Introduction to Java and the Eclipse IDE

Steven Latré, Jeroen Famaey and Tom De Schepper – University of Antwerp

2016 - 2017

1 Introduction

The objectives of this tutorial are the following:

- Provide a simple introduction to the Eclipse IDE by walking through the creation of basic Java applications.
- Give an overview of the most important concepts in Object-Oriented (OO) programming widely used to design computer programs.

2 Before you start

- Download and install the appropriate software on your computer
  - Java SE Development Kit (JDK 7 – `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`)
  - Eclipse IDE (Java Developers Edition – Luna, Version 4.4 – `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr1`)
- Verify that the sample code is also available on your local computer. You can download the samples from Blackboard.

3 The Eclipse IDE

An Integrated Development Environment (IDE) is a single program in which all software development is done. It provides tools that can greatly increase the programming productivity. The following tools are or can be integrated in one Graphical User Interface (GUI):

- Source code editor
- Compiler and/or interpreter

- Building tools
- Source code debugger
- Online of offline help function
- GUI designer

Typically, a modern IDE is graphical and supports different types of programming languages. Some examples of commonly used IDEs are: Eclipse, NetBeans and Microsoft Visual Studio. During the lab sessions we will use the Eclipse IDE, which is released under an open source license and provides the services common to creating desktop applications. This brief tutorial will help you to become familiar with the Eclipse IDE. Specifically, you will learn how to create projects, create programs, compile, run and debug them.

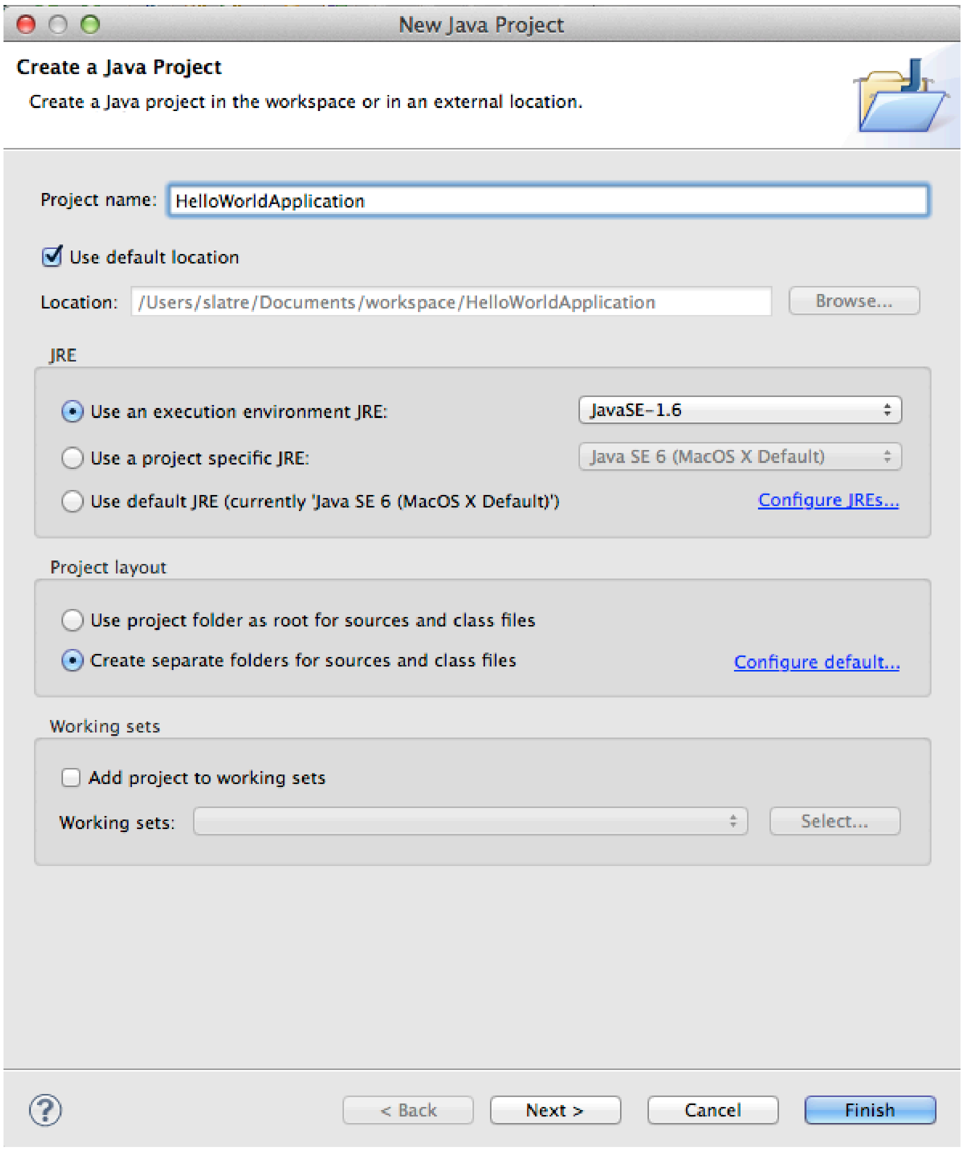
3.1 Tutorial (15 minutes)

A project contains the information about programs and their dependent files (e.g. build scripts). To create and run a program, you first have to create a project. To create an IDE project you should do the following:

1. Start the Eclipse IDE.
2. You may have to choose a workspace location. This is the default location where Eclipse will store all the Java projects.
3. In Eclipse, choose File → New → Java Project
4. In the "New Java Project" window, provide a name for the project (e.g., HelloWorldApplication). This is shown in Figure 1).
5. Click Finish

The project is created and a new project tree appears in the Eclipse Projects window (as shown in Figure 2). The layout on your computer may differ slightly from this screenshot. However, you can easily identify the following components:

1. The Package Explorer window is located in the upper left corner. This area contains a tree view of the components of the project, including source files, libraries that your code depends on, and so on.
2. The large block on the right hand side is the Source Editor. You can edit source code in here.
3. The pane at the bottom will give us information about possible compilation errors and will also show the console's output.
4. On the right hand side you have an "Outline" pane, which gives you information about the structure of the project.



Project name: HelloWorldApplication

[✓] Use default location

Location: /Users/slatre/Documents/workspace/HelloWorldApplication

[✓ Use an execution environment JRE:] JavaSE-1.6

Use a project specific JRE: Java SE 6 (MacOS X Default)

Use default JRE (currently 'Java SE 6 (MacOS X Default)')

Use project folder as root for sources and class files

[✓ Create separate folders for sources and class files]

Add project to working sets

Working sets:

Figure 1: The Eclipse "New Java Project" window

3.1.1 Using the Source Editor

Files and directories associated with an application project are organized in a logical way in the Package Explorer window. Double-clicking a source file automatically opens the file in the source editor to the right. You can now click somewhere in the source code, and start coding. When you write code this way, you will notice that the editor automatically highlights fragments of code according to the syntax of the programming language. Also, suggestions for code-completion are provided as you type.

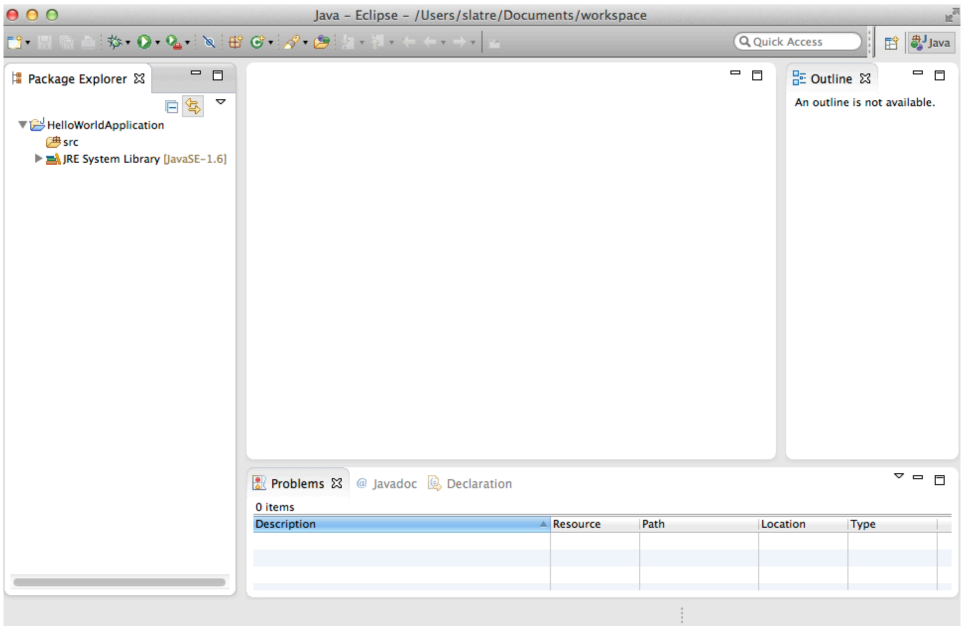


Figure 2: The Eclipse program

3.1.2 Creating a class

We will now create your first Java class. Choose File → New → Class and type in the name of the class (e.g., HelloWorldApplication). It is also recommended to define a package name (e.g., java_tutorial). If you want this class to be runnable (i.e., having a main method): enable the "public static void main" stub, as shown in Figure 3. Don't worry if you forget to enable this, you can always define the main method manually.

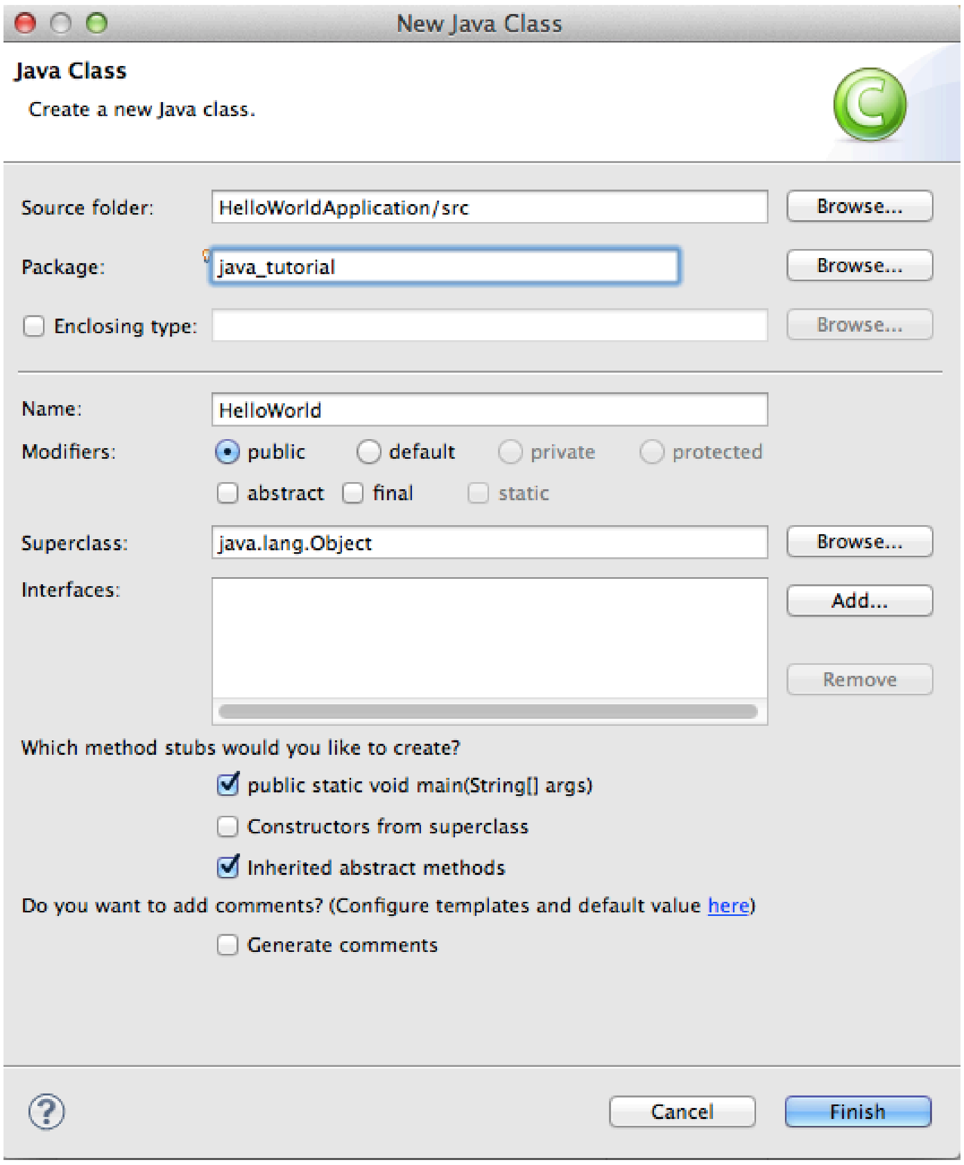
We will add a simple statement to our main class that will output something to the console. Type the following statement in the main method of our newly created class.

```
System.out.println("Hello world!");
```

3.1.3 Compiling the source code

Typically, the source code is automatically compiled as you type. You can turn this off by disabling Project → Build Automatically. If this is turned off, you can manually compile our source code by choosing Project → Build Project.

When you build the project, Eclipse will typically create a "bin" folder behind the scenes that contains the appropriate ".class" files. The standard Eclipse configuration does not show this class file by default. However, you can see it in the explorer window of your operating system (look for the location of the workspace and go into the "bin" folder).



Source folder: HelloWorldApplication/src

Package: java_tutorial

[ ] Enclosing type:

Name: HelloWorld

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[✓] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

Figure 3: The New Java Class window

3.1.4 Running the program

Once your project is built, you can run it. To do this, choose Run → Run. You can also click the "play" button on the tool bar. In case you forgot to build your project, you don't need to worry: Eclipse will build your project automatically when you click run. On the bottom pane a "Console" tab will pop up and it will show "Hello World". Congratulations, you successfully ran your first program.

3.1.5 Using the debugger (optional)

The Eclipse IDE contains a debugger utility enabling you to set breakpoints and execute programs line by line helping you to identify bugs. As the program executes, you can watch the values stored in variables, observe the methods that are being called, and get to know what events occur in the program. To start the Eclipse debugger click the "bug" icon instead of the "play" icon when running a program. Eclipse will switch to a debug view where you can monitor the program's execution.

4 Object-oriented concepts

4.1 Using a class

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in fields (or variables) and exposes its behaviour through methods (or functions). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation. A class is a blueprint or prototype that models the state and behaviour of a real-world object.

In the example below, we show a class Person that represents a possible implementation of a person.

```
public class Person {
        // Constant - male gender
        public static final int MALE = 0;
        // Constant - female gender
        public static final int FEMALE = 1;
        // Variables
        private String forename;
        private String surname;
        private int age;
        private int gender;

        /**
         * Constructor
         *
         * Creates an instance of the Person class with
         * specified field values
         *
         * @param firstname
         *            - firstname of the person
         * @param surname
         *            - surname of the person
         * @param age
         *            - age of the person
         * @param gender
```

```
 *             - gender of the person
 */
public Person(String firstname, String surname, int age,
    int gender) {
        this.firstname = firstname;
        this.surname = surname;
        this.age = age;
        this.gender = gender;
}

/**
 * Returns the firstname field of the person
 *
 * @return the firstname field of the person
 */
public String getFirstname() {
        return firstname;
}

/**
 * Set the firstname field of the person
 *
 * @param firstname
 *             - the firstname of the person
 */
public void setFirstname(String firstname) {
        this.firstname = firstname;
}

/**
 * Returns the surname field of the person
 *
 * @return the surname field of the person
 */
public String getSurname() {
        return surname;

}

/**
 * Set the surname field of the person
 *
 * @param the
 *             surname of the person
 */
public void setSurname(String surname) {
        this.surname = surname;
}

/**
 * Returns the age field of the person
 *
 * @return the age field of the person
 */
```

```
public int getAge() {
        return age;
}

/**
 * Set the age field of the person
 *
 * @param the
 *            age of the person
 */
public void setAge(int age) {
        this.age = age;
}

/**
 * Returns the gender field of the person
 *
 * @return the gender field of the person
 */
public int getGender() {
        return gender;
}

/**
 * Returns string representation of the gender
 * field of the person
 **
 * @return string representation of the gender field
 * of the person
 */
public String getGenderString() {
        return (gender == 0) ? "Male" : "Female";
}

/**
 * Set the gender field of the person
 *
 * @param the
 *            gender of the person
 */
public void setGender(int gender) {
        this.gender = gender;
}

/**
 * Increase the age of the person by the specified amount
 *
 * @param n
 *            - the number of years by which the age field
 *            should be increased
 */
public void increaseAge(int n) {
        this.age += n;
}
```

```
        /**
         * Set the full name of the person in one operation
         *
         * @param firstname
         *            - the firstname of the person
         * @param surname
         *            - the surname of the person
         */
        public void setFullName(String firstname, String surname) {
                this.firstname = firstname;
                this.surname = surname;
        }

        /**
         * Return a string representing the person
         *
         * @return details of the person ('firstname surname
         * (age: gender)')
         */
        @Override
        public String toString() {
                return new String(firstname + "␣" + surname
                    + "␣(Age:␣" + age
                        + "␣,Gender:␣" + getGenderString() + ")");
        }

        /**
         * Return a copy of the person
         *
         * @return a copy of the Person instance
         */
        public Person copy() {
                return new Person(firstname, surname, age, gender);
        }
}
```

The fields firstname, surname, age, and gender represent the object's state, and the methods increaseAge, setFullName, setters, and getters (these methods are preceded by set or get) for the fields define its interaction with the outside world. Notice that the class Person contains two static final fields. Static fields are associated with the class rather than its objects. As such, all objects that are an instance of Person share these fields. Static fields can be invoked using the class name (dot) field (or method in case of a static method – e.g. Person.MALE). The final keyword denotes the fields are constants and their values cannot be changed at runtime.

The class Person does not contain a main method. That's because it is not a complete application, but rather the blueprint for persons that might be used in an application. The responsibility of creating and using new Person objects belongs to some other class in your application. How this Person class can be used in a program is shown in the next example.

```
public class PersonDemo {
    public static void main(String[] args) {
        // Create two different Person objects
        Person person1 = new Person("John", "Doe", 25, Person.MALE);
        Person person2 = new Person("Jane", "Doe", 25, Person.FEMALE);
        // Invoke methods on those objects
        person1.increaseAge(3);
        person2.increaseAge(5);
        // Print object content
        System.out.println(person1.toString());
        System.out.println(person2.toString());
    }
}
```

In this program, two persons are created with some field values. Next, the internal state of the objects is updated by invoking one of the objects methods. Finally, the field values are retrieved using the toString() method.

The output of this program displays the firstname, surname, age, and gender for the two persons:

```
John Doe (Age: 28, Gender: Male)
Jane Doe (Age: 30, Gender: Female)
```

4.2 Aggregating classes

Aggregation is a relationship between two classes that can be described as a "has-a" relationship. For example, imagine a class Student that stores information about individual students at a school. Now let's say there is a class Subject that holds the details about a particular subject (e.g., history, geography). If the class Student is defined to contain a Subject object then the relationship between the two classes is the aggregation relationship. It can be said that the Student object has-a Subject object. This concept is shown in the code fragment below. There is an aggregation relationship between the two classes because the Student class contains an array of Subject objects.

```
class Subject {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Student {
    private Subject[] studyAreas = new Subject[10];
}
```

4.3 Class inheritance

Inheritance is a compile-time mechanism that allows you to extend a class (called the base class or superclass) with another class (called the derived class or subclass), primarily for the purpose of code reuse. This derived class can be seen as an extension of the base class because it inherits the methods and data of the base class, while adding its own functionality. Java only allows a class to have one immediate base class, also known as single class inheritance. This concept is illustrated in the next example, where a three-dimensional point uses a two-dimensional point as the base class.

```
public class Point2D {
        private double x,y;

        public Point2D(double x, double y) {
                this.x = x;
                this.y = y;
        }

        public double getX() {
                return x;
        }

        public double getY() {
                return y;
        }

        @Override
        public String toString() {
                return new String("("+x+", "+y+")");
        }
}


public class Point3D extends Point2D {
        private double z;

        public Point3D(double x, double y, double z) {
                super(x, y);
                this.z = z;
        }

        public double getZ() {
                return z;
        }

        @Override
        public String toString() {
                return new String("("+getX()+", "+getY()+", "+getZ()+")");
        }
}
```

The class Point3D reuses the code of the class Point2D and extends it with a third coordinate to represent points in tree-dimensional space. Note that when you construct an object, the default base class constructor is called implicitly, before the body of the derived class constructor is executed. So, objects are constructed top-down under inheritance. Since every object inherits from the Object class, the Object() constructor is always called implicitly. However, you can call a superclass constructor explicitly using the built-in super keyword. This keyword, if used, always needs to be the first statement of a constructor.

4.4 Polymorphism

Polymorphism allows you to assign a different behaviour or value in a subclass, to something that was already declared in a parent class. For example, a method can be declared in a parent class, but each subclass can have a different implementation of that method (the name remains the same but the implementation differs). This allows each subclass to differ, without the parent class being explicitly aware that a difference exists.

Polymorphism in Java is classified into two general types:

1. **Compile-time polymorphism:** the overloading occurs when several methods having the same names differ in their method signature. Overloading is determined at the compile time and the overloaded methods may all be defined in the same class. This type of polymorphism is illustrated in the next code example.

```
class Person {
        private String name;
        private String nickName;

        public void setName(String name) {
                this.name = name;
        }

        public void setName(String name, String nickName) {
                this.name = name;
                this.nickName = nickName;
        }
}
```

2. **Run-time polymorphism:** occurs when a subclass method has the same name and signature as a method in the parent class. When overriding methods in this way, Java determines the proper methods to call at the programs run time, not at the compile time. This concept is clarified in the next example.

```
public class Programmer {
    public Programmer() { }

    public String getInfo() {
        return "I'm␣a␣programmer!";
    }
}


public class JavaProgrammer extends Programmer {
    public JavaProgrammer() { }

    public String getInfo() {
        return "I'm a Java programmer!";
    }
}


public class PolymorphismTester {
    public static void main(String[] args) {
        Programmer programmer = new Programmer();
        Programmer javaProgrammer = new JavaProgrammer();
        System.out.println(programmer.getInfo());
        System.out.println(javaProgrammer.getInfo());
    }
}
```

Running the main program PolymorphismTester results in the following output:

```
I'm a programmer!
I'm a Java programmer!
```

Although both the objects are of types 'Programmer' (look at the left side of = sign in the main program PolymorphismTester), a call

- programmer.getInfo() returns → "I'm a programmer!"
- javaProgrammer.getInfo() returns → " I'm a Java programmer!"

because the getInfo() has a different implementation in the subclass.

4.5 Interfaces

An interface is a specification, or contract, for a set of methods that a class, that implements the interface, must conform to in terms of type signature of its methods. The class that implements the interface provides an implementation for each method. The interface itself can have either public, package, private or protected access defined, but methods are only declared and not implemented (the methods do not have a body). You can define data in an interface, but it is less commonly used. If there are data fields defined in an interface, then they are implicitly defined to be public, static and final. The example below demonstrates how to use an interfaces. First, a interface Shape is given containing only one public method calculateArea(). Please note that this is only a method definition: there is no body containing the implementation of that method in the interface class. Each class implementing this interface must provide an implementation for this method signature. As each shape object must conform to the interface you can safely invoke the calculateArea(), because objects that implement the interface must have an implementation for all the methods. Furthermore, when dealing with class hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn't depend on specific types.

```
public interface Shape {
    public double calculateArea();
}

public class Rectangle implements Shape {
        private double length , width;

        public Rectangle(double length, double width) {
                this.length = length;
                this.width = width;
        }

        public double getLength() {
                return length;
        }

        public double getWidth() {
                return width;
        }

        public double calculateArea() {
                return length*width;
        }
}


public class Circle implements Shape{
        private double radius = 0;

        public Circle() {}

        public Circle(double r) {
                radius = r;
        }

        public double getRadius() {
                return radius;
        }

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

Since you can inherit multiple interfaces, they are very often a useful mechanism to allow a class to have different behaviors in different situations of usage by implementing multiple interfaces.

5 Exercises

5.1 Using classes

5.1.1 Circle class

The code fragment below provides the implementation of a class Circle.

```
public class Circle {
    protected double radius = 0;

public Circle() {} // default constructor
        public Circle(double r) { // constructor
        radius = r;
    }

    public Circle(Circle c) { // copy-constructor
        this(c.getRadius());
    }

    public double getRadius() { // accessor
        return radius;
    }

    public void changeRadius(int r) { // modifier
        radius = r;
    }

    public double calculateGirth() {
        return 2.0*radius*Math.PI;
    }

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

    @Override
    public String toString() { // Return circle info
        return "Circle␣:\n\tradius␣=␣"+radius+"\n";
    }
}
```

Given this circle class, create a Java program that

- creates 5 circles with arbitrarily chosen radius values
- displays the info of the circles on the screen
- finds the two circles with the largest radius, calculates the sum of the two girth values and prints the result to the screen

Note: Try to keep your code as compact as possible and try avoiding duplicate method calls.

5.1.2 Complex numbers

Given the class Complex (see source files) representing a complex number, calculate the sum, difference, product and quotient of the following complex numbers:

- 5 + 4i and 3 - 6i
- 7i and 8
- i and -3 - 2i

Note: Try to keep your code as compact as possible and try avoiding duplicate method calls.

5.1.3 Vehicle

Create a class Vehicle using the Eclipse IDE containing the following fields: brand, serial number and fabrication_year. Choose an appropriate datatype for each variable and make them publicly accessible (read-only) using methods. Also provide a constructor and override the method toString() of the Object class.

5.1.4 Class book

Create a class Book containing fields that allow the class to store the author, category identification_number, title and if it is currently_borrowed. The borrowing status variable is the only field that should be mutable. Provide a constructor and the necessary methods for this class to function.

5.2 Aggregating classes

Model a building using aggregation. A building contains rooms and staircases. A room contains desks and chairs. Program the classes Building, Room, Staircase, Desk and Chair with some sample fields. Make use of aggregation to define the relationships between the classes.

5.3 Class inheritance

Consider the UML diagram in Figure 4.

Create a class Person containing the fields: name, gender and birth_year. Extend this class with a class Employee and add new variables to store an employee identification_number, the role and department_name where he or she is active. Next, extend the Employee class with 3 subclasses: a commissioned-based employee, a hourly employee and a salaried employee. Each subclass of

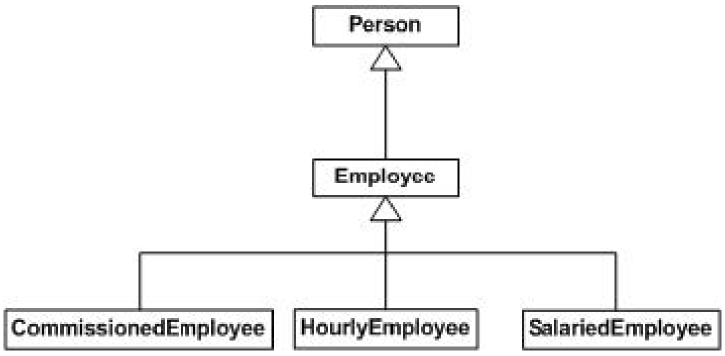


Figure 4: UML Diagram for the class inheritance exercise

Employee should have a calculateSalary() method, which is different for each subtype Employee. Finally, provide a program to test the different classes, therefore:

- create an instance of each subclass of Employee and set their appropriate values
- display info about each employee to the screen

5.4 Polymorphism

In this exercise we will practice the polymorphism concept. Therefore, create a base class named Shape containing one method called draw(). When this method is invoked it should return a String containing a message like: Im a Shape drawing. Create subclasses of the Shape class according to the model illustrated in Figure 5 below.

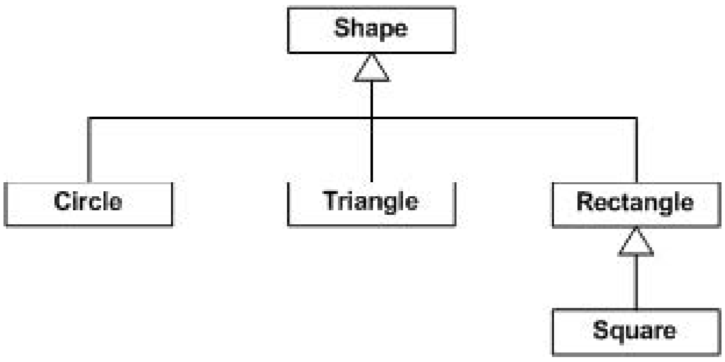


Figure 5: UML Diagram for the polymorphism exercise

Override the draw() method in each subclass and provide an appropriate implementation. Finally, test the polymorphism principle by creating an object of each subclass and invoking the draw method on the Shape objects.

5.5 Interfaces

The following code fragments present the source code for the interface Sortable and a basic sorting algorithm (bubble sort).

```
interface Sortable {
        public boolean idIsSmallerThan(Sortable s);
}

class BubbleSort {
        public static void sort(Sortable[] a) {
                for (int i = 0; i < a.length - 1; i++) {
                        for (int j = 0; j < a.length - i - 1; j++) {
                                if (a[j + 1].idIsSmallerThan(a[j])) {
                                        Sortable temp;
                                        temp = a[j];
                                        a[j] = a[j + 1];
                                        a[j + 1] = temp;
                                }
                        }
                }
        }
}
```

Given these code fragments, do the following:

- Create the class Order implementing the interface Sortable. This class should be able to store the order identification number and the order description. Provide an implementation of the idIsSmallerThan method. This method should return true if the order identification number of the current object is lower than that of the one provided.

- Create an array containing 5 orders and print the fields of the array elements to the screen.

- Do the sorting by invoking the method sort on the BubbleSort class. Print again the contents of the array to the screen and verify the orders are sorted in ascending order.