Lab 6

This lab continues the case study of Simplified Blackjack with amendments of design. The student has to demonstrate their skills and understanding with the implementation of abstract class, interface and method overriding.

Lab Guide and Worksheet

BITP 3113 Object Oriented Programming

Table of Contents

[Learning Outcomes 2](#_Toc400983599)

[Abstract method and Abstract Class 2](#_Toc400983600)

[Java interface 4](#_Toc400983601)

[Implementing Java interface 5](#_Toc400983602)

[Case Study 7](#_Toc400983603)

[Exercise 1 7](#_Toc400983604)

[Exercise 2 8](#_Toc400983605)

[Exercise 3 8](#_Toc400983606)

[Exercise 4 9](#_Toc400983607)

[Exercise 5 9](#_Toc400983608)

[Exercise 6 9](#_Toc400983609)

[**How are you doing?** 11](#_Toc400983610)

6 Polymorphism

# Learning Outcomes

At the end of this lab, the student is able to:-

* Define abstract class and abstract method
* Extends from abstract class and override abstract method
* Define Java interface class
* Implements to Java interface class
* Override method from Java interface class

# Abstract method and Abstract Class

Object oriented uses two types of implementation for a method which are concrete method and abstract method. A Concrete method is method with implementation. An empty method is still considered as a concrete method. An abstract method is a method without implementation. When a class has an abstract method, the class must be defined as an abstract class. Figure 1shows the syntax to declare an abstract method and abstract class.

|  |
| --- |
| accessSpecifier abstract class className {  accessSpecifier abstract returnType methodName (parameter1, parametern);  } |

Figure : Syntax to declare abstract method and abstract class

|  |  |
| --- | --- |
| An abstract class can have other class properties including concrete method. In UML, abstract class and abstract method is represented with italic font as shown in Figure 2. Figure 3 shows an example of valid abstract class. | Figure 2 : UML representation of abstract class |

|  |
| --- |
| public **abstract** class People {    private String address;    **public abstract void displayDetails();**  public void setAddress(String address){  this.address = address;  }  public String getAddress () {  return this.address;  }  } |

Figure : Example of abstract class

The purpose of having an abstract class is to allow increases the degree of reusability to a higher level. It should accelerate design and implementation of a system. However, an abstract class to let other classes to extend to it and override its own implementation. Figure 4 represent inheritance from abstract class in UML.



Figure : Student inherits from abstract class People

A subclass that extends from an abstract class must override all abstract methods. Figure 5 shows an example of class Student extending form an abstract class, People and overriding an abstract method, displayDetails().

|  |
| --- |
| public class Student **extends People** {    private String name;  private String matricNo;  private double cgpa;  private int totalCredit;  public Student(String name, String matricNo) {    this.name = name;  this.matricNo = matricNo;  this.cgpa = 0.0;  this.totalCredit = 0;  }    **public void displayDetails(){**  **System.out.println(super.getAddress());**  **}**  // other implementations    } |

Figure : Overriding abstract method

# Java interface

Java does not support multiple inheritance. interface is used to solve this problem. A Java interface has a class like structure that has methods without implementation. In UML, Java interface is presented as shown in Figure 6. The relationship between the class and interface is called realize.



Figure : Interface representation in UML

Figure 7 shows the syntax to define an interface.

|  |
| --- |
| accessSpecifier interface interfaceName {  accessSpecifier returnType methodName (parameter1, parametern);  } |

Figure : Syntax to declare interface

The purpose of interface is to realize inheritance and polymorphism. This is to promote greater abstraction and encapsulation for the program. Figure 8 shows an example of interface and declaration of method without implementation.

|  |
| --- |
| interface Assessment {    String getGrade (int Marks);  String assesingProgress(int marks);  } |

Figure : Example of interface

## Implementing Java interface

An interface allows the programmer to provide its own implementation. Interface is to realize inheritance. Therefore a class must subclass from interface using implements keyword. A class may implements more than one interface. Figure 9 shows the syntax to implement interface.

|  |
| --- |
| accessSpecifier className implements interface1, interfacen {  // details of implementation  } |

Figure : Syntax to implements interface

A subclass that implements an interface must override all methods that are defined in the interface. If some of the methods are not required by the subclass, it is necessary to provide an empty implementation. Figure 10 shows an example of PhD class that implements Assessment interface from Figure 8. All methods are overridden in PhD class. getGrade() method is not required in PhD class therefore the method has an empty implementation.

|  |
| --- |
| public class PhD implements Assessment {  // continuation from previous definition  String getGrade (int Marks) {  // details of implementation  }    String assesingProgress(int marks) {  if ((marks >= 90) && (marks <= 100)) {  return “Pass with no amendments”;    } else if ((marks >= 80) && (marks <= 89)) {  return “Pass with minimum amendments”;  } else {    return “Work is not sufficient for PhD”;  }    }    } |

Figure : Example of PhD implementing Assesment interface

A class can may implements as many interface as it is required and at the same time a class may extends from one subclass. This is valid in Java as shown in Figure 11.

|  |
| --- |
| public class PhD extends Student implements Assessment {  // details of implementation  } |

Figure : A class extending from a subclass and implements an interface

# Case Study

The System Analyst finds that the requirement of the application has changed due to the changes of the business. New classes are added and some of the changes are applied to the classes. The new design is shown n Figure 12.

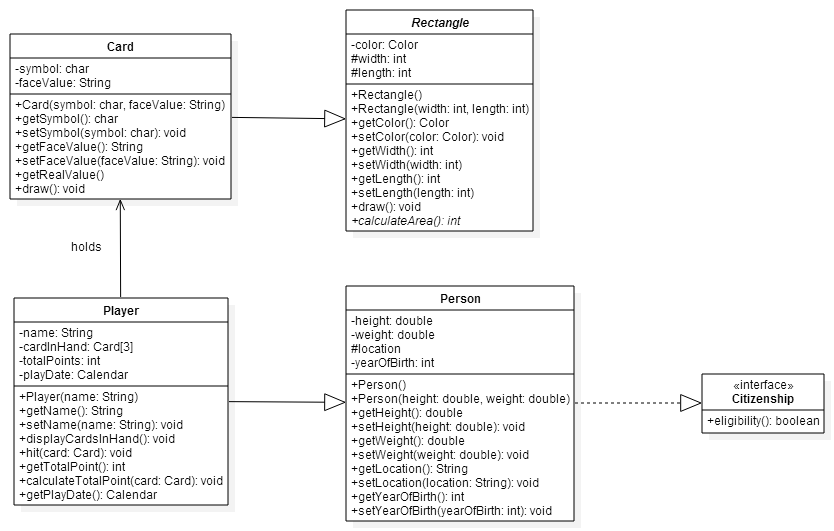


Figure : New design for Simplified BlackJack

Exercise 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Open Rectangle.java in Notepad++. 2. Add a new method named calculateArea() as show in Figure 12. This is an abstract method. 3. You might need to some changes to Rectangle.java to implement the design in Figure 12. 4. Save and compile Rectangle.java. Re-compiles if there are any errors. 5. Record the result in Table 1. | Table 1: Compilation result for Rectangle.java   |  |  | | --- | --- | | Expected Result | Result | | Rectangle.java compiles successfully | Pass / Fail | |

Exercise 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Open Card.java in Notepad 2. Override the abstract method. The method should calculate the area of the card and return the result of the calculation. 3. Save and compile Card.java. Re-compiles if there are any errors. 4. Record the result in Table 2. | Table 2: Compilation result for Rectangle.java   |  |  | | --- | --- | | Expected Result | Result | | Card.java compiles successfully | Pass / Fail | |

Exercise 3

1. Create a Java class named DemoCard. This class will have a main() method.
2. The class will display output as shown in Table 3.
3. Save, compile and run DemoCard.java.
4. Record the result in Table 3.

Table : Output from DemoCard.java

|  |  |
| --- | --- |
| Expected Result | Result |
| Rectangle 1 : 10 x 6 (Total Area is : 60.0)  +----------+  |C |  | |  | |  | |  | K|  +----------+  Rectangle 2 : 5 x 2 (Total Area is : 10.0)  +-----+  |D 5|  +-----+ | Pass / Fail |

Exercise 4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Create a Java interface named Citizenship as shown in Figure 12. 2. Save and compile Citizenship.java. Re-compiles if there are any errors. 3. Record the result in Table 4. | Table 4: Compilation result for Citizenship.java   |  |  | | --- | --- | | Expected Result | Result | | Citizenship.java compiles successfully | Pass / Fail | |

Exercise 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Open Person.java in Notepad++. 2. Implements the design in Figure 12. 3. Override method defined in the interface. That method will evaluate the location of the Player. The method will return false if the Player does not reside either in Denmark, Algeria or France. 4. Save and compile Person.java. Re-compiles if there are any errors. 5. Record the result in Table 4. | Table 5 : Compilation result for Person.java   |  |  | | --- | --- | | Expected Result | Result | | Person.java compiles successfully | Pass / Fail | |

Exercise 6

1. Create a class named DemoBlackJackImplements. This class will have method main().
2. Create a fourth Player. Set the player card so that he will have 21 points. This player will reside in India.
3. DemoBlackJackImplements will produce an output as shown in Table 6. The program will evaluate the players point if the players reside in one of the location as stated in Exercise 5.
4. Save and compile DemoBlackJackImplements.java. Re-compiles if there are any errors.
5. Record the result in Table 6.

Table : Output from DemoBlackJackImplements.java

|  |  |
| --- | --- |
| Expected Result | Result |
| This is to demo Black Jack game that uses Interface and Abstract class  ----------------------------------------------------------------------  Player 1 : Emma McKay @ Melaka (Total point : 14)  Player 2 : Ahmad Ismail @ Denmark (Total point : 18)  Player 3 : Mr Zidane @ Algeria (Total point : 22)  Player 4 : Madhu Taneja @ Mumbai, India (Total point : 21)  -----> Player 1 Emma McKay is not eligible.  \*\*\* Winner is Player 2 : Ahmad Ismail  +----------+  |D |  | |  | |  | |  | 2|  +----------+  +----------+  |S |  | |  | |  | |  | K|  +----------+  +----------+  |S |  | |  | |  | |  | 6|  +----------+  -----> Player 4 Madhu Taneja is not eligible. | Pass / Fail |

# How are you doing?

This section is to self- evaluate your skills defining and creating objects. Calculate the number of PASS and FAIL (including incomplete) from Exercise 1 until 6.

|  |  |
| --- | --- |
| Number of PASS result | Number of FAIL result |
|  |  |

**Score Guide**

If (Total Score for 2nd column == 0){

“You’re damn good!”

} Else If (Total Score for 1st column > Total Score for 2nd column) {

“You’re good. You can do better. Finish all your exercises.”

} Else {

“Work harder, dude! You should finish all the exercises.”

}

Complete the table below and email it to your lab coordinator.

|  |  |  |
| --- | --- | --- |
| NAME | MATRIC NO | SECTION |
|  |  |  |