This lab is a student-driven lab. The student will take responsibility of his own learning and go over the material to fully understand it. Seek external resources on OOP and Java as needed.

**Objectives**

1. **Very Important:** Learning OOP concepts. 2- Learning Java programming essentials

**Part 1 – Classes and Objects**

**Important Concepts**

**Class**: A class is a description of an aggregated set of data attributes and operations that represent a concept. For example, the class Car has the attributes: model, make, year, color and has the operations: speed, brake, turn right, turn left, etc. A class is a user-defined type.

**Object**: An object is an instance of a class. It is the actual implementation of one item that belongs to a certain class. For example, Ali’s car is an instance of the class car and it is Nissan, Sunny, 2008, white, etc. and it can perform all the operations that are defined for the class Car.

**Object Oriented Programming**: Object-oriented programming is developing programs by creating objects and defining how they interact with each other rather than creating functions / methods.

**Data Field / Attribute:** It is a variable that defines one of the characteristics of objects that belong to a certain class. An attribute will have different values for different objects belonging to the same class, e.g., Ali’s car color is white, but Mohammad’s is red and Sayed’s is metallic.

**Method / Operation**: Is a function that can be preformed on the objects of a certain class.

**Reference Variable**: This is a variable that points to an object. The variable does not have any useful data in itself, but it contains the address of the actual object in memory.

**UML**: A visual language used represent and model object oriented programs.

**Access Specifiers**: These are keywords that define who can access the class members (data fields and methods). The two important ones are **private** which means that only methods members of the same class can access this private class member and **public** which means that all methods in any other class can access this class member

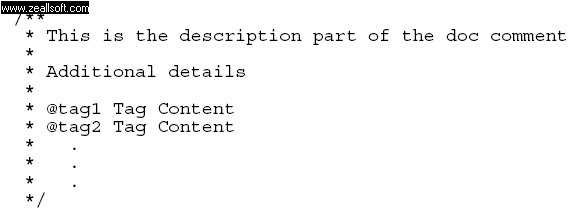
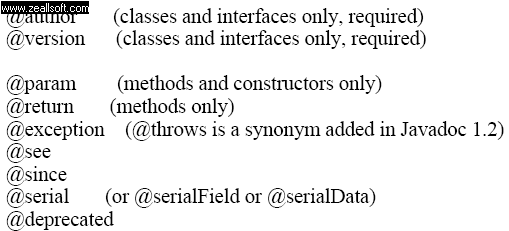


**Java Coding Style**: These are a set of rules that make the code readable and easier to maintain. See the attached document.

**Class Representation in UML**: is as show in the figure. – is private and + is public.

**Java Doc:** Java Doc is a utility with Java that allows the developer to add comments in a certain format and generate Html documentation for his classes automatically from these comments. A summary of Java Doc commenting style is shown below.

**Student Activities**



**Activity 1:**

Expand the attached code (which includes **Square** and **Rectangle** classes and a **Demo** class to show how to use them). Build a **Circle** class and a **Triangle** class and add to them the necessary attributes and methods. To do so:

1. Draw the classes and their data fields and methods in UML notation
2. Develop the classes in Java
3. Follow the coding style attached
4. Add adequate Java Doc comments and generate Html documents for the classes
5. Change the **Demo** class to give the options to create circles and triangles.

**Activity 2:**

Create a class that represents an alarm clock that displays time in 12 hours format. Create a demo to show how objects of this class work. Follow the same steps in activity 1. The class will have these attributes:

* **hours**
* **minutes**
* **seconds**
* Any other attributes you like to add

It will have the following methods:

* **Clock (int h, int m, int s)**
* **setTime (int h, int m, int s)**
* **incrementSeconds (int addedSeconds)**
* **incrementMinutes (int addedMinutes)**
* **incrementHours (int addedHours)**
* **toString ()**
* Any other methods you like to add

**Part 2 – Constructors, Overriding and Binding**

**Important Concepts**

**Constructor**: A constructor is a method that is called automatically when an object is created. Usually, a constructor is used to initialize the data fields of the object. A constructor has the same name of the class and has no return type. A constructor cannot return any value and typically has a **public** access modifier.

**Default Constructor**: It is provided automatically by Java if the developer does write a constructor. The default constructor sets the object’s numerical data fields to 0 and boolean values to false. Reference variables (that point to objects) are set **null**.

**Method Signature**: It is a distinguishing header for that method that defines (1) its name and (2) the names and types of method parameters.

**Method Overloading**: This means that two or more methods in a class may have the same name as long as their parameter lists are different.

**Method Binding**: The process of matching a method call with the correct method is known as binding.

**Java Packages:** Are ready made libraries of classes provided by Java for the developer. They make the Java Application Programmable Interface (API). To invoke a package, use **import** statement.

\* You must use proper coding style and generate documentation for your programs.

**Student Activities**

**Activity 1: Constructors**

For the given code of a **Clock** class, it is required to add to the class the following constructors. Create a demo to show the usage of these constructors.

* **Clock (int givenHours)**

Sets **hours** to **givenHours** and **minutes** and **seconds** to 0.

* **Clock (int givenHours, int givenMinutes)**

Sets **hours** and **minutes** to **givenHours** and **giveMinutes** and **seconds** to 0.

* **Clock (int givenHours, int givenMinutes, int givenSeconds)**

Sets **hours**, **minutes** and **seconds** to the given values.

* **Clock (Clock givenClock)**

Sets **hours**, **minutes** and **seconds** to the same values of another given **Clock**.

**Activity 2: Method Overloading**

For the given code of a **Clock** class, you can see after activity 1 that the constructor is overloaded. It is required to overload **incrementSeconds**, **incrementMinutes** and **incrementHours** to have two versions. The first version does not take any parameters and increments the corresponding attribute by one. The second takes an integer value and increments the corresponding attribute by this value. It is possible to build the second version by calling the first version several times, although this will take longer to execute using loops.

**Activity 3: Identifying Classes**

For the given problem description, identify the necessary classes for developing a solution in Java. Apply the method you learned in CS214 by identifying nouns and keeping only relevant nouns at the end. Then for each class you identified, suggest the necessary attributes (data fields) and operations (methods) of the class and draw a UML model for this class.

**Extra**: If you have time in the lab or if not, when you go home, implement the classes you identified in Java and a demo application to show how they work

**Problem Description**: It is required to develop a system for managing dry cleaning services. At first a customer brings one or more pieces of cloths for cleaning. These can be suits, jackets, trousers, skirts, dresses, blankets, etc. Then the receptionist registers the customer information including name, address and phone number. Then he opens a service request for the customer that includes a description of the items to be cleaned, the cost of cleaning, the due date of the items and the delivery method. (either the customers picks the items or they are delivered to his address) items and the delivery method.

**Part 3 – Static Methods, Deep Copy and Aggregation**

**Important Concepts**

A **static method** is a method that belongs to an entire class and not to any specific instance of this class. Such a method is called on the class name. It is usually a utility or service method that is not particularly related to a specific object.

A **static data field** is a data field that belongs to the class and not to any specific object. Hence, there is only one copy from this data field that all objects from this class can modify and access. Also, it can be accessed directly using the class name. This is usually for constants that do not depend on instances of the same object like PI or taxRate.

**toString()** method is a default t method with every class. However it does not of much use unless you override it and write your own version that does what you want it to do.

**equals()** method is a default t method with every class that compares that addresses of two objects. However, to be useful, you need to override it and write your own version that compares the objects in a meaningful way for your application.

**Shallow Copy** is creating a copy if the address or reference of an object, which means that two variables will be pointing to the same object.

**Deep Copy** is creating a new copy of an object with the same values for the data fields.

**Coding Style** is a set of rules for writing readable and organized code that is easy to understand and maintain.

**Aggregation** happens when a class owns an object of another class as a data field in this class.

**Student Activities**

**Activity 1: Static Methods and Data Fields**

For the given **CurrencyConverter** class and the **CurrencyConverterDemo** class, it is required to extend the class functionality to allow converting between Euros and Egyptian pounds and between British pounds and Egyptian pounds. It is required to modify the demo to allow the user to choose the conversion he likes and input the amount he wants to convert, and then display the result.

Add any necessary static data fields and methods.

**Activity 2: toString and equals Methods and Deep Copy**

For the given **Car** class override **toString** and **equals** methods so that they make useful functions. Create a method **Car getDeepCopy ()** that returns a new car with the same attributes. Then write a demo to show the use of these methods.

**Activity 3: Class Aggregations**

Write an application that calculates the price of carpeting rectangular rooms. To calculate the price, you multiply the area of the floor (width x length) by the price per square foot of the carpet.

First create a class called **RoomDimension** that has two fields: **length** and **width**. This class should have a method to return the room area.

Then create a class **RoomCarpet** that has a **RoomDimension** object as a field. It should have a field for the cost of the carpet per square foot and a method for returning the total cost of the carpet.

A UML diagram is shown for these two classes.

Create an application to demonstrate the use of these two functions.

|  |
| --- |
| **RoomCarpet** |
| * + **size : RoomDimension**   + **careptCost : double** |
| **+ RoomCarpet (dim : RoomDimension, cost : double)**  **+ getTotalCost() : double**  **+ toString() : String** |

|  |
| --- |
| **RoomDimension** |
| * + **length : RoomDimension**   + **width : double** |
| **+ RoomDimension (len : double, wid : double)**  **+ getArea() : double**  **+ toString() : String** |

**Part 4 – Static Methods, Deep Copy and Aggregation**

**Important Concepts**

**Inheritance** is deriving a new class from an existing class using **extends** keyword. The new class will have all the methods and data fields of the original class plus its own ones. The original class is called **parent** or **super class**. The new one is called **child** or **base class**.

**Overriding** happens when a child class re-implements a method that was already inherited from the parent class. Do not mix **overriding** with **overloading**.

**Object** is the parent (or grandparent) class of every class in Java. It give its children a basic form of some methods like **toString (),** **equals ()**, **clone ()** and **getClass ()**.

**Access modifiers** are Java keywords that define the level of access of each data field or method. There are four of them: **private**, **protected**, **public** and default or package access. Read about the differences between them.

An **Abstract Class** is an incomplete class that is missing the body of one or more of its methods. Or it can be a complete class but has the keyword abstract in its header. An abstract class cannot be instantiated. It is only used as a parent class for children classes that extend it and provide full implementation for all incomplete methods.

An **Interface** is the extreme case of an abstract class. It has only method headers and not method bodies. It cannot not have data fields. Basically, an interface is a way for defining how a certain behavior should be implemented by unrelated classes whose developers like to add to them this behavior.

***Shape***

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**-color\_: int**

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**+Shape (color: int)**

**+setColor(color: int): void**

**+getColor(): int**

**+*getArea*(): double**

**Circle**

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**-radius\_: double**

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**Circle (……………)**

**+getArea(): double**

**Rectangle**

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**-length\_: double**

**- width\_: double**

**¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯**

**Rectangle (………….)**

**+getArea(): double**

**Polymorphism** means having many forms. Java supports a few types of polymorphism. One of them is overriding. Also, a reference variable is polymorphic because it can reference objects of types different from its own, as long as those types are subclasses of its type. It is the object’s type, rather than the reference type that determines which method is called at runtime. The same applies to interfaces, where you can create a reference variable whose type is an interface. His variable can point to an object from a class that implements the interface.

**Student Activities**

**Activity 1: Inheritance and Abstract Classes**

You have the following UML inheritance tree and the accompanying code. The name of class ***Shape***in UML is italic because it is abstract. The same for method ***getArea ().***Javadoc comments were omitted to save space. It is required to:

1. Type this program, debug it, compile it and run it. Each class needs to be written in a separate file.
2. Write a new class Square that extends Shape and test it.
3. Add a method **toString()** to the three children classes that prints useful information about them.
4. Modify the UML inheritance tree after adding these changes.

public abstract class Shape {

private int color\_;

protected static int RED=1;

protected static int BLUE=2;

protected static int GREEN=3;

Shape (int color) {

setColor (color);

}

public void setColor

(int color) {

color\_ = color;

}

public int getColor () {

return color\_;

}

public *abstract double getArea* ();

}

public class TestShapes{

public static void main(String[] args) {

Shape [] shapes = new Shape [4];

shapes [0] = new Circle (1, Shape.RED);

shapes [1] = new Circle (2, Shape.GREEN);

shapes [2] = new Circle (3, Shape.BLUE);

shapes [3] = new Rectangle (1,2, Shape.RED);

for (int i = 0; i < 4; i++) {

System.out.println ("Shape number " + i + " is instance of "

+ shapes [i].getClass()

+ " and its area is "

+ shapes [i].getArea());

System.out.println (shapes [i].getClass().toString());

}

shapes [2] = new Rectangle (3,4, Shape.RED);

}

}

public class Circle extends Shape {

private double radius\_;

protected static double PI = 3.14;

Circle (double radius, int color) {

super (color);

radius\_ = radius;

}

public double getArea () {

return PI \* radius\_ \* radius\_;

}

}

public class Rectangle extends Shape {

private double length\_, width\_;

Rectangle (double length, double width, int color) {

super (color);

length\_ = length;

width\_ = width;

}

public double getArea () {

return length\_ \* width\_;

}

}

**Activity 2: Interfaces**

We have two attached files that implement a Queue. **QueueArray** uses arrays and the other **QueueArrayList** uses an **ArrayList**. It is required to:

1. Create an interface called **Queue** that defines what behavior any queue should have.
2. Create a small demo program that demonstrates the functionality of Queues and use a variable of the typ**e Queue** interface you created to hold a reference to a Queue object. See how you can change the type you are using fro**m QueueArray** t**o QueueArrayList** and vise versa with the demo still working.

**Activity 3: Interfaces and Abstract Classes**

**Objective:** to understand the differences between abstract classes and interfaces.

Suppose you have the following code:

**abstract** **class** Time {

**public** **abstract** **int** getMinutes();

}

**class** Days **extends** Time {

**private** **int** days;

**public** Days(**int** days) {

**this**.days = days;

}

**public** **int** getMinutes() {

**return** days \* 24 \* 60;

}

}

**class** HoursMinutes **extends** Time {

**private** **int** hours;

**private** **int** minutes;

**public** HoursMinutes(**int** hours, **int** minutes) {

**this**.hours = hours;

**this**.minutes = minutes;

}

**public** **int** getMinutes() {

**return** hours \* 60 + minutes;

}

}

And also the following code that is similar to the previous one but using interface instead of abstract class.

**interface** Time {

**int** getMinutes();

}

**class** Days **implements** Time {

**private** **final** **int** days;

**public** Days(**int** days) {

**this**.days = days;

}

**public** **int** getMinutes() {

**return** days \* 24 \* 60;

}

}

**class** HoursMinutes **implements** Time {

**private** **final** **int** hours;

**private** **final** **int** minutes;

**public** HoursMinutes(**int** hours, **int** minutes) {

**this**.hours = hours;

**this**.minutes = minutes;

}

**public** **int** getMinutes() {

**return** hours \* 60 + minutes;

}

}

**public** **class** Demo {

**public** **static** **void** main(String args[]) {

Time t1 = **new** Days(10);

Time t2 = **new** HoursMinutes(15, 59);

System.*out*.println(t1.getMinutes());

System.*out*.println(t2.getMinutes());

}

}

Now what's the difference between using abstract classes and interfaces in the example above if you want to add to Time a method: public int getSeconds();**?**

**Java Tools**

All these tools are free open source tools. Some of them are industrial tools with lots of features and others are lightweight tools.

1. **Eclipse.** Industrial tool. Has many additional plug-ins to add features to it. e Learning OOP concepts.
2. **NetBeans.** Industrial tool with many features and a GUI builder.
3. **JCreator.** Lightweight easy to use tool.
4. **DrJava.** The easiest and most simple tool. Consist of one file only. Very basic functionality. You should install a Java compiler and then it provides an interface for the compiler.
5. There are also other good tools.
6. You can also use Java compiler and runtime environment at the command line.

**Part 5 – Event-driven Programming and GUI in Java**

**The goal of this part is** to train students on (1) developing event-driven programs and (2) building GUI applications from scratch (piece by piece without a GUI builder).

**Important Concepts**

* **Event-driven programming** is writing programs that operate in response to events, generated by the user or by the environment. The most famous examples are applications with Graphical User Interfaces.
* An **event** is an action that takes place within a program, such as the clicking of a button
* An **event listener** is an object that automatically executes one of its methods when a specific event occurs.
* **Swing** (**import javax.swing.\*;**) is the Java library responsible of GUI.

**Student Activities**

**Activity 1: Building an Application with Basic GUI**

**Task 1:** Simulate the given program (Kilometer to mile conversion) and create a Celsius to Fahrenheit degree converter.

**Task** **2:** Modify the program so that it also does the opposite conversion (Fahrenheit to Celsius), i.e., the same program can convert either ways. This will need at least an extra button to do the opposite conversion.

**Task** **3:** Modify the program so that you use the following methods to enhance the program look and feel and functionality:

* **setToolTipText (String) // To give tips on the buttons**
* **setLocation (int x, int y) // To put the frame in the middle**
* **setIcon (ImageIcon img) // To put an image on the button**
* **....... *Any three more attributes***

**Activity 2: Building a GUI Application with More Widgets**

**Task 1**: Develop an application that gives the user five buttons and a label. The buttons have five labels of animal names: "Bird", "Cat", "Dog", "Pig" and "Rabbit". Whenever a button is pressed, the label area displays the image of the corresponding animal. The five animal pictures are included.

**Task 2**: Write a new version of the program that uses Radio Buttons instead of regular buttons. Read.