LAB #12 Classes II

# Objectives:

* Learn how to write and use the **toString( )** method
* Learn how to write and use the **equals** method
* Default instance and static variable values
* Default Constructors
* Copy Constructors
* Learn how to **overload** constructors and methods
* Learn how to use **this** keyword in calling overloaded constructors.

**1. Overriding (i.e., Redefining) the toString and equals methods of the Object class**

All classes in Java inherit from the Object class, directly or indirectly. The Object class has some basic methods like getClass(), public String toString(), public boolean equals(Object obj),.. etc. The default **public String toString()** method in **Object** returns the String representation of an object in the form: “class name @ hexadecimal hash code”. We can override the **toString()** method in our class to return meaningful String representation of the object that contains information about its instance variables.

**Note:** the **toString** method is invoked automatically whenever an object reference is used in a **String** context, i.e., when the reference is used in print, println, printf, or String concatenation operation:

Example:

**Student student1 = new Student(923400050, "Ahmad Muhsin", "COE", 4.0);**

**System.out.println(student1);**

**// equivalent to: System.out.println(student1.toString());**

**String str = "Student1 is " + student1;**

**// Equivalent to: String str = "Student1 is " + student1.toString();**

Similarly, the **public boolean equals(Object obj)** of the **Object** class compares object references (i.e. addresses) for equality: For any non-null reference values **x** and **y**, the call: **x.equals(y)** to the equals method of the **Object** class returns true if and only if, **x** and **y** refer to the same object, i.e., if **x == y** has the value true. It is recommended to override the **equals** method in our class to compare objects based on one or more instance variables.

Example: Overriding the toString and equals methods in the Student class:

**public String toString(){**

**return "ID: " + studentID + ", Name: " + studentName + ", Major: " + major + ", GPA: " + gpa;**

**}**

**// Two student objects are equal if theirIDs are equal**

**public boolean equals(Object obj){**

**if(obj == null)**

**return false;**

**else if(this.getClass() != obj.getClass())**

**return false;**

**else{**

**Student theStudent = (Student)obj;**

**return this.studentID == theStudent.studentID;**

**}**

**}**

**Example: Overiding the toString method in the Employee class**

In this case, we want the **toString** method to return a formated String. To do so, we use the **format** method of the **String** class:

public String toString(){

return String.format("ID: %d, Name: %s, Salary: %.2f Saudi Riyals", id, name, salary);

}

1. **Default instance and class variable values**

If you provide no explicit initialization to instance or class variables, they will be given default initial values, which are based only on the type of the variable. The table below shows the default initial values for each of the variable types. Local variables, except arrays, are not given default initial values. They must be initialized explicitly before they are used.

|  |  |
| --- | --- |
| Type | Default Value |
| boolean | false |
| byte | (byte) 0 |
| short | (short) 0 |
| int | 0 |
| long | 0L |
| char | ‘\u0000’ |
| float | 0.0f |
| double | 0.0d |
| object reference | null |

Example: Default values

class Dog{

private int size; // initialized to 0

private String name; // initialized to null

public String color; // initialized when the constructor is called

Dog(String color){

this.color = color;

}

public int getSize(){

return size;

}

public String getName(){

return name;

}

}

public class DefaultValues {

public static void main(String[] args) {

Dog dog1 = new Dog(“brown”);

System.out.println("Dog size is " + dog1.getSize());

System.out.println("Dog name is " + dog1.getName());

System.out.println("Dog color is " + dog1.color);

}

}

1. **Default Constructor**

If you declare a class with no constructors, the compiler will automatically create a default constructor for the class. A default constructor takes no parameters (it is a no-argument constructor) and has an empty body. Because the compiler will automatically generate a default constructor if you do not declare any constructors explicitly, all classes are guaranteed to have at least one constructor. The compiler gives default constructors the same access level as their class.

**Note:** If a class has at least one explicit constructor, the compiler will not create a default constructor for the class. To be able to use a no-argument constructor for such a class, one must define the no-argument constructor.

**Example1: Using Default Constructor**

**class Point{**

**private int x;**

**private int y;**

**public String toString(){**

**return "[x = " + x + " , y = " + y + "]";**

**}**

**public int getX(){**

**return x;**

**}**

**public int getY(){**

**return y;**

**}**

**public void setX(int x){**

**this.x = x;**

**}**

**public void setY(int y){**

**this.y = y;**

**}**

**}**

**public class NoArgumentConstructor01 {**

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

**Point p1 = new Point(); // calling the default constructor**

**System.out.println(p1);**

**p1.setX(3);**

**p1.setY(5);**

**System.out.println(p1);**

**}**

**}**

**Example2: Trying to use the default constructor in a class that has an explicit constructor causes a compile error**

**class Point{**

**private int x;**

**private int y;**

**public Point(int x, int y){**

**this.x = x;**

**this.y = y;**

**}**

**public String toString(){**

**return "[x = " + x + " , y = " + y + "]";**

**}**

**public int getX(){**

**return x;**

**}**

**public int getY(){**

**return y;**

**}**

**public void setX(int x){**

**this.x = x;**

**}**

**public void setY(int y){**

**this.y = y;**

**}**

**}**

**public class NoArgumentConstructor02 {**

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

**Point p1 = new Point(); // error: constructor Point in class Point cannot be**

**// applied to given types;**

**System.out.println(p1);**

**p1.setX(3);**

**p1.setY(5);**

**System.out.println(p1);**

**}**

**}**

# Copy Constructors

A copy constructor is a constructor with a single argument of the same type as the class. The copy constructor should create an object that is a separate object but with the instance variables set so that the object is an exact copy of the argument object.

Example:

**class Complex {**

**private double re, im;**

**// A normal parametrized constructor**

**Complex(double re, double im) {**

**this.re = re;**

**this.im = im;**

**}**

**// copy constructor**

**Complex(Complex c) {**

**re = c.re;**

**im = c.im;**

**}**

**public double getRealPart(){**

**return re;**

**}**

**public double getImaginaryPart(){**

**return im;**

**}**

**public void setRealPart(double re){**

**this.re = re;**

**}**

**public void setImaginaryPart(double im){**

**this.im = im;**

**}**

**// Overriding the toString of Object class**

**public String toString() {**

**return "(" + re + " + " + im + "i)";**

**}**

**// Overriding the equals method of Object class**

**public boolean equals(Object obj){**

**if(obj == null)**

**return false;**

**else if(getClass() != obj.getClass())**

**return false;**

**else{**

**Complex cmplx = (Complex) obj;**

**return re == cmplx.re && im == cmplx.im;**

**}**

**}**

**}**

**public class ComplexDriver {**

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

**Complex c1 = new Complex(10, 15);**

**// The following involves a copy constructor call**

**Complex c2 = new Complex(c1);**

**// Note that the following does not involve a copy constructor call as**

**// non-primitive variables are just references.**

**Complex c3 = c2;**

**System.out.println("Before modifying c2:");**

**System.out.println("c1 is " + c1);**

**System.out.println("c2 is " + c2);**

**System.out.println("c3 is " + c3);**

**//Modify c2**

**c2.setRealPart(8.0);**

**c2.setImaginaryPart(3.5);**

**System.out.println("\nAfter modifying c2:");**

**System.out.println("c1 is " + c1);**

**System.out.println("c2 is " + c2);**

**System.out.println("c3 is " + c3);**

**}**

**}**

Output:

Before modifying c2:

c1 is (10.0 + 15.0i)

c2 is (10.0 + 15.0i)

c3 is (10.0 + 15.0i)

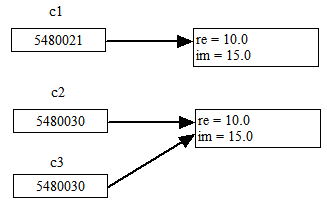
After modifying c2:

c1 is (10.0 + 15.0i)

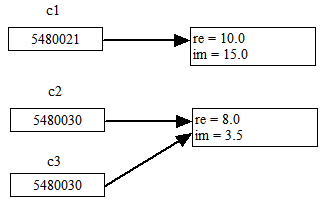
c2 is (8.0 + 3.5i)

c3 is (8.0 + 3.5i)

Note: The state of the objects before c2 is modified:



The state of the objects after c2 is modified:



# Overloading methods and constructors

Overloading means having more than one constructor in a class or having more than one method with the same name in a class. In the case of constructor, the purpose of overloading is to allow the user to have as many options as possible when creating an object of the class, thus making the class more flexible to use. In the case of methods, overloading allows the same name to be used for methods that perform similar tasks.

The condition for overloading is that the overloaded methods and/or constructors must have different signatures. The **number**, the **type** and the **order** of its parameters determine the signature of a constructor or a method.

# 

# Using *this* keyword to call overloaded constructors

# The call *this(argumentList)* is used to call an overloaded constructor.

The advantages of using ***this*** as can be seen from the given Employee class example is that programs become shorter and that we do not have to think of different names for the parameters of constructors and methods.

**Note:** If **this** is used to call another constructor, the call must be in the first line of the calling constructor.

# Example# 3

The following example implements the Employee class. Notice how the constructors are overloaded. Also notice the overloading of the deductions methods.

**public class Employee {  
 private int id;   
 private String name;   
 private double salary;  
 public Employee(int id, String name, double salary)throws IllegalArgumentException {  
 if(salary < 3000)  
 throw new IllegalArgumentException("salary < 3000");  
 this.id = id;  
 this.name = name;  
 this.salary = salary;  
 }  
   
 public Employee(String name, int id, double salary) {  
 this(id,name,salary);  
 }  
   
 public Employee(int id, String name) {  
 this(id, name, 3000.0); // Use minimum salary  
 }  
 public Employee(String name, int id) {  
 this(id, name, 3000.0); // Use minimum salary  
 }  
   
 public void setSalary(double salary)throws IllegalArgumentException {  
 if(salary < 3000.0)  
 throw new IllegalArgumentException("salary < 3000.0");   
 this.salary = salary;  
 }  
   
 public int getID() {  
 return id;  
 }  
   
 public String getName(){  
 return name;  
 }  
   
 public double getSalary(){  
 return salary;  
 }  
   
 public void deductions(double telephoneBills) throws IllegalArgumentException {   
 if(telephoneBills <= 0)  
 throw new IllegalArgumentException("Invalid telephone bill amount");  
 else  
 salary -= telephoneBills;  
 }**

**public void deductions(double telephoneBills, double medicalBills)   
 throws IllegalArgumentException {  
 if(telephoneBills <= 0 || medicalBills <= 0)  
 throw new IllegalArgumentException("Invalid bill amount");  
 else  
 salary -= (telephoneBills + medicalBills);  
 }  
   
 public void raiseSalary(double percentIncrease) throws IllegalArgumentException {  
 if(percentIncrease <= 0)  
 throw new IllegalArgumentException("Invalid percentIncrease");  
 else  
 salary += salary \* percentIncrease/100;  
 }  
   
 public String toString() {  
 return "\nID: " + id + "\nName: " + name+"\nSalary: "+salary + " SAR";  
 }  
  
 public boolean equals(Object obj){  
 if(obj == null)  
 return false;  
 else if(this.getClass() != obj.getClass())  
 return false;  
 else{  
 Employee employee = (Employee) obj;  
 return this.id == employee.id;  
 }  
 }  
}**

**Note:** In case of invalid parameters, the Constructors and methods **deductions** and **raiseSalary** throw exceptions rather than display error messages using:

System.out.println(“Error: Invalid parameter”);

This is good programming practice; the decision on what is to be done in case of error should be done by the driver program. To enable this, the driver program should call these methods in a try-catch block. Example:

double deductAmount = scanner.nextDouble( );

try{

emp1.deductions(deductAmount);

} catch(IllegalArgumentException e){

System.out.println(e);

// …

}

## // The following shows how the overloaded constructors and methods of Employee class may be used:

## import java.util.Scanner; import java.util.InputMismatchException;

## public class EmployeeDriver { public static void main(String[] args){ Scanner scanner = new Scanner(System.in); int number; String name; double salary; System.out.print("Enter Name for Employee 1: "); name = scanner.nextLine(); System.out.print("Enter ID Number for Employee 1: "); number = scanner.nextInt(); try{ System.out.print("Enter Salary for Employee 1 (>= 3000): "); salary = scanner.nextDouble(); scanner.nextLine(); // skip new line character //any of the following constructors be used to create the object Employee emp1 = new Employee (number, name, salary); // or Employee emp1 = new Employee (name, number, salary); System.out.print("\nEnter Name for Employee 2: "); name = scanner.nextLine(); System.out.print("Enter ID Number for Employee 2: "); number = scanner.nextInt(); //if we do not know the salary, we can use one of the following constructors Employee emp2 = new Employee (number, name); //or Employee emp2 = new Employee (name, number); emp2.setSalary(emp1.getSalary()); // set the salary of employee2 to be equal to that of employee1 emp1.deductions(50.0); emp2.deductions(60.0, 40.0); System.out.println(emp1); System.out.println(emp2); } catch(InputMismatchException | IllegalArgumentException e){ System.out.println(e); } } }

## Lab Tasks

**Task#1:**

Modify the given **Box** class file as follows:

* Add a one-argument constructor, which receives only the length. It then calls the three-argument constructor supplying this length for each parameter: length, width and height (i.e it forms a cube)
* Add a copy constructor
* Add a **toString** method
* Add an **equals** method: Two box objects are equal if their corresponding dimensions are equal.
* Add appropriate set- and get-methods [Lab11 task].

The class **BoxDriver** should be modified to create two Box objects using each of the two Constructors and then print them using **toString** and compare them using the **equals** method. It must also create a copy of one of the Box objects using the copy constructor. Your main method must handle both InputMismatchException and IllegalArgumentException.

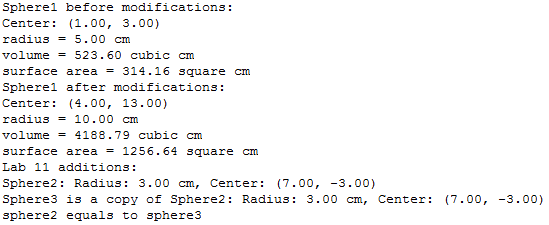
Sample program runs:

|  |
| --- |
|  |
|  |
|  |
|  |

**Task#2:**

1. Modify the **Sphere** class that you implemented in Lab11 to include the **toString** and the **equals** method, also include a copy constructor.
2. Modify the **SphereDriver** class you wrote in Lab11 to test the two constructors, the **equals** method, and the **toString** method.

Sample program run:



**Task#3:**

Modify the **Student** class you wrote in Lab11 to have the following fields, constructors and methods:

**Fields:**

private String name;

private double totalScore;

private int numberOfQuizzes;

private String quizlist = "";

**Constructors:**

public Student(String name, double score) throws IllegalArgumentException

public Student(double score, String name) // calls above constructor

public Student(String name)

public Student(Student student) // Copy constructor

**Methods:**

public String getName()

public double getAverage( ) throws UnsupportedOperationException *//this should throw the exception if no*

*// quiz has been taken.*

public double getTotalScore( )

public void addQuiz(double score) throws IllegalArgumentException

public String toString( )

public boolean equals(Object obj) // comparison based on name

Write an application **TestStudent** that prompts for and reads a student name. It then creates a Student object. It then prompts for and reads the number of quizzes **n** that the student has taken. It then prompts for and reads the scores of the student in **n** quizzes and adds each to the ***totalScore*** of the student using ***addQuiz( )*** method. Finally, the application prints the student object using the toString() method, and it also prints his quiz average.

The **main** method must handle java.util.InputMismatchException, java.lang.IllegalArgumentException and java.lang.UnsupportedOperationException.

**Sample program runs:**

|  |
| --- |
|  |
|  |
|  |