**Final Exam Instructions**

**OBJECT-ORIENTED PROG**

* This is a take-home exam. You can use any resources that are available for you to finish this exam, except
  + Outsourcing the exam to any person or to any third party websites
  + Copying from other students work
  + Copying direct quotes from the books or internet
* Do not lose your opportunity to learn while working on the exam. Understand the concept and write answers on your own.
* Usually, in life, we have several choices. Unfortunately, you don’t have any choice on this exam. You have to answer all the questions and each part of the problem.
* All the topics on this exam were discussed in class . So, you cannot claim that the questions are out of the syllabus!
* Refer to Microsoft Word tutorials for proper formatting
* Points will be deducted for grammatical and spelling mistakes
* No two brains think alike unless you are soulmates. Definitely your answers will not be same as other students.
* Read the code of academic integrity before you start the exam. <https://www.nwmissouri.edu/policies/academics/Academic-Integrity.pdf>
* Push your source code to GitHub and provide your GitHub link at the end of the document and in the comment section.
* Don’t use examples that already explained in class or worksheets.
* Provide the input and output screenshots for every program.

**Final Exam OBJECT-ORIENTED PROG 01FA20 150 pts**

1. (20-Points) Define the terms abstract classes and interfaces. What are the similarities and differences between abstract classes and interfaces? Why interfaces are preferred over abstract classes? Explain and demonstrate with examples.

Ans) **Abstract class**: A restricted class that cannot be used to construct objects is an abstract class (to access it, it must be inherited from another class). It is only possible to use an abstract method in an abstract class, and it does not have a body. The subclass provides the body (inherited from).

**Interfaces:** An interface is a type in the Java programming language, much as a class is a type. An interface, like a class, describes methods. An interface, unlike a class, never implements methods; instead, classes that implement the interface implement the interface's methods.

**Differences and Similarities:**

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Abstract class** | **Interface** |
| Class type | An abstract class has protected and public abstract methods. | An interface can have only public abstract methods. |
| Constructor | An abstract class can declare constructors. | An interface cannot declare constructors. |
| Inheritance/ Implementation | The class can inherit only one Abstract Class. | A Class can implement multiple interfaces. |
| Access Modifiers | Abstract Class can have an access modifier. | The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier. |
| Limit of Extensions | It can extend only one class or one abstract class at a time. | It can extend any number of interfaces. |

**Interfaces are preferred over Abstract classes:** because an abstract class allows you to create functionality that subclasses can enforce or override, while an interface allows you to state but not implement functionality. Although a class can only extend one abstract class, it can implement multiple interface.

**Example Interface01**: Here Interface01 class is a subclass which is implementing the engine interface ,later interface methods are implemented then we are creating object for Interface01 class and at last we are calling methods and passing the parameters.

**DriverClass.java:**

package question01;

/\*\*

\*

\* @author S542312

\*/

public class Interface01 implements Engine {

int speed;

int gear;

public void speedUp(int a) {

this.speed=a;

System.out.println("speed"+speed);

}

public void changeGear(int a) {

this.gear=a;

System.out.println("gear"+gear);

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

Interface01 obj=new Interface01();

obj.changeGear(5);

obj.speedUp(120);

}

}

**Engine.java:**

package question01;

/\*\*

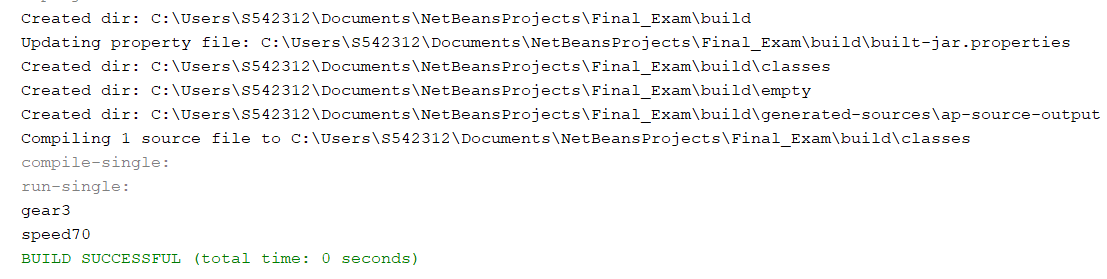
\*

\* @author S542312

\*/

interface Engine {

}



**Example02 Abstract class:** Starting in employee details abstract keyword is used to create abstract class and later abstract method is created, next hr details were created using the abstract method implementation.

**Employee Details:**

package question01;

/\*\*

\*

\* @author S542312

\*/

public abstract class EmployeeDetails {

private String name;

private int emp\_ID;

public void commonEmpDetaills()

{

System.out.println("Name"+name);

System.out.println("emp\_ID"+emp\_ID);

}

public abstract void confidentialDetails(int m,String r);

}

**HR Details:**

package question01;

/\*\*

\*

\* @author S542312

\*/

public class HR extends EmployeeDetails {

private int salary;

private String performance;

@Override

public void confidentialDetails(int s,String p) {

this.salary=s;

this.performance=p;

System.out.println("Salary = "+salary);

System.out.println("Performance = "+performance);

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

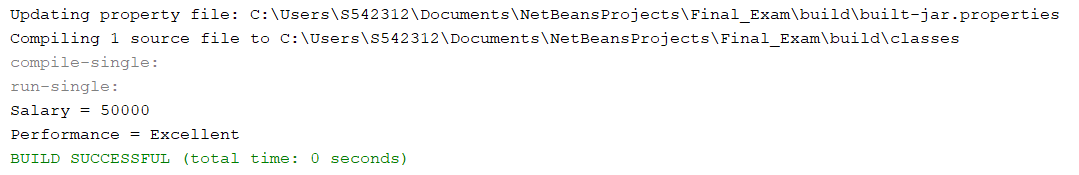
// TODO code application logic here

HR hr =new HR();

hr.confidentialDetails(50000,"Excellent");

}

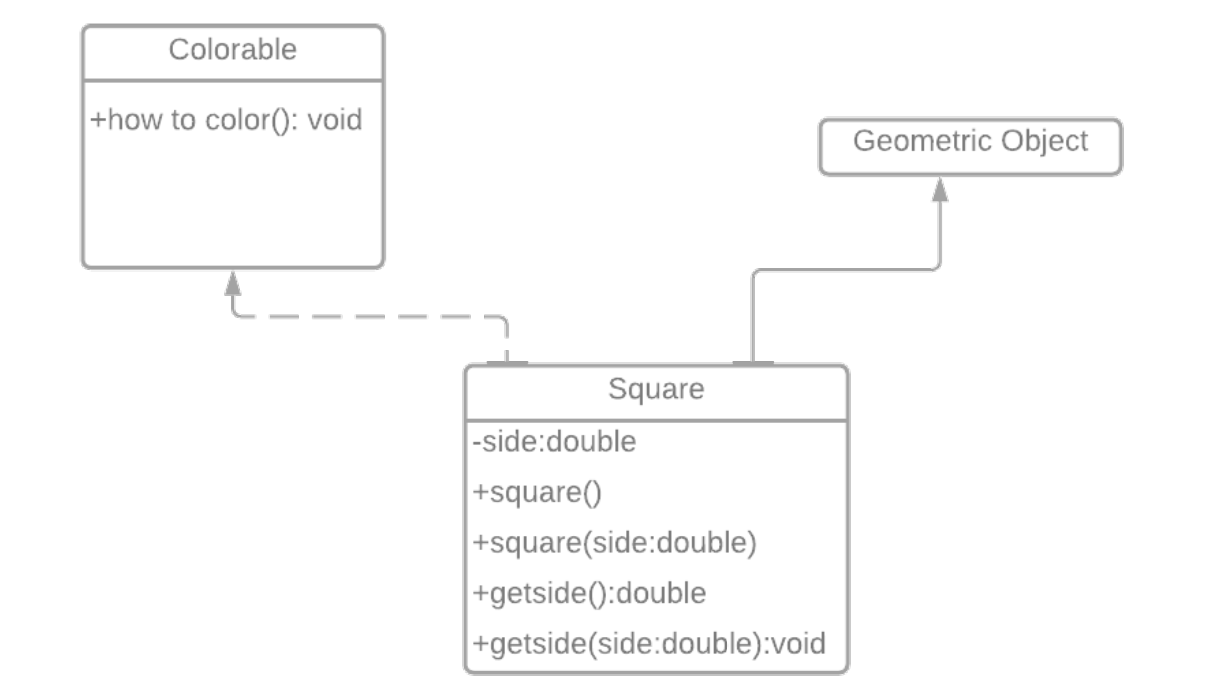
}



1. (10-Points) Design an interface named Colorable with a void method named howToColor(). Every class of a colorable object must implement the Colorable interface. Design a class named Square that extends GeometricObject and implements Colorable Implement howToColor to display the message Color all four sides.

Draw a UML diagram that involves Colorable, Square, and GeometricObject. Write a test program that creates an array of five GeometricObjects. For each object in the array, display its area and invoke its howToColor method if it is colorable.

Ans) **UML Diagram:**



**Test Program:**

**Interface Colorable:**

package question02;

/\*\*

\*

\* @author S542312

\*/

public interface Colorable {

public void howToColor();

}

**Class Square:**

package question02;

/\*\*

\*

\* @author S542312

\*/

class Square extends GeometricObject implements Colorable {

public double side;

public Square() {

side=0.0;

}

public Square(double side) {

this.side = side;

}

public double getSide() {

return side;

}

public void setSide(double side) {

this.side = side;

}

@Override

public double getArea() {

double area = side \* side;

return area;

}

@Override

public void howToColor() {

System.out.println("Color all four sides.");

}

}

**Class GeometricObject:**

package question02;

/\*\*

\*

\* @author S542312

\*/

class Circle extends GeometricObject{

private double radius;

public Circle() {

}

public Circle(double radius) {

this.radius = radius;

}

public double getRadius() {

return radius;

}

public void setRadius(double radius) {

this.radius = radius;

}

@Override

public double getArea() {

return radius \* radius \* Math.PI;

}

public double getDiameter() {

return 2 \* radius;

}

public double getPerimeter() {

return 2 \* radius \* Math.PI;

}

/\* Print the circle info \*/

public void printCircle() {

System.out.println("The circle is created " + getDateCreated()

+ " and the radius is " + radius);

}

}

abstract class GeometricObject {

private String color = "white";

private boolean filled;

private final java.util.Date dateCreated;

protected GeometricObject() {

dateCreated = new java.util.Date();

}

protected GeometricObject(String color, boolean filled) {

dateCreated = new java.util.Date();

this.color = color;

this.filled = filled;

}

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}

public boolean isFilled() {

return filled;

}

public void setFilled(boolean filled) {

this.filled = filled;

}

public java.util.Date getDateCreated() {

return dateCreated;

}

@Override

public String toString() {

return "created on " + dateCreated + "\ncolor: " + color

+ " and filled: " + filled;

}

public abstract double getArea();

}

**Class Rectangle:**

package question02;

/\*\*

\*

\* @author S542312

\*/

class Rectangle extends GeometricObject{

private double width;

private double height;

public Rectangle() {

width=0.0;

height=0.0;

}

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

public double getWidth() {

return width;

}

public void setWidth(double width) {

this.width = width;

}

public double getHeight() {

return height;

}

public void setHeight(double height) {

this.height = height;

}

@Override

public double getArea() {

return width \* height;

}

public double getPerimeter() {

return 2 \* (width + height);

}

}

**Driver Class:**

package question02;

/\*\*

\*

\* @author S542312

\*/

public class main {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

GeometricObject[] obj = {new Square(4), new Circle(6), new Square(3), new Rectangle(2, 4), new Square(5.5)};

for (int i = 0; i < obj.length; i++) {

System.out.println("Area is " + obj[i].getArea());

if (obj[i] instanceof Colorable) {

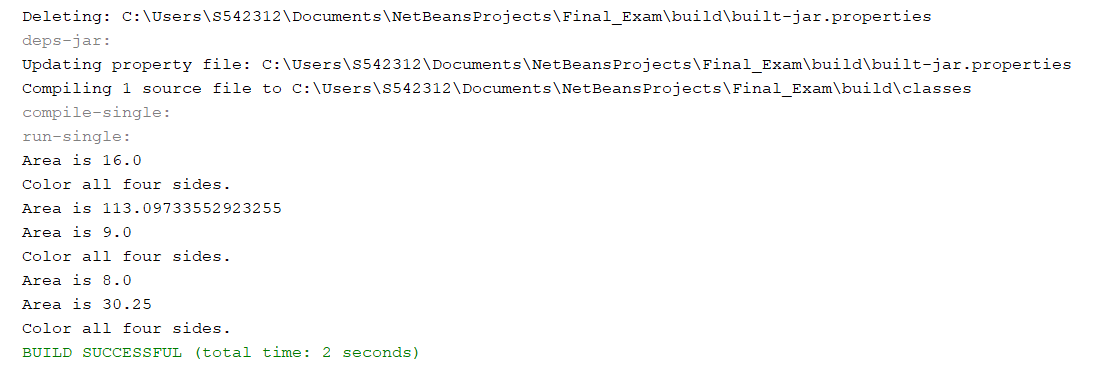
((Colorable) obj[i]).howToColor();

}

}

}

}



1. (10-Points) What is casting? What are different types of casting? Explain and demonstrate with examples.

Ans) **Casting:** In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

There are two types of casting in java, they are:

1. Widening Casting
2. Narrowing Casting

**Widening Casting:** Widening casting is done automatically when passing a smaller size type to a larger size type.

byte -> short -> char -> int -> long -> float -> double

**Narrowing Casting:** Narrowing casting must be done manually by placing the type in parentheses in front of the value.

double -> float -> long -> int -> char -> short -> byte

**Wedening Example01:** In this following example we have been converted integer to double value,

package question03;

/\*\*

\*

\* @author S542312

\*/

public class widening01 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

int number = 10;

System.out.println("The integer value: " + number);

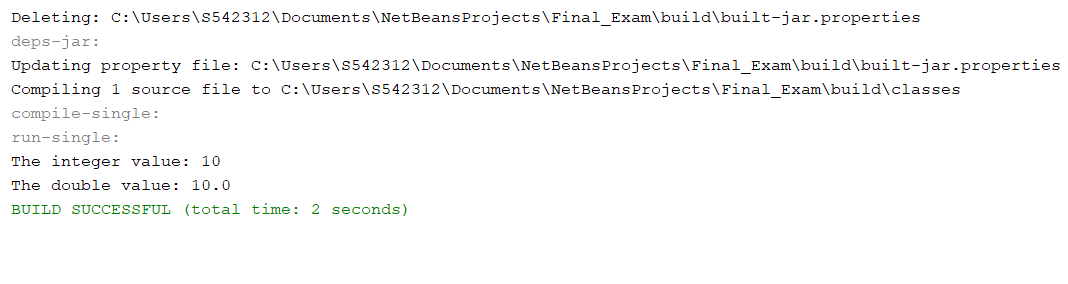
// convert into double type

double data = number;

System.out.println("The double value: " + data);

}

}



**Wedening Example02:** In the following example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

package question03;

/\*\*

\*

\* @author S542312

\*/

public class widening02 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

int a = 10;

//automatically converts the integer type into long type

long b = a;

//automatically converts the long type into float type

float c = b;

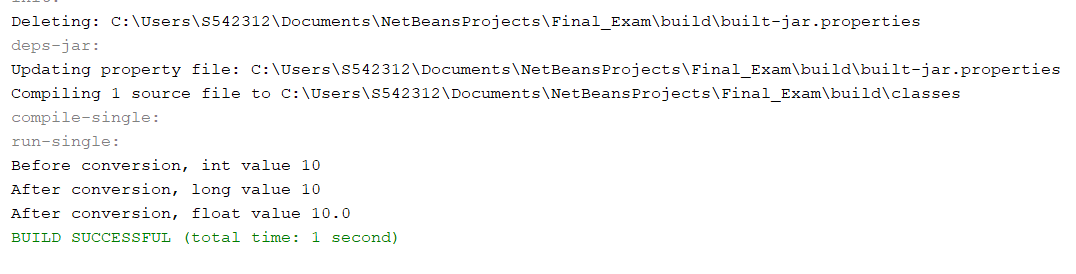
System.out.println("Before conversion, int value "+a);

System.out.println("After conversion, long value "+b);

System.out.println("After conversion, float value "+c);

}

}



**Narrowing Example01:** In this narrowing example, we have converted the double type to integer value.

package question03;

/\*\*

\*

\* @author S542312

\*/

public class narrowing01 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

// create double type variable

double number = 10.99;

System.out.println("The double value: " + number);

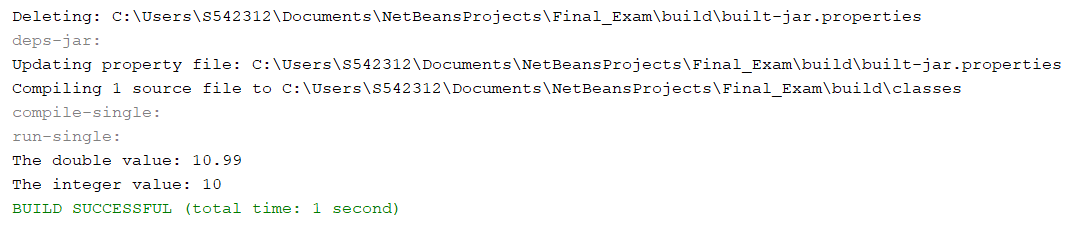
// convert into int type

int data = (int)number;

System.out.println("The integer value: " + data);

}

}



**Narrowing Example02:** In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

package question03;

/\*\*

\*

\* @author S542312

\*/

public class narrowing02 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

double x = 166.66;

//converting double data type into long data type

long y = (long)x;

//converting long data type into int data type

int z = (int)y;

System.out.println("Before conversion: "+x);

//fractional part lost

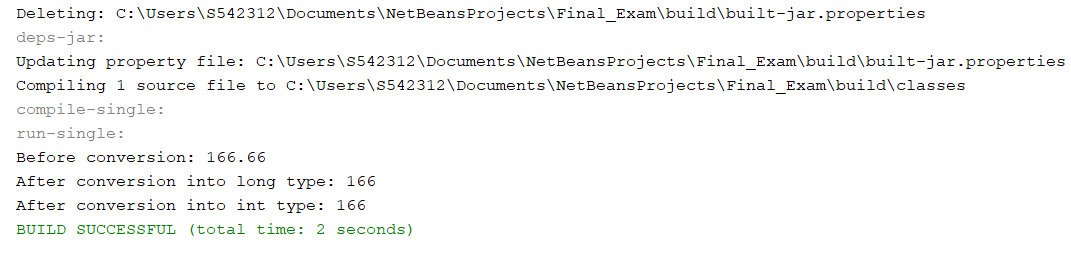
System.out.println("After conversion into long type: "+y);

//fractional part lost

System.out.println("After conversion into int type: "+z);

}

}



1. (15-Points) Suppose that Fruit, Apple, Orange, GoldenDelicious, and McIntosh are defined in the following inheritance hierarchy:

Fruit

Orange

Apple

GoldenDelicious

McIntosh

Assume that the following code is given:

Fruit fruit = new GoldenDelicious();

Orange orange = new Orange();

Answer the following questions and explain why these Statements are legal or illegal.

1. Is fruit instanceof Fruit?

Ans) The class GoldenDelicious is represented by the object fruit.

The Apple Class has a subclass called GoldenDelicious. The Fruit class has a subclass called Apple.

As a result, the class Golden Delicious is a subclass of the Fruit class. A sub-class instance is the same as its super-class instance.

Hence, fruit is instanceof Fruit.

1. Is fruit instanceof Orange?

Ans) The object fruit contains an instance of the class Golden Delicious.

Orange is not a super-class of the class Golden Delicious.

Hence, fruit is not instanceof Orange.

1. Is fruit instanceof Apple?

Ans) The class Golden Delicious is represented by the item fruit.

The Apple Class has a subclass called Golden Delicious.

A sub-class instance is the same as its super-class instance.

Hence, fruit is instanceof Apple.

1. Is fruit instanceof GoldenDelicious?

Ans) The class Golden Delicious is represented by the item fruit.

Hence, fruit is instanceof Golden Delicious.

1. Is fruit instanceof McIntosh?

Ans) The class GoldenDelicious is represented by the object fruit. The class McIntosh is not a super-class in the GoldenDelicious class.

Hence, fruit is not instanceof McIntosh.

1. Is orange instanceof Orange?

Ans) The object orange contains an instance of the class Orange.

Hence, orange is instanceof orange.

1. Is orange instanceof Fruit?

Ans) GoldenDelicious is a sub-class of the Apple Class, and the object orange includes an instance of the class Orange.

Hence, orange is instanceof Fruit.

1. Is orange instanceof Apple?

Ans) The class Orange is represented by the object orange. The class Orange does not have a super-class named Apple.

Hence, orange is not an instanceof Apple.

i. Suppose the method makeAppleCider is defined in the Apple class. Can fruit invoke this method? Can orange invoke this method?

Ans) Fruit should call the makeAppleCider method since it is an instance of the class Apple.

Since orange is not an instance of the class Apple, it is unable to use the makeAppleCider process.

1. Suppose the method makeOrangeJuice is defined in the Orange class. Can orange invoke this method? Can fruit invoke this method?

Ans) Since orange is an instance of the class Orange, it can call the makeOrangeJuice process.

Fruit cannot call the makeOrangeJuice method because it is not an instance of the class Orange.

1. Is the statement Orange p = new Apple() legal?

Ans) Since the class Apple is not a subclass of the class Orange, an instance of the class Apple cannot be allocated to an Orange object.

Hence, the statement Orange p = new Apple() is illegal.

1. Is the statement McIntosh p = new Apple() legal?

Ans) Since the class Mcintosh is a sub-class of Apple, assigning an instance of the super-class to an object of its sub-class results in an error.

Hence, the statement Mcintosh p=new Apple() is illegal.

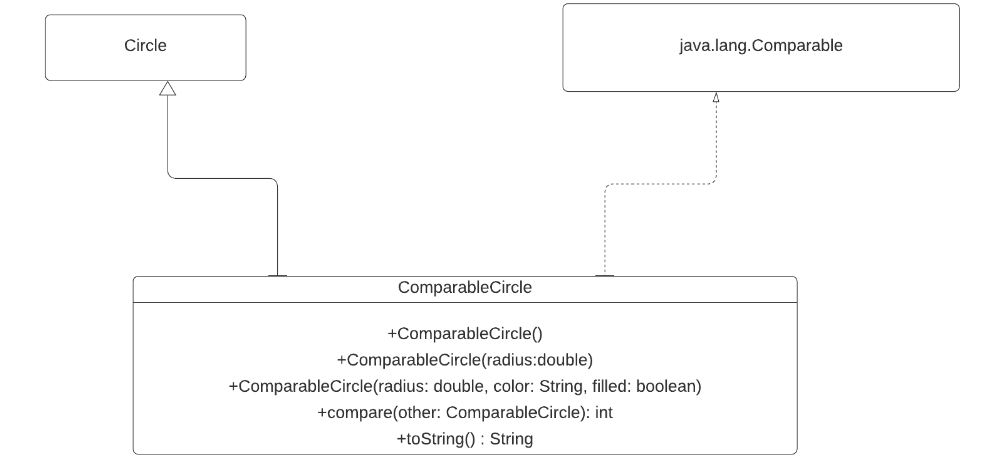
1. Is the statement Apple p = new McIntosh() legal?

Ans) The class Apple is the super-class of the class McIntosh, and Java allows you to assign a sub-class instance to a super-class object.

Hence, the statement Apple p= new Mcintosh() is legal.

1. (10-Points) Define a class named ComparableCircle that extends Circle and implements Comparable. Draw the UML diagram and implement the compareTo method to compare the circles on the basis of area. Write a test class to find the larger of two instances of ComparableCircle objects.

Ans) **UML Diagram:**



**Program:**

**Circle.java:**

package question05;

import question02.GeometricObject;

/\*\*

\*

\* @author S542312

\*/

public class Circle

extends GeometricObject {

private double radius;

public Circle() {

}

public Circle(double radius) {

this.radius = radius;

}

public Circle(double radius,

String color, boolean filled) {

this.radius = radius;

setColor(color);

setFilled(filled);

}

/\*\* Return radius \*/

public double getRadius() {

return radius;

}

/\*\* Set a new radius \*/

public void setRadius(double radius) {

this.radius = radius;

}

@Override /\*\* Return area \*/

public double getArea() {

return radius \* radius \* Math.PI;

}

/\*\* Return diameter \*/

public double getDiameter() {

return 2 \* radius;

}

@Override /\*\* Return perimeter \*/

public double getPerimeter() {

return 2 \* radius \* Math.PI;

}

@Override /\*\* Implement the toString method in GeometricObject \*/

public String toString() {

return super.toString() + "\nDate created: " + getDateCreated() +

"\nRadius: " + radius;

}

}

**ComparableCircle:**

package question05;

/\*\*

\*

\* @author S542312

\*/

public class ComparableCircle extends Circle

implements Comparable<ComparableCircle> {

public ComparableCircle() {

}

/\*\* Construct a CoparableCircle with specified radius \*/

public ComparableCircle(double radius) {

super(radius);

}

/\*\* Construct a CoparableCircle with specified properties \*/

public ComparableCircle(double radius, String color, boolean filled) {

super(radius, color, filled);

}

@Override // Implement the compareTo method defined in Comparable

public int compareTo(ComparableCircle o) {

if (getArea() > o.getArea())

return 1;

else if (getArea() < o.getArea())

return -1;

else

return 0;

}

@Override // Implement the toString method defined in Circle

public String toString() {

return super.toString() + "\nArea: " + getArea();

}

}

**Main Class:**

package question05;

/\*\*

\*

\* @author S542312

\*/

public class TestClass {

/\*\* Main method \*/

public static void main(String[] args) {

// Create two instances of ComparableCircle objects

ComparableCircle comparableCircle1 = new ComparableCircle(12.5);

ComparableCircle comparableCircle2 = new ComparableCircle(18.3);

// Display comparableCircles

System.out.println("\nComparableCircle1:");

System.out.println(comparableCircle1);

System.out.println("\nComparableCircle2:");

System.out.println(comparableCircle2);

// Find and display the larger of the two ComparableCircle objects

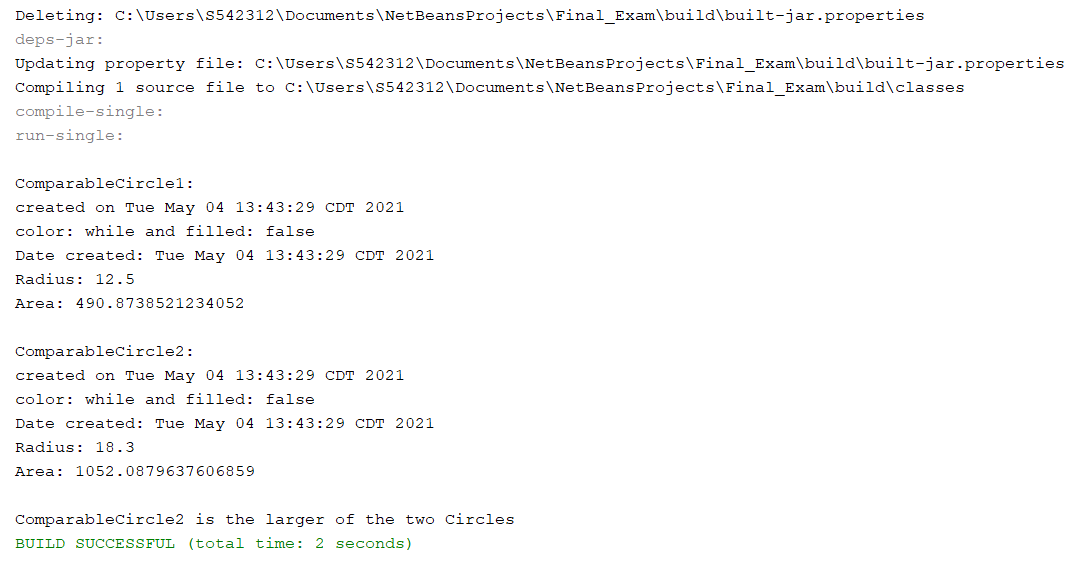
System.out.println((comparableCircle1.compareTo(comparableCircle2) == 1

? "\nComparableCircle1 " : "\nComparableCircle2 ") +

"is the larger of the two Circles");

}

}



1. (15-Points) What is an exception? What are checked and unchecked exceptions? Explain and demonstrate with examples.

Ans) **Exception:** An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. This block of code is called an exception handler.

There are two types of exceptions, they are:

1. Checked Exceptions
2. Unchecked Exceptions
3. **Checked Exceptions: t**heseare the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

Examples: IOException

ClassNotFoundException

FileNotFoundException

1. **Unchecked Exceptions:** are the exceptions that are not checked at compiled time. In java, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

Examples: NullPointerException

ArrayIndexOutOfBoundsException

ArithmeticException

**Example01:** The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

**Unchecked Exception:**

package question06;

import java.io.\*;

/\*\*

\*

\* @author S542312

\*/

public class UncheckedExcep {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

try {

int m[] = new int[2];

System.out.println("Access element three :" + m[3]);

} catch (ArrayIndexOutOfBoundsException e) {

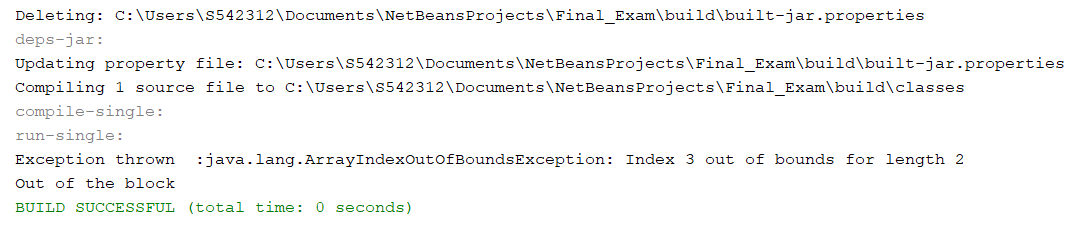
System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}



**Example02:** Now, when you run this program, because the input.txt file  does not exist, the exception is thrown as shown in the below. As you can see the message is showing the cause of the problem. The root cause of the problem is that the file does not exist.

**FileNotFoundExceptionExample:**

package question06;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

/\*\*

\*

\* @author S542312

\*/

public class FileNotFoundExceptionExample

{

public void checkFileNotFound()

{

try

{

FileInputStream in = new FileInputStream("input.txt");

System.out.println("This is not printed");

}

catch (FileNotFoundException fileNotFoundException)

{

fileNotFoundException.printStackTrace();

}

}

public static void main(String[] args)

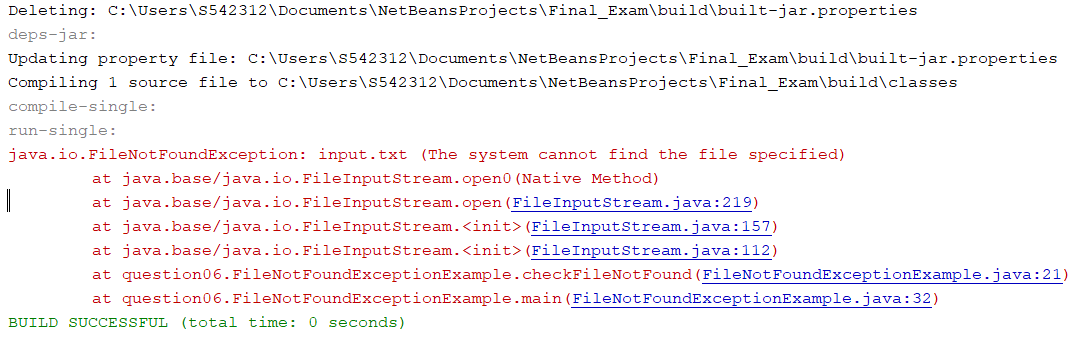
{

FileNotFoundExceptionExample example = new FileNotFoundExceptionExample();

example.checkFileNotFound();

}

}



1. (10-Points) Write a program that meets the following requirements:

* Creates an array with 100 randomly chosen integers.
* Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message Out of Bounds.

Ans)

**Program:**

package question07;

import java.util.Random;

import java.util.Scanner;

/\*\*

\*

\* @author S542312

\*/

public class RandomIntegers {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Random r = new Random();

int[] arrays = new int[100];

for (int i = 0; i < arrays.length; i++) {

arrays[i] = r.nextInt(100);

}

System.out.print("Enter index value 0 - " + (arrays.length - 1) + " : ");

int index = sc.nextInt();

if (index < 0 || index > arrays.length - 1) {

System.out.println("Out of Bounds");

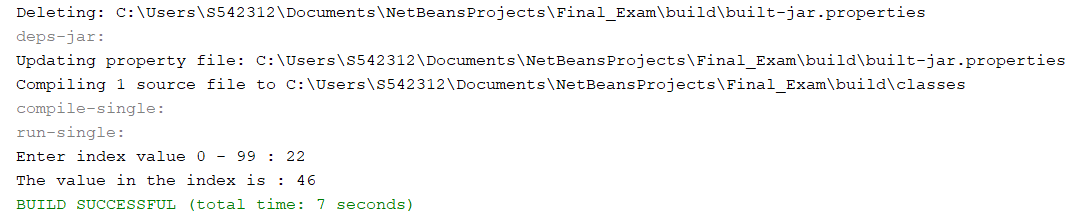
} else {

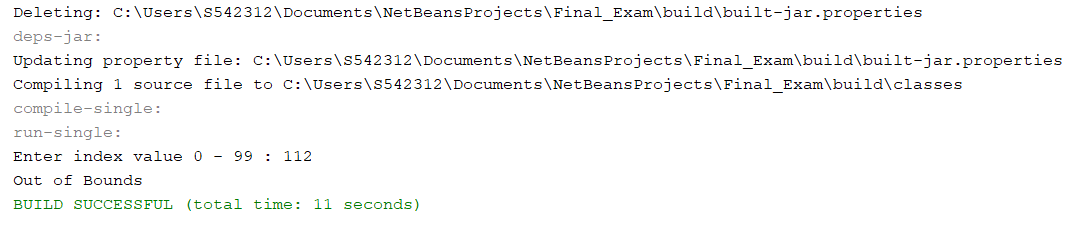
System.out.println("The value in the index is : " + arrays[index]);

}

}

}





1. (10-Points) What is the purpose of declaring exceptions? How do you declare an exception, and where? Can you declare multiple exceptions in a method header? Explain and demonstrate with examples.

Ans) The purpose of declaring exceptions is to tell the Java runtime system what can go wrong. You declare an exception using the throws keyword in the method declaration. You can declare multiple exceptions, separated by commas.

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature. You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

**Yes**, declare multiple exceptions in a method header are possible. If the method declares multiple exceptions, add list of the exceptions, separated by commas, after throws.

**Example01:** According to the precedence compiler check **number[10]=80/0**from right to left. That’s why 80/0 to throw ArithmeticException object and the handler of this exception executes Zero cannot divide any number.

we can combine two Exception using the | operator and either one of them executes according to the exception occurs.

**MultipleException:**

package question08;

/\*\*

\*

\* @author S542312

\*/

import java.io.\*;

class MultipleException {

public static void main(String[] args)

{

try {

int number[] = new int[10];

number[10] = 80 / 0;

}

catch (ArithmeticException e) {

System.out.println(

"Zero cannot divide any number");

}

catch (ArrayIndexOutOfBoundsException e) {

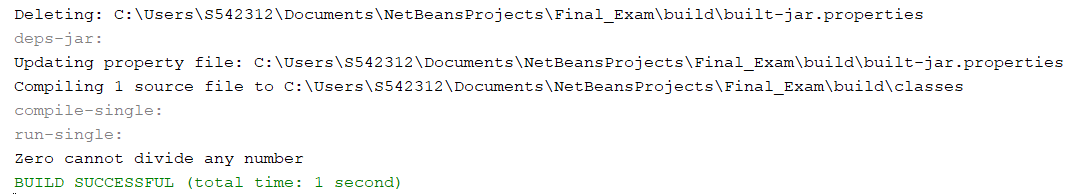
System.out.println(

"Index out of size of the array");

}

}

}



**Example02:** A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler, that is, the bytecode has no replication of exception handlers.

**Test.java:**

package question08;

import java.util.Scanner;

/\*\*

\*

\* @author S542312

\*/

public class Test

{

public static void main(String args[])

{

Scanner scn = new Scanner(System.in);

try

{

int n = Integer.parseInt(scn.nextLine());

if (99%n == 0)

System.out.println(n + " is a factor of 99");

}

catch (NumberFormatException | ArithmeticException ex)

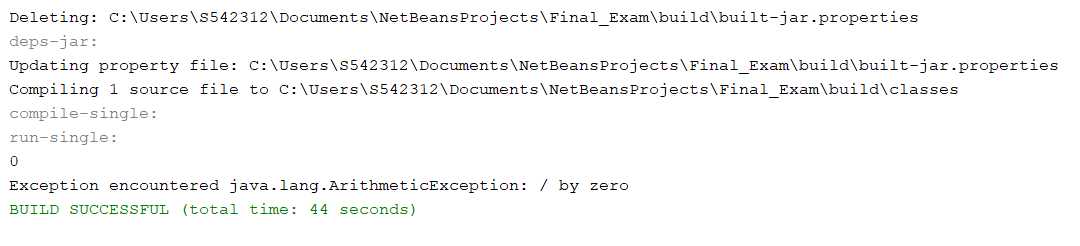
{

System.out.println("Exception encountered " + ex);

}

}

}



1. (10-Points) What is the keyword throw used for? What is the keyword throws used for? Can you throw multiple exceptions in one throw statement? Explain with examples.

Ans) The **throws keyword** is used to declare which exceptions can be thrown from a method, while the throw keyword is used to explicitly throw an exception within a method or block of code. The throws keyword is used in a method signature and declares which exceptions can be thrown from a method.

In our code throws more than one exception, we can choose if we want to, use a separate try block for each statement that could throw an exception or use one try block for multiple statements that might throw multiple exceptions.

**Example01:** Java throws an Arithmetic exception when a calculation attempt is done to divide a number by zero, where the zero is an integer. Since I have divided 14 by 0, where 0 is an integer, Java throws the above exception.

**Throws.java:**

package question09;

/\*\*

\*

\* @author S542312

\*/

public class Throws {

double division(double x, double y) throws ArithmeticException{

double m = x/y;

return m;

}

}

**ThrowsDriver.java:**

package question09;

/\*\*

\*

\* @author S542312

\*/

public class ThrowsDriver {

int division(int x, int y) throws ArithmeticException{

int m = x/y;

return m;

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

ThrowsDriver thr = new ThrowsDriver();

try{

System.out.println(thr.division(14, 0));

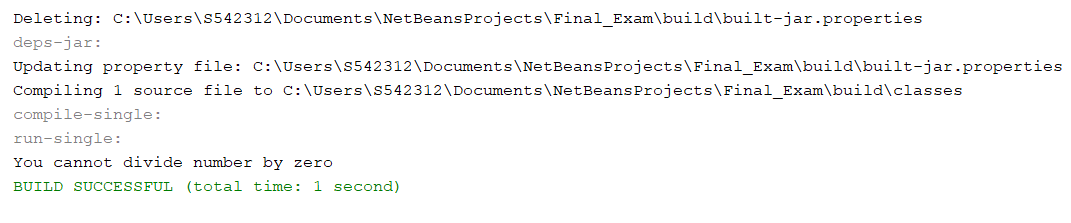
}

catch(ArithmeticException e){

System.out.println("You cannot divide number by zero");

}

}



**Example02:** In this example we can look art NullPointerException Example uses an object reference that has the null value. Calling an instance method on the object referred by a null reference. If the reference type is an array type, accessing or modifying the slots of a null reference.

**ThrowsException:**

package question09;

/\*\*

\*

\* @author S542312

\*/

public class ThrowExcep

{

static void fun()

{

try

{

throw new NullPointerException("demo");

}

catch(NullPointerException e)

{

System.out.println("Caught inside fun().");

throw e; // rethrowing the exception

}

}

public static void main(String args[])

{

try

{

fun();

}

catch(NullPointerException e)

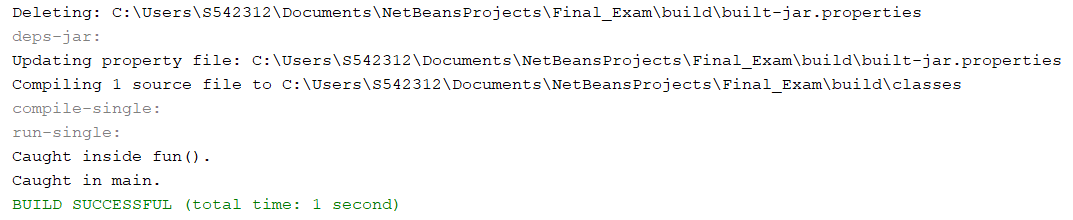
{

System.out.println("Caught in main.");

}

}

}



1. (15-Points) What is a recursive method? What is an infinite recursion? Explain and demonstrate with examples. Implement the search (element) in a list using recursion.

Ans) **Recursive method:** The technique of making a function call itself is known as recursion. This technique allows you to break down complex problems into smaller, easier-to- solve problems.

Recursion can be a complex concept to grasp. Experimenting with it is the only way to find out how it works.

New storage locations for variables are allocated on the stack when a recursive call is made. The old variables and parameters are removed from the stack as each recursive call returns. As a result, recursion usually consumes more memory and is slower. A recursive approach, on the other hand, is much easier and takes less time to write, debug, and maintain.

**Infinite Recursion:** If a recursion never hits a base case, it will continue to make recursive calls indefinitely, never terminating the program. This is referred to as infinite recursion, and it is usually not recommended.

Infinite recursion is the non-terminating execution of a block of code. Infinite recursion is usually caused by a bug in code.

**Program:**

**Example01:** The idea is to compare x with first element in arr[]. If element is found at first position, return it. Else recursion for remaining array and find x.

package question10;

/\*\*

\*

\* @author S542312

\*/

public class Recursion {

static int arr[] = {2, 34, 44, 7, 66};

/\* Recursive Method to search x in arr[l..r] \*/

static int recSearch(int arr[], int p, int q, int x)

{

if (q < p)

return -1;

if (arr[p] == x)

return p;

if (arr[q] == x)

return q;

return recSearch(arr, p+1, q-1, x);

}

public static void main(String[] args) {

// TODO code application logic here

int z = 44;

//Method call to find x

int index = recSearch(arr, 0, arr.length-1, x);

if (index != -1)

System.out.println("Element " + x + " is present at index " +

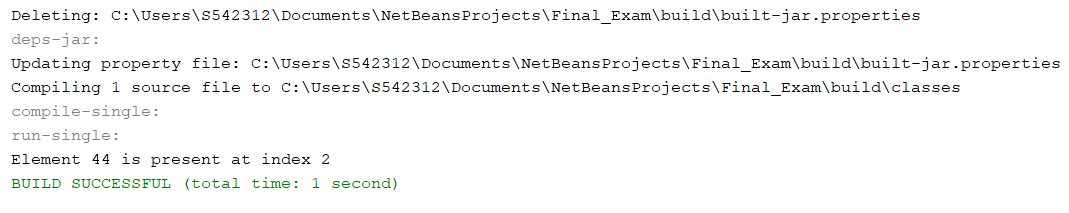
index);

else

System.out.println("Element " + x + " is not present");

}

}



**Example02:** The factorial() method is calling itself. Initially, the value of n is 12 inside factorial(). During the next recursive call, 11 is passed to the factorial() method. This process continues until n is equal to 0.

When n is equal to 0, the if statement returns false hence 1 is returned. Finally, the accumulated result is passed to the main() method.

package question10;

/\*\*

\*

\* @author S542312

\*/

public class Recursion02 {

/\*\*

\* @param args the command line arguments

\*/

static int factorial( int f ) {

if (f != 0) // termination condition

return f \* factorial(f-1); // recursive call

else

return 1;

}

public static void main(String[] args) {

// TODO code application logic here

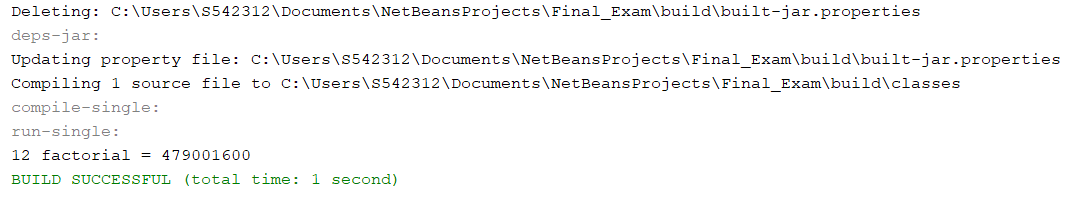
int number = 12, result;

result = factorial(number);

System.out.println(number + " factorial = " + result);

}

}



1. (10-Points) Write a java program that illustrates how equals() and hashCode() methods work? Explain your code in comments.

Ans) **Program:** If hashCode() is called more than once on the same Object during the application's execution, it must always return the same Integer value, as long as no information used in the equals(Object) comparison on the Object is changed. It is not necessary for this Integer value to remain constant from one program execution to the next.

* If the equals(Object) method determines that two Objects are identical, the hashCode() method must return the same Integer for each of the two Objects.
* If two Objects are unequal, the Integer value generated by the hashCode() method on each of the two Objects does not have to be distinct, according to the equals(Object) method. It can be the same, but generating a distinct Integer on each of the two Objects is better for hashing-based Collections like HashMap, HashTable, and so on.

package question11;

/\*\*

\*

\* @author S542312

\*/

public class hash\_equal {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

String x = "University";

String y = "University";

if(x.equals(y)){ //checking the equality of objects using equals() methods

System.out.println("x & y are equal variables, and their respective hashvalues are:" + " "+ x.hashCode() + " & " + y.hashCode());

}

String p = "Mehal";

String q= "Reddy";

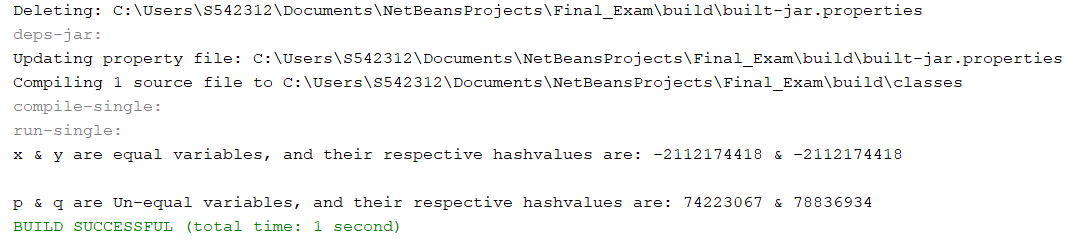
if(!p.equals(q)){ //checking the equality of objects using equals() method

System.out.println("\np & q are Un-equal variables, and their respective hashvalues are:" + " "+ p.hashCode() + " & " + q.hashCode());

}

}

}



1. (15-Points) Design Employee class and Employee driver class as follows:
2. **Employee Class implements Comparable<Employee**>

* Data fields named empId, empName and empSalary
* A constructor with parameters, listed in the same order as above.
* Create getter methods for all the parameters.
* A toString method that prints the empId, empName and empSalary. There should be one space between each value output.
* Because Employee implements the Comparable interface, you must also implement the compareTo method as defined by the Comparable interface. Define this method in such a way that the natural ordering of employees will be by id number, in ascending order.

1. **EmployeeDriver Class**

* Begin by filling an ArrayList with at least 5 employees. Add employees in random order – not by id number, not by name, and not by salary. The original list should not be in order by any of these attributes.
* Use an enhanced for loop to print the original list.
* Call the one-parameter sort method of the Collections class to sort the list by its natural order (empId number) and then print the list again.
* Call the two-parameter sort method of the Collections class, supplying a new Comparator<Employee> that sorts by salary. Print the list again.
* Call the two-parameter sort method of the Collections class, supplying a new Comparator<Employee> that sorts by name. Print the list again.

Ans) **Program:**

**Employee.Java:**

package question12;

/\*\*

\*

\* @author S542312

\*/

public class Employee implements Comparable<Employee> {

private int empId;

private String empName;

private double empSalary;

/\*\*

\* @param empId

\* @param empName

\* @param empSalary

\*/

public Employee(int empId, String empName, double empSalary) {

this.empId = empId;

this.empName = empName;

this.empSalary = empSalary;

}

public int getEmpId() {

return empId;

}

public void setEmpId(int empId) {

this.empId = empId;

}

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

public double getEmpSalary() {

return empSalary;

}

public void setEmpSalary(double empSalary) {

this.empSalary = empSalary;

}

@Override

public String toString() {

return "Emp Id: " + empId + ", Emp Name: " + empName + ", Emp Salary: "

+ empSalary;

}

@Override

public int compareTo(Employee other) {

if (this.empId < other.getEmpId())

return -1;

else if (this.empId > other.getEmpId())

return 1;

else

return 0;

}

}

**SortByName.java:**

package question12;

import java.util.Comparator;

/\*\*

\*

\* @author S542312

\*/

public class SortByName implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

if (e1.getEmpName().compareTo(e2.getEmpName()) < 0) {

return -1;

} else if (e1.getEmpName().compareTo(e2.getEmpName()) > 0) {

return 1;

} else {

return 0;

}

}

}

**SortBySalary.java:**

package question12;

import java.util.Comparator;

/\*\*

\*

\* @author S542312

\*/

public class SortBySalary implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

if (e1.getEmpSalary() > e2.getEmpSalary()) {

return 1;

} else if (e1.getEmpSalary() < e2.getEmpSalary()) {

return -1;

} else {

return 0;

}

}

}

**MainClass.java:**

package question12;

import java.util.ArrayList;

import java.util.Collections;

/\*\*

\*

\* @author S542312

\*/

public class EmployeeDriver {

public static void main(String[] args) {

ArrayList<Employee> emp = new ArrayList<Employee>();

emp.add(new Employee(12345, "Mehal", 96000));

emp.add(new Employee(12346, "Roopin", 34000));

emp.add(new Employee(12347, "Naveen", 60000));

emp.add(new Employee(12348, "Yaswant", 56000));

emp.add(new Employee(12349, "Vinod", 78000));

System.out.println("Displaying the employees List :");

display(emp);

Collections.sort(emp);

System.out.println("\n\nAfter Sorting displaying the employees List :");

display(emp);

Collections.sort(emp, new SortBySalary());

System.out.println("\n\nAfter Sorting by salary displaying the employees List :");

display(emp);

Collections.sort(emp, new SortByName());

System.out.println("\n\nAfter Sorting by employee name displaying the employees List :");

display(emp);

}

private static void display(ArrayList<Employee> emps) {

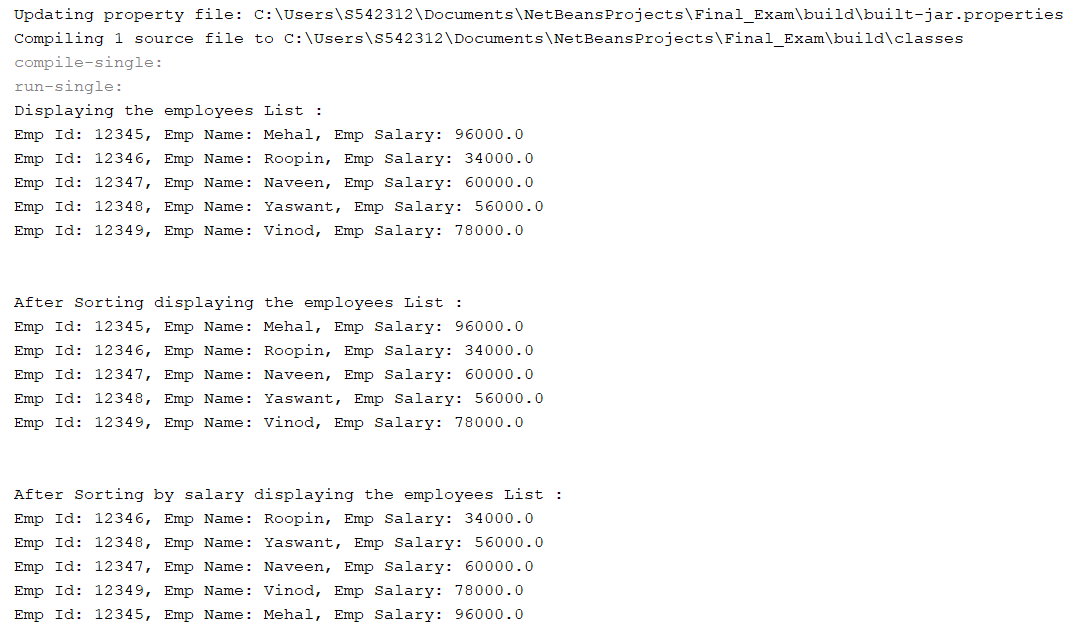
for (Employee emp : emps) {

System.out.println(emp);

}

}

}



**GitHub Link:**