
    if (club.memberNumber(p) == -1)
    {
        System.out.println("Enter " + p + "'s age, and press Enter");
        int age = k.nextInt();
        k.nextLine();
        return new ClubMember(p.getFirstName(), p.getSurname(),
                                                  age,false);
    }
    else
    {
        return null;
    }
}
```

The `equals` Method in the `Person` Class

```
public boolean equals(Person otherPerson)
{
 return (firstName.equals(otherPerson.firstName)
               && surname.equals(otherPerson.surname));
}
```

- Is this an example of overriding or overloading a method?

The `equals` Method in the `Person` Class

```
public boolean equals(Person otherPerson)
{
 return (firstName.equals(otherPerson.firstName)
                && surname.equals(otherPerson.surname));
}
```

- Is this an example of overriding or overloading a method?
- It is an example of method *overloading*. The `Person` class still inherits the `Object` class's `equals` method that takes a parameter of type `Object`.

Testing Person Objects for Equality (1)

```
public static void main(String[] args)
{
    Person p1 = new Person("Anna","Smith");
    Person p2 = new Person("Anna","Smith");
    System.out.println(p1.equals(p2));
    Object o1 = new Person("Anna","Smith");
    Object o2 = new Person("Anna","Smith");
    System.out.println(o1.equals(o2));
}
```

- The above program prints:

```
true
false
```

Testing `Person` Objects for Equality (2)

```
public boolean equals(Object obj)
{
    if (this == obj) return true;
    if (!(obj instanceof Person)) return false;
    Person p = (Person) obj;
    return (firstName.equals(p.firstName)
            && surname.equals(p.surname));
}
```

- After adding the above method to the `Person` class, both tests from the `main` method on the previous slide return `true`.
- The above method overrides the Object class's `equals` method.

Abstract Classes

- An *abstract method* is a method that has only the heading, with no body. The heading of an abstract method contains the reserved word `abstract` and ends with a semicolon. For example:

  ```
  public void abstract print();
  ```
- An *abstract class* is a class that is declared with the reserved word `abstract` in the heading:

  ```
  public abstract class ClassName { ... }
  ```
  - An abstract class can contain instance variables, constructors and non-abstract methods.
  - An abstract class can contain abstract method(s).
  - If a class contains an abstract method, then it must be declared as abstract.
  - You cannot instantiate an object of an abstract class type. You can only declare a reference variable of an abstract class type.
  - You can instantiate an object of a subclass of an abstract class, but only if the subclass *implements* (defines) all the abstract methods of the superclass.
- Abstract classes are used as superclasses from which other subclasses within the same context can be derived. They can implement their shared behaviour.

Interfaces

- An *interface* is a class that contains *mainly abstract methods* (prior to Java 8, only abstract methods were allowed in an interface) and/or named constants.
- Interfaces are defined using the reserved word `interface` in place of the reserved word `class`. For example:

  ```
  public interface Account { ... }
  ```
- An interface type specifies required operations. For example:

  ```
  public interface Account
  {
      void deposit(double amount);
      void withdraw(double amount);
      ...
  }
  ```
- You cannot instantiate an object of an interface type. You can only declare a reference variable of an interface type.

Implementing Interfaces

- The abstract methods specified by an interface (that are by default `public` and `abstract`) can be defined in many ways in the classes that implement the interface.
- A class that *implements an interface* must implement all its abstract methods (unless it is an abstract class). Its heading contains the reserved word `implements`. For example:

  ```
  public class SavingsAccount implements Account
  ```
- An interface can have many different implementations. For example, for the `Account` interface, we can also have:

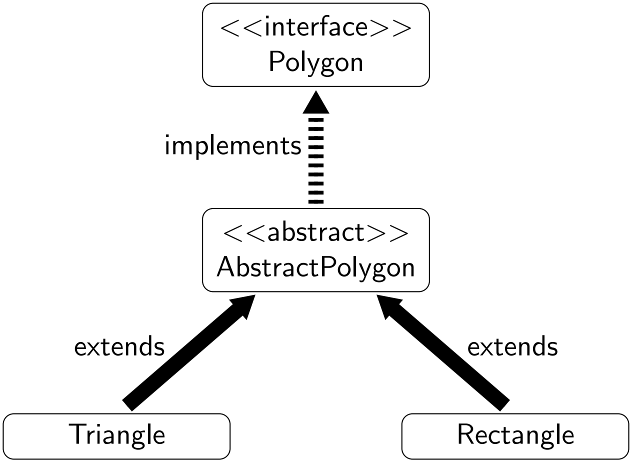
  ```
  public class CurrentAccount implements Account
  ```
- A class can implement many interfaces:

  ```
  public class ClassName implements InterfaceName1,
                                    InterfaceName2
  ```

Designing `Polygon` Classes

- We have been asked to design interfaces and classes for the application programs to be used by a maths teacher during geometry lessons about polygons.
- Pupils are supposed to learn how to compute the perimeters and areas of the chosen polygons.
- For simplicity, we will consider only polygons whose areas can be computed knowing only the lengths of their sides. For example: triangles, rectangles, squares.

Designing the `Polygon` Hierarchy



The Polygon Interface

```
import java.util.*;

public interface Polygon
{
    String getName();
    int getNumberOfSides();
    double[] getSideLengths();
    double perimeter();
    double area();
    void readSideLengths(Scanner s);
    void specialiseName(String preciseName);
}
```

The AbstractPolygon Class

```
import java.util.*;
public abstract class AbstractPolygon implements Polygon
{
    private String name;
    private int numberOfSides;
    private double[] sideLengths;

    public AbstractPolygon(String name, int numberOfSides)
    {
        this.name = name;
        this.numberOfSides = numberOfSides;
        sideLengths = new double[numberOfSides];
    }

    public String getName() { return name; }
    public int getNumberOfSides() { return numberOfSides; }
    public double[] getSideLengths() { return sideLengths;}
    public double perimeter() {...}
    public void readSideLengths(Scanner s) {...}
    public void specialiseName(String preciseName) { name = preciseName; }
    public abstract double area();
    public String toString() {...}
}
```

The AbstractPolygon Class — The perimeter Method

```
public double perimeter()
{
    double perimeterLength = 0;
    for (int i = 0; i < sideLengths.length; i++)
        perimeterLength += sideLengths[i];
    return perimeterLength;
}
```

The AbstractPolygon Class — The readSideLengths Method

```
public void readSideLengths(Scanner s)
{
    System.out.println("Please enter " + numberOfSides +
    " side lengths in order and press ENTER");
    for (int i=0; i < numberOfSides; i++)
        sideLengths[i] = s.nextDouble();
    s.nextLine();
}
```

The AbstractPolygon Class — The toString Method

```
public String toString()
{
    String s = name + ": [";
    for (int i=0; i < numberOfSides; i++)
    if (i != numberOfSides - 1)
        s = s + sideLengths[i] + ", ";
    else
        s = s + sideLengths[i] + "]";
    return s;
}
```

The Triangle Class

```
import java.util.*;
public class Triangle extends AbstractPolygon
{
    public Triangle()
    {
        super("Triangle", 3);
    }

    public void readSideLengths(Scanner s)
    {
        System.out.println("Enter the lengths of the sides of your triangle.");
        System.out.println("The length of any side in a triangle\n" +
        "is less than the sum of the lengths of the other two sides.");
        super.readSideLengths(s);
        double[] side = getSideLengths();
        if (side[0] == side[getNumberOfSides()-1])
                                        specialiseName("Equilateral Triangle");
    }

    public double area() // computed according to Heron's formula
    {
        double[] side = getSideLengths();
        double p = perimeter()/2;
        double value = p*(p-side[0])*(p-side[1])*(p-side[2]);
        return Math.sqrt(value);
    }
}
```

The Rectangle Class

```java
import java.util.*;
public class Rectangle extends AbstractPolygon
{
    public Rectangle()
    {
        super("Rectangle", 4);
    }

    public void readSideLengths(Scanner s)
    {
        System.out.println("Enter the lengths of the sides of your rectangle.");
        System.out.println("A well defined rectangle has two pairs " +
                           "of sides of equal lengths.");
        super.readSideLengths(s);
        double[] side = getSideLengths();
        if (side[0] == side[getNumberOfSides()-1]) specialiseName("Square");
    }

    public double area()
    {
        double[] side = getSideLengths();
        return side[0]*side[3];
    }
}
```

Testing Polygon Classes

```
import java.util.*;
public class UsingPolygons
{
   public static void main(String[] args)
   {
       Scanner sc = new Scanner(System.in);

       Triangle t = new Triangle();
       t.readSideLengths(sc);
       System.out.println(t);
       System.out.println("perimeter: " + t.perimeter());
       System.out.println("area: " + t.area());

       Rectangle r = new Rectangle();
       r.readSideLengths(sc);
       System.out.println(r);
       System.out.println("perimeter: " + r.perimeter());
       System.out.println("area: " + r.area());
   }
}
```

Polymorphism

- Method names in Java are *polymorphic* (literally, have many shapes). They have a different meaning depending on the context.
- Method polymorphism is achieved in Java by:
  - method *overloading* in classes;
  - method *overriding* in subclasses and
  - having methods with the same name (but *different implementations*) in different classes that implement the same interface.
- Reference variables in Java are polymorphic. A variable can hold references to objects of many types: the declared type or any subtype of the declared type.
- When declaring a reference variable, we can use two different types on the left and right sides of the assignment operator:

  ```
  StaticType var1 = new DynamicType(...);
             var1 = new DifferentDynamicType(...);
  ```
- The same method call may at different times invoke different methods, depending on the dynamic type of the variable (due to dynamic binding).