**Final Exam Instructions**

**OBJECT-ORIENTED PROG 01FA2**

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.

|  |
| --- |
| **Abstract Class:**   * An abstract *class* is a class that is declared abstract, it may or may not include abstract methods. * Abstract classes are not instantiated, but they can be sub-classed. * It is partially implemented, class. * There is no object creation for abstract classes. * Abstraction is a process of hiding the implementation details and just showing the functionality. * An abstract class cannot be instantiated – it serves only as a parent class. * Syntax   public abstract class ClassName{  }  **Interfaces:**   * It is *a* mechanismto achieve abstraction. Interfaces can have a declaration of a method, but we can’t implement a method functionality( body) in the interface class. * It is used to achieve abstraction and multiple inheritances. * It follows the Is-A relationship. * Since Java 8, we can have default and static methods in an interface. * Interfaces in Java are like classes, except:   1. All methods are abstract.  2. All methods are public.  3. It has no instance value   * Syntax   interface <interface\_name>{  // declare constant fields  // declare methods that abstract  // by default.  }  **Differences between Abstract Classes and Interfaces**   * Abstract classes won’t support multiple inheritance, but interfaces support multiple inheritance. * An abstract class can be extended using the keyword "extends" but interfaces. * The interface has only static and final variables, but the Abstract class can have final, non-final, static, and non-static variables**.** * An abstract class can have class members like private, protected but interfaces are public by default.   **Interfaces are preferred over abstract classes:**   * One of the main purposes of using interface classes is, it has us achieve multiple inheritance. * Interfaces helps us to define the functionality, not implement it whereas abstract class create functionality that subclasses can implement or override. * Interfaces provide a safe and powerful increase of functionality - enables composition whereas abstract classes have no chance so it uses inheritance concept * In interfaces, Classes can implement several Mix-ins(methods for use by other classes without having to be the parent class of those other classes). * There is great damage to the hierarchy in abstract classes whereas in interfaces it uses “Implement Inheritance “in each class and adds the required methods in each class.   **Abstract Class Example:**  public abstract class Rectangle(){  public abstract double Area();  }  **Interface Class Example:**  public abstract class Rectangle(){  double Area();  }  **Similarities:**   * An object can’t be created for both interface and an abstract class. * All the abstract methods must be overridden by implementing a concrete class. * Both interface and abstract classes don’t have a default constructor. * By using both we can go for dynamic polymorphism. * Both interfaces and abstract classes contain static and final variables. * Both interfaces and abstract classes are inherited from a common domain itself using the extends keyword. * Both provide static method implementation**.**   **Program**  **ItalianFood Interface**  package ques01;  /\*\*  \*  \* @author Nithya Karepe  \*/  public interface ItalianFood {  public boolean getPepperroni();  public boolean getPineapple();  }  **PizzaType Class**  package ques01;  /\*\*  \*  \* @author Nithya Karepe  \*/  public abstract class PizzaType implements ItalianFood {  public String vegPizza;  public String nonVegPizza;  String type;  public PizzaType(String type) {  this.type = type;  }  public String getType() {  return type;  }  @Override  public boolean getPepperroni() {  if (!type.equalsIgnoreCase("nonVegPizza")) {  return false;  }  return true;  }  @Override  public boolean getPineapple() {  if (!type.equalsIgnoreCase("vegPizza")) {  return false;  }  return true;  }  public abstract double getCost();  public String AboutPizza() {  return "Pizza is a savory dish of Italian origin consisting of \n "  + "a usually round, flattened base of leavened wheat-based dough \n"  + " topped with tomatoes, cheese, and often various other ingredients, \n"  + " which is then baked at a high temperature, traditionally in a wood-fired oven.";  }  @Override  public String toString() {  return "type: " + type;  }  }  **PizzaBusiness**  package ques01;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class PizzaBusiness extends PizzaType implements ItalianFood {  public PizzaBusiness(String type) {  super(type);  }  @Override  public double getCost() {  if (type.equalsIgnoreCase("vegPizza")) {  return 6.4;  } else if (type.equalsIgnoreCase("nonVegPizza")) {  return 7.8;  }  return 0.0;  }  @Override  public String toString() {  return super.toString() + "\nCost: " + getCost() + "\n" + "Pineapple Topping: " + getPineapple() + "\n"  + "Pepperroni Topping: " + getPepperroni() + "\n";  }  }  **Class PizzaDriver**  package ques01;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class PizzaDriver {  public static void main(String[] args) {  PizzaBusiness p1 = new PizzaBusiness("nonVegPizza"); System.out.print(p1.toString());  }  }  **Output :** |

2.(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 side.

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.

|  |
| --- |
| **Interface Colorable**  package ques02;  /\*\*  \*  \* @author Nithya Karepe  \*/  public interface Colorable {  public void howToColor();  }  Class GeometricObject  package ques02;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class GeometricObject {  }  **GeometricObject**  package ques02;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class GeometricObject {  }  **ColorableDriver**  package ques02;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class ColorableDriver {  public static void main(String[] args) {  GeometricObject[] geometricObjects = new Square[5];  Square s1 = new Square(1.0);  Square s2 = new Square(2.0);  Square s3 = new Square(3.0);  Square s4 = new Square(4.0);  Square s5 = new Square(5.0);    geometricObjects[0] = s1;  geometricObjects[1] = s2;  geometricObjects[2] = s3;  geometricObjects[3] = s4;  geometricObjects[4] = s5;  for (GeometricObject go : geometricObjects) {  if (go instanceof Colorable) {  Colorable c = (Colorable) go;  c.howToColor();  }  Square s = (Square) go;  System.out.printf("Area %.2fcm\u00B2\n", s.getArea());  }  }  }  **Output**    **UML Diagram** |

3. (10-Points) What is casting? What are different types of casting? Explain and demonstrate with examples.

|  |
| --- |
| Casting from a subclass to a superclass is called upcasting. Usually upcasting is implicitly done.  Upcasting is somewhat similar to inheritance in Java. It’s common to use reference variables to refer to a more specific type which is done most of the time unknowingly.  Some of the common examples are listed below:  List<String> list = new ArrayList<>();  Map<Integer,String> map = new HashMap<Integer,String>();  To demonstrate upcasting let’s define an HairCareProduct class:  package ques03;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class HairCareProduct {  private String brand;  private double price;  public HairCareProduct(String brand, double price) {  this.brand = brand;  this.price = price;  }  public String getBrand() {  return brand;  }  public double getPrice() {  return price;  }  @Override  public String toString() {  return "brand=" + brand + ", price=" + price ;  }  }  Now extending the HairCareProduct to a more specific subclass  /\*\*  \*  \* @authorNithya karepe  \*/  public class HairConditioner extends HairCareProduct {  public HairConditioner(String brand, double price) {  super(brand, price);  }      public String veganProduct() {  if (super.getBrand().equalsIgnoreCase("Herbal Essences")) {  return "It's a vegan product which means that a product does"  + " not contain any animal products or animal-derived ingredients.";  }  return "Not a HairCare Vegan Product ";//To change body of generated methods, choose Tools | Templates.  }  @Override  public double getPrice(){  return super.getPrice();  }  @Override  public String toString() {  return super.toString()+"\n"+ veganProduct();  }    }  Now we can create an object of HairConditioner class in the below ways:  HairConditioner hc = new HairConditioner();  Or  HairCareProduct hcp = new HairConditioner();  Or  HairCareProduct hcp = hc;  All of the above are implicit.  We can also do it explicitly like below  hcp = (HairCareProduct) hc;  In Java there is no need to do explicit cast up. The compiler undestands and doesn’t display any errors.  Note, that reference can refer to any subtype of the declared type.  Using upcasting, we can restrict the number of methods available to HairConditioner instance. Now we can’t do anything that is specific to HairConditioner – we can’t call the veganProduct() on the hcp variable.  Although HairConditioner object remains HairConditioner object, calling veganProduct() would cause the compiler error:  // hcp.veganProduct(); The method veganProduct() is undefined for the type HairCareProduct  Upcasting, allows us to use polymorphism, by using the methods available from the superclass. This allows us to access the common methods available in the superclass when different subclass objects are added to a superclass list. Incase subclass override these methods the subclass specific behavior is displayed or executed.  Since every class by default inherits the Object class in Java, we have the inbuilt methods from Object class available even in the subclass.  To invoke veganProduct() we need to downcast hcp.  But now we’ll describe what gives us the upcasting. Thanks to upcasting, we can take advantage of polymorphism.  Downcasting is the casting from a superclass to a subclass.  In the same above example  HairCareProduct hcp = new HairConditioner();  We know that hcp variable refers to the instance of HairConditioner.  Here if we want to call hcp.veganProduct() since the method is not available in the HairCareProduct class, this would throw an error.  To overcome the above error we can downcast like  ((HairConditioner) hcp).veganProduct();  Here adding (HairConditioner) before the hcp variable is down casting it and will allow us to access the methods specific to the HairConditioner class.  Usually we can use instanceof operator before downcasting to check if the object belongs to the specific type and n to run into any errors  Below is the driver class which show the usage of both upcasting & downcasting  package ques03;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class HairDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  int price = 12;  double price2 = (double)price;  HairConditioner c = new HairConditioner("Herbal Essences", price2);    //upcasting  HairCareProduct h= c;  h = (HairCareProduct) c;  System.out.println(h.toString());    HairCareProduct h1 = new HairConditioner("Dove", 11);    if(h1 instanceof HairConditioner ){  //downcasting  ((HairConditioner) h1).getPrice();  ((HairConditioner) h1).veganProduct();  }    System.out.println(h1.toString());  }    }  Output : |

4.(15-Points) Suppose that Fruit, Apple, Orange, GoldenDelicious, and McIntosh are defined in the following inheritance hierarchy:

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.

**a. Is fruit instanceof Fruit?**

Yes,Instance of subclass is also an instance of super class.

**b. Is fruit instanceof Orange?**

No, it super class instance can’t always be sub-class.

**c. Is fruit instanceof Apple?**

We know that GoldenDelicious is the sub class of Apple.Instance of sub class is an instance of super class

**d. Is fruit instanceof GoldenDelicious?**

Yes,fruit is an instance of GoldenDelicious

**e. Is fruit instanceof McIntosh?**

No,as fruit is not an instance of McIntosh since it not a superclass of GoldenDelicious(Apple is superclass).fruit contains an instance if GoldenDelicious

**f. Is orange instanceof Orange?**

Yes,orange is instance of Orange.

**g. Is orange instanceof Fruit?**

Yes,Orange object is an instance of Orange. Orange is a sub class of Fruit so orange instanceof Fruit

**h. Is orange instanceof Apple?**

No, here Apple is not an super class of Orange(object orange contains instance of class Orange)

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

Fruit can invoke method orange but orange cannot as orange is not an instance of Apple whereas fruit is an instance of Apple.

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

The object orange is an instance of the class Orange therefore, orange can invoke makeOrangeJuice method whereas fruit is not an instance of Orange so it cannot.

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

Illegal Statement, instance of Apple cannot be assigned to object of orange class because Apple is not a sub class of Orange

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

Illegal Statement,super class(Apple) instance cannot be assigned to sub-class reference(McIntosh)

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

legal Statement, Apple is the super-class for class McIntosh and assignment of the instance of a sub-class to an object of the super class is possible.

5.(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.

|  |
| --- |
| **Circle**  package ques05;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Circle {  double radius;  public Circle(double radius) {  this.radius = radius;  }  public double getArea(){  return Math.PI \* radius \* radius;  }    }  **ComparableCircle**  package ques05;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class ComparableCircle extends Circle implements Comparable {  public ComparableCircle(double radius) {  super(radius);  }  @Override  public int compareTo(Object o) {  Circle c = (Circle) o;  if (this.getArea() > c.getArea()) {  return 1;  } else if (this.getArea() < c.getArea()) {  return -1;  }  return 0;  }  }  **CircleDriver**  package ques05;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class CircleDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  ComparableCircle c1 = new ComparableCircle(2.0);  ComparableCircle c2 = new ComparableCircle(3.0);  int compare = c1.compareTo(c2);  if (compare == 1) {  System.out.println("Circle c1 is larger");  } else if (compare == -1) {  System.out.println("Circle c2 is larger");  } else {  System.out.println("Both Circles are of equal area.");  }  }  }  **Output**    **UML Diagram** |

6.(20-Points) What is an exception? What are checked and unchecked exceptions? Explain and demonstrate with examples.

|  |
| --- |
| Exception:   * The general terminology of an exception is that, it is an unexpected/unwanted occurrence of an event that disturbs the normal flow of a program/process. * All exception classes are subtypes of the java.lang.Exception class. * Exception is a subclass of throwable class. There is another subclass in throwable class, those are called errors. * Errors can’t be handled by the user whereas exceptions can be handled. * Exceptions can be handled by using try-catch block or throws keyword. * Advantages of Exception Handling in Java  1. Provision to Complete Program Execution 2. Easy Identification of Program Code and Error-Handling Code 3. Meaningful Error Reporting  * Example:   Errors:  1. RAM is out of memory  2. JVM is not working properly.  Exceptions:  1. FileNotFound exception occurs when the application failed attempt to open the file denoted by a specified pathname or when the application wants to write but the file is read-only  2. NullPointerException occurs in code when we try to access/ modify an object which has not been initialized yet.    There are 2 types of exceptionsThey are : Checked Exception and Unchecked Exceptions  **Checked Exception**   * The exceptions that are checked at compile time are called checked exceptions. * These exceptions are also known as compile time exceptions. * The exception is handled using try-catch block or throws keyword. * The programmer should take care of (handle) these exceptions without ignoring. * Examples : ClassNotFoundException, IOException, SQLException etc  1. ClassNotFoundException is thrown when JVM tries to load a particular class but couldn’t find the path. Class.forName()is used to load the class by passing String name of a class and it’s not found on the classpath. 2. IOException is thrown when the programmers use in the code to throw a failure in Input & Output operations. 3. FileNotFound exception occurs when the application failed attempt to open the file denoted by a specified pathname or when the application wants to write but the file is read-only   **Unchecked Exception**   * This type of exceptions are checked during the run-time, and occur due to the bad data provided by the user during the user-program interaction. * These exceptions are also known as runtime exceptions * By extending runtime exception we can create a custom exception * These exceptions need not be handled at compile time * Examples Arithmetic Exception, ArrayStoreException etc  1. ArithmeticException: Arithmetic error, such as divide-by-zero. 2. ArrayIndexOutOfBoundsException: Array index is out-of-bounds. 3. ArrayStoreException: Assignment to an array element of an incompatible type. 4. NullPointerException occurs in code when we try to access/ modify an object which has not been initialized yet.   **CheckedException**  package ques06;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.util.Scanner;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class CheckedExceptions {  public void fileNotFoundException() {  File file = new File("nofile.txt");  try {  FileInputStream sc1 = new FileInputStream(file);  } catch (FileNotFoundException e) {  //Any file not found exceptions by the Scanner sc1 will be caught here  System.out.println("File Not Found");    }  }  public void classNotException() {  try {  Class.forName("ques06.UncheckedException");  } catch (ClassNotFoundException ex) {  System.out.println("Class Not Found");  }  }  }  **UncheckedException**  package ques06;  /\*\*  \*  \* @author S540109  \*/  public class UncheckedException {  int a = 0;  int b = 0;  String s = null;  String s1 = null;  public UncheckedException(int a, int b, String s) {  this.a = a;  this.b = b;  this.s = s;  }  public void arithemeticException()  try {  int output = a / b;  System.out.println("Result: " + output);  } catch (ArithmeticException e) {  System.out.println("Denominator shouldn't be zero");  }  }  public void stringIndexOutOfBoundsException() {  try {  //String str="beginnersbook";  System.out.println(s.length());  System.out.println(s.charAt(0));  System.out.println(s.charAt(23));  } catch (StringIndexOutOfBoundsException e) {  System.out.println("StringIndexOutOfBoundsException");  }  }  }  **ExceptionDriver**  package ques06;  /\*\*  \*  \* @author S540109  \*/  public class ExceptionDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // Checked Exceptions  CheckedExceptions e = new CheckedExceptions();  e.fileNotFoundException();  e.classNotException();  UncheckedException e2 = new UncheckedException(30,0, "Final Exam");  e2.arithemeticException();  e2.stringIndexOutOfBoundsException();  }  }  In the CheckedExceptions class within the method fileNotFoundException(), we notice that we use the File while creating the object for FileInputStream, this requires the file to be present else it would throw FileNotFoundException. Any subsequent use of the FileInputStream object would depend on its creation in the first place hence it is required to either handle that using a catch or let the user know it by declaring using throws in the method declaration.  In the CheckedExceptions class within the method classNotException(), we notice that we use the Class.forName , this again throws the ClassNotFoundException as the system does not know how to proceed further hence we catch it after the try block.  All of the Checked exceptions will throw compiler errors if not handled by either try catch block or at the method declaration using the throws keyword  In the UncheckedException class within the method arithemeticException(), we notice that we use a/b where bis the denominator, if for suppose the value of b is 0 we run into the ArithmeticException. in some cases the value of b or the division itself is only determined at the runtime so we run into the Unchecked Exception.  In the UncheckedException class within the method stringIndexOutOfBoundsException(), we can have string of varied length and if trying to retrieve value at a specific index which is out of bounds it’ll run into StringIndexOutOfBoundsException. Here the string or the index can be provided by the user hence this is also an Unchecked Exception.  Unchecked Exceptions will never throw complier errors but can fail at runtime hence whenever possible if there is a possibility of unchecked exception user need to handle them either by try catch block or at the method declaration using the throws keyword  **Output :** |

7. (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.

|  |
| --- |
| package ques07;  import java.util.Scanner;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class OutOfBound {  /\*\*  \* @param args the command line arguments  \*/  int[] a = new int[100];  /\*\*  \*  \* @return  \*/  public static void main(String[] args) {  // TODO code application logic here  Scanner sc = new Scanner(System.in);  int[] a = new int[100];  int i = 0;  while (i < a.length) {  a[i] = (int) (Math.random() \* 100) + 1;  i++;  }  // Prompt the user to enter the index of the array  System.out.print("Enter index: ");  try {  // Display the corresponding element value  System.out.println("The corresponding element is "  + a[sc.nextInt()]);  } catch (ArrayIndexOutOfBoundsException ex) {  System.out.println("Out of Bounds.");  }  }  }  **Output 1**    **Output 2** |

8.(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.

|  |
| --- |
| Exception usually means error, when we are writing code, some exceptions can be handled and some cannot be, but all these exceptions provide the user with a meaningful information to either point that there is something wrong that is happening or an unexpected flow the user is running into. An exception can occur for many different reasons it could be a code issue or a jvm issue or a connection issue.   * The purpose of declaring exceptions is to provide the user using that piece of code, to let them know of the possible exceptions that a user can run into. * In Java one of the main concepts is code reusability, so in order for the code to be reusable the code must mention all the exceptions that could happen due to either bad user input or any other issue. * The throws keyword is used to declare the exception that can occur during the program flow. It gives information about the exception to the programmer. * It is always better to handle the exceptions so that there is no abrupt break in the flow. * All exceptions should be Throwable.   Within code generally there are 3 places we use exceptions   * When creating a custom exception. You can have a checked exception by extending the Exception class. If you want to have an unchecked exception we can extend RuntimeException class. * When catching the exception after try block in try catch. While throwing the exception you always throw a single exception in a single java statement but when catching the exception you can catch multiple exceptions or also use the parent/ super class which has multiple exception subclasses * In the method header if the method throws an exception. Within the method for custom exceptions or user thrown exceptions you’d use the throw keyword. Also Here you can declare multiple exceptions in the method header using the throws keyword.   In the below example if you notice we only throw one exception at a time. Once the exceptions is thrown it stops the flow of execution, hence in our example we would either get ArithmeticException or NumberFormatException depending on the value of b.  However during the method declaration you’d still need to mention all the exceptions that can be thrown from within that method hence we have  myMethod() throws ArithmeticException, NumberFormatException  package ques08;  import java.io.\*;  import java.text.NumberFormat;  /\*\*  \*  \* @author nithya Karepe  \*/  public class MultiHeader {  /\*\*  \* @param args the command line arguments  \*/  static void myMethod() throws ArithmeticException, NumberFormatException {  int a = 9;  int b = 0;  if (b == 0) {  throw new ArithmeticException("Arithmetic Exception");  }  a = Integer.parseInt(null);  throw new NumberFormatException("NumberFormat Exception");  }  public static void main(String args[]) {  try {  myMethod();  } catch (Exception ex) {  System.out.println(ex);  }  }  }  **Output** |

9. (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.

|  |
| --- |
| Throw keyword is used to throw an exception, either a custom one or inbuilt java defined exception, we throw exceptions to let the user know that the current flow of code needs to be halted or cannot proceed further. While throwing the exception you always throw a single exception in a single java statement  We use Throws keyword in the method header if the method throws an exception or multiple exceptions. Here these exceptions could be user thrown exceptions or because of the code like in case of creating a File, by default the method using the file needs to throw the FileNotFoundException.  No we cannot throw multiple exceptions in a single throw statement as once the exceptions is thrown it stops the flow of execution and there is no point in going to next exception. The whole point of exception is to let the user know of the error and narrow down the cause of the error. If multiple exceptions can be thrown from single place it would loose the meaning and it would be confusing. Instead of throwing multiple exceptions the use can throw Exception which is like the root class for all the exceptions.  Below are some of the sample code where I was able to use the throw, throws keywords and display the use of exceptions  **GradeCheckWithThrow**  package ques09;  import java.io.IOException;  /\*\*  \*  \* @author Nithya  \*/  public class GradeCheckWithThrow {  static void checkGrade(String grade) {  if (!grade.equalsIgnoreCase("A") || !grade.equalsIgnoreCase("B")) {  throw new ArithmeticException("Retake Course");  } else {  System.out.println("Good Job!");  }  }  public static void main(String[] args) {  checkGrade("B");  }  }    Checked  package ques09;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  /\*\*  \*  \* @author S540109  \*/  public class Checked {  public void fileNotFound() throws FileNotFoundException{  File file = new File("nofile.txt");  FileInputStream sc1 = new FileInputStream(file);  System.out.println("File Not Found");    }  public void classNotException() throws ClassNotFoundException {  Class.forName("ques09.Unchecked");    }  }  **Unchecked**  package ques09;  /\*\*  \*  \* @author S540109  \*/  public class Unchecked {  int a = 0;  int b = 0;  String s = null;  public Unchecked(int a, int b, String s) {  this.a = a;  this.b = b;  this.s = s;  }  public void arithmetic() throws ArithmeticException{  int output = a / b;  System.out.print("Result: " + output);  }  public void stringIndexOutOfBound() throws StringIndexOutOfBoundsException {  //String str="beginnersbook";  System.out.println(s.length());  System.out.println(s.charAt(0));  System.out.println(s.charAt(23));  }  }  **Driver**  package ques09;  import java.io.FileNotFoundException;  /\*\*  \*  \* @author S540109  \*/  public class Driver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) throws ArithmeticException, StringIndexOutOfBoundsException, FileNotFoundException, ClassNotFoundException{  // TODO code application logic here  Checked e = new Checked();  e.fileNotFound();  e.classNotException();  Unchecked e2 = new Unchecked(30,0, "Final Exam");  e2.arithmetic();  e2.stringIndexOutOfBound();  }    }  **Output** |

10. (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.

|  |
| --- |
| **Recursion**:  The technique in which the function calls itself is called recursion.   * Recursion helps to break complicated problems so that we have a simple solution. * There are two requirements for recursive function, they are  1. Stop Condition - when some condition is satisfied, the function returns a value without a further recursive call 2. Recursive Call – the function calls itself with an input which is a step closer to the stop condition  * Advantages:  1. Reduces time complexity. 2. Adds more clarity and reduces the time for debugging the code. 3. Recursion is better at tree traversal. 4. More memory usage.  * Syntax   returntype method(){  //code to be executed  method();//calling same method  }  **Recursion - Factorial Process with Example and Explanation :**  package ques10;  import java.util.Scanner;  /\*\*  \* @author Nithya Karepe  \*  \*/  public class FactorialRecursion {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int num = sc.nextInt();  long factorial = calcFactorial(num);  System.out.println(num + "! = " + factorial);  sc.close();  }  public static long calcFactorial(int num) {  if (num >= 1)  return num \* calcFactorial(num - 1);  else  return 1;  }  }  **Output :**     * Function calcFactorial(num) is called from the main() function. * Here we can take any random number to calculate the factorial. * Consider num 4, As 4 is greater than 0, 4 is multiplied to the result of calcFactorial() i.e 3 (num(4) -1)and it goes on. * Asit is called from the same function, it is a recursive call. * In each recursive call, the value of the argument number is decreased by 1for every step * Here in the example, the function is calling itself multiple numbers of times and comes out of the loop when the condition is completely satisfied i.e then the number becomes 0. * There is no recursive call when the number becomes 0. * Answer 4! = 4\*3\*2\*1 = 24, Displays 24. * Here we observe, after each time the method is called the bigger problem is split into smaller problems.   **Infinite Recursion:**   * Any recursion which continues without is called infinite recursion. * In this type of recursion, there is no base condition. * It keeps on calling itself as there no condition which is getting satisfied. * To say more precisely, It occurs if the recursion step does not reduce the problem.   **Infinite Recursion - Factorial Program with Example and Explanation**  package ques10;  import java.util.Scanner;  /\*\*  \* @author Nithya Karepe  \*  \*/  public class InfiniteFactorialRecursion {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int num = sc.nextInt();  long factorial = infiniteRecursionFactorial(num);  System.out.println(num + "! = " + factorial);  sc.close();  }  // example for infinite recursion  public static long infiniteRecursionFactorial(int num) {  return num \* infiniteRecursionFactorial(num - 1);  }  }  **Output**     * We have taken factorial example to demonstrate infinite recursion * Firstly, Function infiniteRecursionFactorial(num) is called from the main() function. * We have taken the num to be 4, it starts following the method but as there is no condition to stop, it keeps on running. * Here from the above example, we understand that the method is never-ending. This is happening because there is no base condition in the method (infiniteRecursionFactorial(int num)). * In this case, the condition is not satisfied at any point. * There are infinite times of recursive calls as there is no endpoint   **BinarySearchRecursion**  package ques10;  import java.util.ArrayList;  import java.util.Collections;  import java.util.Scanner;  /\*\*  \* @author Nithya Karepe  \*  \*/  public class BinarySearchRecursion {  public static void main(String args[]) {  ArrayList<Integer> list = new ArrayList<>();  while (list.size() < 100) {  list.add((int) ((Math.random() \* 100) + 1));  }  // binary Search works only on sorted elements  Collections.sort(list);  System.out.println("List of numbers to search from are:");  for (int j = 0; j < list.size(); j++) {  System.out.print(list.get(j) + " ");  }  Scanner sc = new Scanner(System.in);  System.out.println("\nEnter the number to search :");  int x = sc.nextInt();  sc.close();  boolean result = binarySearch(list, 0, list.size() - 1, x);  if (result)  System.out.println("Number found");  else  System.out.println("Number not found");  }  // method for binarySearch given the arraylist using recursion, left, right and  // search number  public static boolean binarySearch(ArrayList<Integer> list, int l, int r, int x) {  if (r >= l) {  int center = l + (r - l) / 2;  // If the number is present at the center  if (list.get(center) == x) {  return true;  }  // If number is smaller than center  if (x < list.get(center)) {  return binarySearch(list, l, center - 1, x);  }  // If number is greater than center  else {  return binarySearch(list, center + 1, r, x);  }  }  // We reach here when the number is not present  return false;  }  }  **Output 1**    **Output 2** |

11. (20-Points) Design Employee class and Employee driver class as follows:

**A.** **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.

**B.** **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.

|  |
| --- |
| **Class Employee**  package ques11;  /\*\*  \* @author Nithya Karepe  \*  \*/  public class Employee implements Comparable<Employee> {  private long empId;  private String empName;  private double empSalary;  public Employee(long empId, String empName, double empSalary) {  super();  this.empId = empId;  this.empName = empName;  this.empSalary = empSalary;  }  public long getEmpId() {  return empId;  }  public String getEmpName() {  return empName;  }  public double getEmpSalary() {  return empSalary;  }  @Override  public String toString() {  return "empId=" + empId + ", empName=" + empName + ", empSalary=" + empSalary;  }  @Override  public int compareTo(Employee e) {  if (this.empId > e.empId)  return 1;  if (this.empId < e.empId)  return -1;  else  return 0;  }  }  **EmployeeDriver**  package ques11;  import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  /\*\*  \* @author Nithya Karepe  \*  \*/  public class EmployeeDriver {  /\*\*  \* @param args  \*/  public static void main(String[] args) {  ArrayList<Employee> employeelist = new ArrayList<>();  employeelist.add(new Employee(4, "John", 3000));  employeelist.add(new Employee(1, "Jay", 8000));  employeelist.add(new Employee(3, "Kim", 4000));  employeelist.add(new Employee(5, "Joe", 6000));  employeelist.add(new Employee(2, "Jack", 5000));  for (Employee e : employeelist) {  System.out.println(e.toString());  }  Collections.sort(employeelist);  System.out.println();  for (Employee e : employeelist) {  System.out.println(e.toString());  }  Comparator<Employee> compareBySalary = new Comparator<Employee>() {  @Override  public int compare(Employee e1, Employee e2) {  if (e1.getEmpSalary() > e2.getEmpSalary())  return 1;  if (e1.getEmpSalary() < e2.getEmpSalary())  return -1;  else  return 0;  }  };  Collections.sort(employeelist, compareBySalary);  System.out.println();  for (Employee e : employeelist) {  System.out.println(e.toString());  }  Comparator<Employee> compareByName = new Comparator<Employee>() {  @Override  public int compare(Employee e1, Employee e2) {  return e1.getEmpName().compareTo(e2.getEmpName());  }  };  Collections.sort(employeelist, compareByName);  System.out.println();  for (Employee e : employeelist) {  System.out.println(e.toString());  }  }  }  **Output** |