5COSC001W - OBJECT ORIENTED PROGRAMMING

Lecture 2: More on Classes

Dr Dimitris C. Dracopoulos

email: d.dracopoulos@westminster.ac.uk

1 Constructing Objects

Objects of a class are created by calling the *constructor* of the class with the `new` operator.

- Constructors are methods of the class and they always have the same name with the class itself.
- A class can have more than one constructors. Each one of them must have a different number and/or type of parameters.

Syntax:

```
new ClassName(parameters)
```

Note, that if no constructor is explicitly defined for a class, then the compiler will synthesise a default no-argument constructor, which can be used to create objects of the class. This will happen, only if no constructor is defined at all for a class. A zero-argument constructor will not be created by default, if a programmer defines a constructor with one or more arguments.

1.1 Constructor Example: The Rectangle Library class

The `Rectangle` class is defined in the Java library (package `java.awt`).

- A `Rectangle` object contains a set of numbers, specifying an area in a coordinate space by its top-left point (x, y), its width, and its height.

```
Rectangle r1 = new Rectangle(5, 10, 20, 30);
```

In the above code:

1. The constructor of the `Rectangle` class accepting four `int`s as parameters is called, to create a `Rectangle` object.
2. The created object has a top left corner at $(5, 10)$, and has a width equal to 20 and a height equal to 30.
3. The created object is returned, and it is stored (its address) in variable `r1`.

To find out what constructors are available for a class, the documentation for the class must be consulted or the code for the class be examined.

An alternative constructor of the `Rectangle` class could be called to create an object:

```
Rectangle r2 = new Rectangle();
```

The no-arguments constructor, creates a `Rectangle` object whose top-left corner is at (0, 0) in the coordinate space, and whose width and height are both zero.

2 Packages

Classes (including library classes) belong to a *package*. Thus, a package is a collection of classes.

- To be able to use a class belonging to a package in a program, the class must first be imported in the program. There are two ways to do that:
  1. Import all the classes of the package into the program, e.g.

     ```
     import java.awt.*;
     ```

  2. Import only the class that needs to be used, e.g.

     ```
     import java.awt.Rectangle;
     ```

  Note that the approach of importing all the classes of a package into a program, does not incur more overhead than the approach of importing a single class. This is because the Java Virtual Machine will actually load a class only when it is needed.

Example

```
import java.awt.Rectangle;

public class MoveTester {
    public static void main(String[] args) {
        Rectangle box = new Rectangle(5, 10, 20, 30);

        // Move the rectangle
        box.translate(15, 25);

        // Print information about the moved rectangle
        System.out.println("After moving, the top-left corner is:");
```

```
        System.out.println(box.getX() + ", " + box.getY());
    }
}
```

Note that in the last statement of the above program, the `+` operator is applied into strings. The operator creates a new `String` object which contains the concatenation of the characters in its `String` arguments.

3 Implementing Classes

Implement a class which simulates a bank account. The following operations must be available (abstraction mechanism for describing a real bank account):

- Deposit money.
- Withdraw money.
- Get the current balance.

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount {
    private double balance;

    /**
        Constructs a bank account with a zero balance.
    */
    public BankAccount()
    {
        balance = 0;
    }

    /**
        Constructs a bank account with a given balance.
        @param initialBalance the initial balance
    */
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    /**
        Deposits money into the bank account.
        @param amount the amount to deposit
    */
```

```
    public void deposit(double amount)
    {
        double newBalance = balance + amount;
        balance = newBalance;
    }

    /**
        Withdraws money from the bank account.
        @param amount the amount to withdraw
    */
    public void withdraw(double amount)
    {
        double newBalance = balance - amount;
        balance = newBalance;
    }

    /**
        Gets the current balance of the bank account.
        @return the current balance
    */
    public double getBalance()
    {
        return balance;
    }
}
```

4 Testing a Class

The following code illustrates how the implemented `BankAccount` class functionality can be tested. This is done by using another class which creates `BankAccount` objects and calls methods on them.

```
/**
   A class to test the BankAccount class.
*/
public class BankAccountTester {
    /**
        Tests the methods of the BankAccount class.
        @param args not used
    */
    public static void main(String[] args) {
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());

        BankAccount harrysSavings = new BankAccount(100);
        harrysSavings.withdraw(30);
```

```
        harrysSavings.withdraw(10);
        harrysSavings.deposit(20);
        double balance = harrysSavings.getBalance();
        System.out.println("Savings account balance: " + balance);
    }
}
```

When the above program is run, it displays:

```
1500.0
Savings account balance: 80.0
```

5 Overloading Methods

A class can have more than one methods with the same name, assuming that these methods have a different signature.

- The signature of a method consists of the combination of its name and its arguments (the specific order, number and type of arguments)

Note that both ordinary methods and constructors can be overloaded.

Example:

```
public class Printer {
    int errorCode;

    // constructor 1
    public Printer() {
        System.out.println("Constructor with no arguments called!");
    }

    // constructor 2
    public Printer(int i) {
        errorCode = i;
        System.out.println("Constructor with an int argument called!");
    }

    public void display() {
        System.out.println(errorCode);
    }

    public void display(int i) {
        System.out.println(i);
    }
```

```
    public void display(String s) {
        System.out.println(s);
    }

    public void display(int i, String s) {
        System.out.println(i + s);
    }

    public void display(String s, int i) {
        System.out.println(i + s);
    }
}
```

Note that the order of arguments is part of the signature of a method. Thus, both `display(int, String)` and `display(String, int)` can be present in class `Printer` because they are methods with different signatures (and the same name, i.e. `display` is overloaded).

6 Inheritance

Hierarchies of classes can be created. This is desirable for the following reasons:

- Reusability of code (reusing the functionality of existing classes).
- Code is easier to maintain.
- Polymorphic behaviour (objects are manipulated via reference variables of the base class).

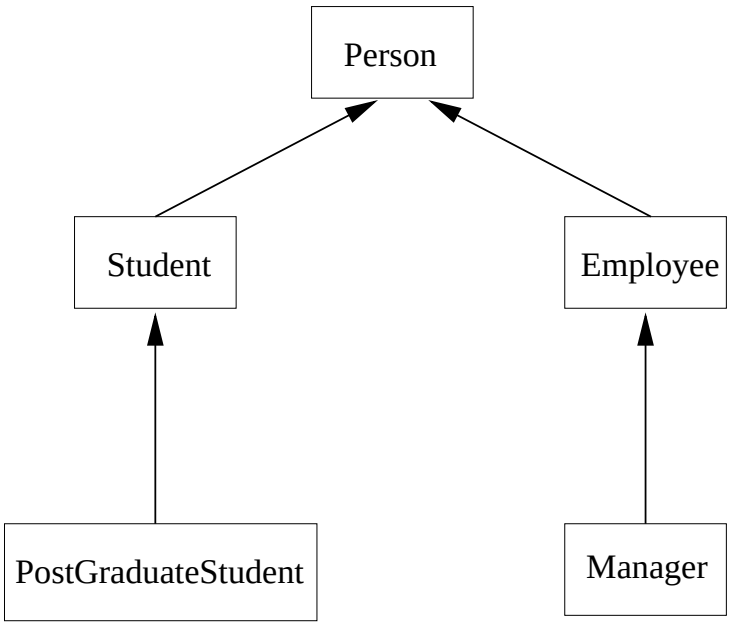

Figure 1: A inheritance tree of classes in UML form.

Figure 1 shows a hierarchy of classes in UML representation (UML is a modelling language for object oriented programming). A `Student` is a `Person` and a `PostgraduateStudent` is a `Student`. Similarly, `Employee` is a `Person` and `Manager` is an `Employee`.

Notice that the *is-a* attribute is characteristic for the relationship of a class with its parent class.

`Student` is a subclass (or child class) of `Person`, and `Person` is a superclass (or parent class) of `Student`.

Note that unlike C++, Java does not support multiple inheritance, i.e. a class can only inherit from a single other class.

6.1 Example:

```
class Person {
    private String name;

    public Person() {

    }

    public Person(String name1) {
        name = name1;
    }

    // initialise the name instance field of the object
    public void setName(String name) {
        /* this is a shortcut for the object we are currently in.
           Thus, this.name is the instance field name within the
           current object */
        this.name = name;
    }

    // prints all information about the object
    public void info() {
        System.out.println("\nname: " + name);
    }
}


class Student extends Person {
    private String school;

    public Student(String school, String name) {
        this.school = school;
        setName(name);
    }

    // prints all information about the object
```

```
    public void info() {
        // call info() method of Person class
        super.info();
        System.out.println("school: " + school);
    }
}


class PostGraduateStudent extends Student {
    private String firstDegree;  // what the first degree was on

    public PostGraduateStudent(String school, String name, String degree) {
        super(school, name);  // call constructor of parent class
        firstDegree = degree;
    }

    public void info() {
        super.info();  // call info() method of the parent class
        System.out.println("firstDegree: " + firstDegree);
    }
}

public class University {
    public static void main(String[] args) {
        Student s1 = new Student("IC", "John");
        Student s2 = new Student("MIT", "Helen");
        PostGraduateStudent s3 = new PostGraduateStudent("Westminster", "George", "music");

        s1.info();
        s2.info();
        s3.info();
    }
}
```

When the above program is run, it displays:

```
name: John
school: IC

name: Helen
school: MIT

name: George
school: Westminster
firstDegree: music
```

A subclass inherits from the parent class:

- All its methods.

- All its fields

Part of a subclass object is an object of the parent class, i.e. an subclass object contains a parent class object.

For example, although class `Student` does not contain a method `setName`, it is able to call the method, as it inherits it from the parent class `Person`. This is illustrated in the above code.

- Although a subclass inherits the fields and methods of the superclass, this does not mean that it can access these fields and methods directly. Private fields and methods of a superclass, can only be accessed by the subclass via calling public (or protected) methods of the superclass accessing them.

  Field `name` is `private` in class `Person`. The field is inherited by subclass `Student` which cannot access it directly because of its privateness. However, it can be indirectly accessed by calling the `public` method `setName` found in the parent class `Person`, which assigns the field a value.

  More details about what fields and methods of a class can be accessed by another class, will be described in Section "Access Specifiers".

In the `University` example, the `this` keyword is used. The `this` keyword refers to the object we are currently in. Therefore `this.name` accesses the `name` instance field of the current object, and `this.foo()` calls method `foo` on the current object (`foo` is synonymous to `this.foo()`).

The `this` keyword can also be used to call alternative constructors of the same class from inside a different constructor. Such a call must be the first statement in a constructor. For example in the code below, the call `this(y)` inside `D(String, int)`, calls the `D(String)` constructor of the class:

```
class D {
    String s;
    int i;

    public D(String x) {
        s = x;
    }

    public D(String y, int p) {
        this(y);  // calls D(String)
        i = p;
    }
}
```

7 Overriding Methods

When a class defines a method with the same signature as a method in a parent class, the method is overridden in the subclass[1].

In the example of the previous section:

[1]The signature of a method consists of its name and the number, type, order of its arguments.

- `Student` overrides the `info` method with its own implementation (initially it inherits the `Person` version of `info`).
- `PostGraduateStudent` overrides the `info` method with its own implementation (initially it inherits the `Student` version of `info`).

Contrast *overriding* with *overloading* described in a set of previous lecture notes.

8 The super keyword

The `super` keyword can be used for three purposes:

- To call a constructor of the parent class. In such a case, the `super` call must be the first statement in the constructor of the subclass:

```
public PostGraduateStudent(String school, String name, String degree) {
        super(school, name);
        firstDegree = degree;
    }
```

  The first line in the above constructor, calls the constructor `Student(String, String)` of the parent class.

- To call any method of the parent class, even if that method is overridden in the current class. There is no need to have such a call as the first statement in a method.
- To access a field of the parent class. For example, `super.x` in class `B` accesses the `A` instance field `x`:

```
class A {
    public int x;
}

class B extends A {
    public int x;

    void foo() {
        int y = super.x;  // accesses x within A, NOT x in B!
    }
}
```

  Note that `super` keywords cannot be combined together. Thus, `super.super.x` in an attempt to access field `x` of the grandparent class will produce a compiler error.