MODULE - 1

Q1.a What is scope and life time of a variable? Discuss each with the help of a java program.

A: Variable scope: A variable's scope is the place in your program where it can be referenced. Variable scope is specified using the 'scope' keyword in the variable declaration. A variable that can be accessed in 'any' scope can be accessed absolutely anywhere in your program. A variable that can be accessed only in a specific scope can only be accessed in that scope. The scope may be a function, a block, a method, a class, or outside of all methods, blocks and classes.

Life time of a variable: The lifetime of a variable refers to the duration for which the variable exists in memory. It starts when the variable is declared and ends when it goes out of scope or is explicitly deallocated.

There are three types of variables:

1. Instance Variables: A variable which is declared inside a class, but is declared outside any methods and blocks is known as instance variable.
   Scope: Throughout the class except in the static methods.
   Lifetime: Until the object of the class stays in the memory.
2. Class Variables: A variable which is declared inside a class, outside all the blocks and is declared as static is known as class variable.
   Scope: Throughout the class.
   Lifetime: Until the end of the program.
3. Local Variables: All variables which are not instance or class variables are known as local variables.
   Scope: Within the block it is declared.
   Lifetime: Until control leaves the block in which it is declared.

Example

```
public class ExampleOfVariable {
   int num1, num2;   //Instance Variables
   static int result;  //Class Variable
   int add(int a, int b){  //Local Variables
      num1 = a;
      num2 = b;
      return a+b;
   }
   public static void main(String args[]){
      scope_and_lifetime ob = new scope_and_lifetime();
      result = ob.add(10, 20);
      System.out.println("Sum = " + result);
   }
}
```

Output:

Sum = 30

Explanation: In above example num1 and num2 are Instance Variables, result is a Class Variable and a & b are Local Variables for method ‘add’

Q1.b With the help of a proper java code, explain type conversion and type casting.

A: Type conversion:

1. Type conversion is the process of converting one data type into another implicitly.
2. It is also known as 'widening type casting' or 'casting down'.
3. It is done automatically.
4. It is safe because there is no chance to lose data.

Type casting:

1. Type casting is the process of converting one data type into another explicitly.
2. It is also known as 'narrowing type casting' or 'casting up'.
3. It is done manually by the programmer.
4. If we do not perform casting then the compiler reports a compile-time error.

Example of Type Conversion

```
public class TypeConversionExample
{
   public static void main(String[] args)
   {
          int x = 7;
          long y = x;
          float z = y;
          System.out.println("Before conversion, int value "+x);
          System.out.println("After conversion, long value "+y);
          System.out.println("After conversion, float value "+z);
   }
}
```

Output:

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

Example of Type Casting

```
public class TypeCastingExample
{
   public static void main(String args[])
   {
          double d = 166.66;
          long l = (long)d;
          int i = (int)l;
          System.out.println("Before conversion: "+d);
          System.out.println("After conversion into long type: "+l);
          System.out.println("After conversion into int type: "+i);
   }
}
```

Output

Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166

Q2.a Explain working of for-each version of for loop using a java program.
A: For-each loop:
The "for-each" loop in Java, also known as an enhanced for loop, is used to iterate through elements in an array or collections (e.g., ArrayList, Set, etc.) without the need for explicit index control. It simplifies the process of iterating through the elements of a collection. Here's a Java program that demonstrates the working of the "for-each" loop with an array:

```
public class ForEachLoopExample {
   public static void main(String[] args) {
      int[] numbers = {1, 2, 3, 4, 5};
      System.out.println("Using a for-each loop to iterate through the array:");
      for (int num : numbers) {
         System.out.print(num + " ");
      }
   }
}
```

Output:
1 2 3 4 5

Explanation:

1. We create an array of integers called `numbers`.
2. We use a "for-each" loop to iterate through the elements of the `numbers` array. The syntax for the "for-each" loop is `for (datatype variable : array/collection)`. In this case, we use `int num` as the loop variable to represent each element of the array.
3. Inside the "for-each" loop, we print each element of the array.

Q2.b Write a java program to demonstrate constructor overloading and method overloading.
A: Constructor Overloading:
In constructor overloading, we define multiple constructors within a class, each having a different number or type of parameters.

Method Overloading:
In method overloading, we define multiple methods within a class with the same name but different parameters.

Example

```
public class OverloadingDemo {

   public OverloadingDemo() {
      System.out.println("Default Constructor");
   }

   public OverloadingDemo(int num) {
      System.out.println("Parameterized Constructor with one integer: " + num);
   }
```

```
    public OverloadingDemo(String text) {
        System.out.println("Parameterized Constructor with one string: " + text);
    }

    public void printInfo() {
        System.out.println("Method with no arguments");
    }

    public void printInfo(int num) {
        System.out.println("Method with one integer argument: " + num);
    }

    public void printInfo(String text) {
        System.out.println("Method with one string argument: " + text);
    }

    public static void main(String[] args) {
        // Constructor Overloading
        OverloadingDemo constructorDemo1 = new OverloadingDemo();
        OverloadingDemo constructorDemo2 = new OverloadingDemo(42);
        OverloadingDemo constructorDemo3 = new OverloadingDemo("Hello");

        // Method Overloading
        OverloadingDemo methodDemo = new OverloadingDemo();
        methodDemo.printInfo();
        methodDemo.printInfo(42);
        methodDemo.printInfo("Hello");
    }
}
```

Output:

```
Default Constructor
Parameterized Constructor with one integer: 42
Parameterized Constructor with one string: Hello
Default Constructor
Method with no arguments
Method with one integer argument: 42
Method with one string argument: Hello
```

In this program:

1. We have three constructors, demonstrating constructor overloading:
   - `OverloadingDemo()`: Default constructor.
   - `OverloadingDemo(int num)`: Constructor with an integer parameter.
   - `OverloadingDemo(String text)`: Constructor with a string parameter.

2. We have three methods, demonstrating method overloading:
   - `printInfo()`: Method with no arguments.
   - `printInfo(int num)`: Method with an integer argument.

- `printInfo(String text)`: Method with a string argument.

In the `main` method, we create instances of the `OverloadingDemo` class and call both constructors and methods with various arguments to demonstrate the overloading concepts.

Module - 2

Q3.a What is inheritance? Write a java program that implements simple inheritance.

A: Inheritance :

Inheritance is one of the fundamental concepts in object-oriented programming (OOP). It allows a new class to inherit properties and behaviors (i.e., fields and methods) from an existing class, promoting code reusability and creating a hierarchy of classes. The class that is inherited from is called the base class or superclass, and the class that inherits is called the derived class or subclass.

Types of Inheritance in Java:

1. Single-level inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Example

```
class Vehicle {
   String brand;
   int year;

   Vehicle(String brand, int year) {
      this.brand = brand;
      this.year = year;
   }

   void displayInfo() {
      System.out.println("Brand: " + brand);
      System.out.println("Year: " + year);
   }
}


class Car extends Vehicle {
   int numberOfDoors;

   Car(String brand, int year, int numberOfDoors) {
      super(brand, year);
      this.numberOfDoors = numberOfDoors;
   }

   void displayCarInfo() {
      displayInfo();
```

```
        System.out.println("Number of Doors: " + numberOfDoors);
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2022, 4);
        System.out.println("Car Information:");
        myCar.displayCarInfo();
    }
}
```

Output:
Car Information:
Brand: Toyota
Year: 2022
Number of Doors: 4

Explanation:

1. We define a base class called `Vehicle` with fields `brand` and `year`. It also has a parameterized constructor and a method `displayInfo` to display the vehicle's information.
2. We define a derived class called `Car`, which inherits from the `Vehicle` class. The `Car` class adds a new field `numberOfDoors`. It has its own constructor that calls the constructor of the superclass using `super(brand, year)`. It also has a new method `displayCarInfo` that reuses the `displayInfo` method from the superclass and adds information specific to cars.
3. In the `main` method, we create an instance of the `Car` class and demonstrate the use of inheritance by displaying car information using methods from both the superclass and the subclass.

Q3.b What are the two different use of super keyword in java? Illustrate each with proper examples.
A: Super:
The super keyword refers to superclass (parent) objects.
In Java, the `super` keyword is used in two different contexts:

1. To Call a Superclass Constructor:
The `super` keyword can be used to call a constructor of the superclass from the constructor of the subclass. This is often used when the subclass needs to perform some additional initialization while reusing the constructor of the superclass.

Example: Using `super` to call a Superclass Constructor

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void makeSound() {
        System.out.println("Animal sound");
    }
}
```

```
class Dog extends Animal {
    String breed;

    Dog(String name, String breed) {
        super(name);
        this.breed = breed;
    }

    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class SuperKeywordExample1 {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever");
        System.out.println("Name: " + myDog.name);
        System.out.println("Breed: " + myDog.breed);
        myDog.makeSound();
    }
}
```

Output:
Name: Buddy
Breed: Golden Retriever
Dog barks

In this example, we use `super(name)` in the `Dog` constructor to call the constructor of the `Animal` superclass, passing the `name` parameter. This ensures that the `name` field of the superclass is initialized correctly.

2. To Access Superclass Members:
The `super` keyword can be used to access members (fields or methods) of the superclass within the subclass, especially when there is a name conflict between the subclass and superclass members.

Example: Using `super` to Access Superclass Members

```
class Parent {
    int age;

    Parent(int age) {
        this.age = age;
    }

    void displayAge() {
        System.out.println("Age in Parent class: " + age);
    }
}

class Child extends Parent {
    int childAge;

    Child(int age, int childAge) {
```

```
        super(age);
        this.childAge = childAge;
    }

    void displayAge() {
        super.displayAge();
        System.out.println("Age in Child class: " + childAge);
    }
}

public class SuperKeywordExample2 {
    public static void main(String[] args) {
        Child myChild = new Child(35, 7);
        myChild.displayAge();
    }
}
```

Output:
Age in Parent class: 35
Age in Child class: 7

In this example, we use `super.displayAge()` in the `Child` class to access the `displayAge` method of the `Parent` superclass. This allows us to call the method from the superclass within the overridden method of the subclass, avoiding name conflicts.

Q4.a What is dynamic method dispatch? Explain how a superclass reference variable can refer to a subclass object with the help of a program.

A: Dynamic method dispatch:
Dynamic method dispatch is a mechanism in object-oriented programming languages, such as Java, that allows a superclass reference variable to refer to a subclass object and invoke overridden methods based on the actual object's type at runtime. This concept is fundamental to achieving runtime polymorphism, where the appropriate method to execute is determined dynamically based on the actual object's type.

Here's how dynamic method dispatch works and how a superclass reference variable can refer to a subclass object in Java:

Create a superclass with a method, and then create a subclass that overrides that method. In Java, you can use inheritance to establish this relationship.

Instantiate an object of the subclass.

You can assign the subclass object to a reference variable of the superclass type. This is where dynamic method dispatch occurs.

Use the reference variable to call a method. The actual method invoked depends on the runtime type of the object.

Example

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}
```

```
class Circle extends Shape {
   @Override
   void draw() {
      System.out.println("Drawing a circle.");
   }
}

class Rectangle extends Shape {
   @Override
   void draw() {
      System.out.println("Drawing a rectangle.");
   }
}

public class DynamicMethodDispatchExample {
   public static void main(String[] args) {
      Shape shape1 = new Circle();
      Shape shape2 = new Rectangle();
      shape1.draw();
      shape2.draw();
   }
}
```

Output:
Drawing a circle.
Drawing a rectangle.

Explanation:

- We have a `Shape` superclass with a `draw` method, and two subclasses, `Circle` and `Rectangle`, which override the `draw` method.
- In the `main` method, we create instances of both `Circle` and `Rectangle` classes.
- We assign these objects to reference variables of the `Shape` superclass, which demonstrates dynamic method dispatch.
- When we call the `draw` method using the `shape1` and `shape2` references, the actual method invoked depends on the runtime type of the objects. In this case, it calls the overridden `draw` methods in the `Circle` and `Rectangle` subclasses.

Q4.b List out the conditions that are need to be followed while using abstract classes. Demonstrate the same by creating an abstract class and method.
A: When using abstract classes in Java, there are certain conditions and rules that need to be followed:

1. Declaration with the `abstract` Keyword: An abstract class is declared using the `abstract` keyword before the `class` keyword.
2. Abstract Methods: Abstract classes can contain abstract methods, which are declared using the `abstract` keyword and have no method body. Subclasses must provide concrete implementations for these abstract methods.
3. Can Have Concrete Methods: Abstract classes can also contain concrete (non-abstract) methods. Subclasses may inherit these methods as is or override them.
4. Cannot Be Instantiated: You cannot create instances of an abstract class using the `new` keyword. Abstract classes exist to be subclassed.
5.May Extend Other Classes: Abstract classes can extend other classes (abstract or non-abstract). Java supports single inheritance for classes, so an abstract class can extend only one class, but it can implement multiple interfaces.

Example: Here's a Java program that demonstrates the use of an abstract class and method

```
abstract class Shape {
    String color;

    Shape(String color) {
        this.color = color;
    }

    abstract double calculateArea();

    void displayInfo() {
        System.out.println("Color: " + color);
    }
}


class Circle extends Shape {
    double radius;

    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}


class Rectangle extends Shape {
    double width, height;

    Rectangle(String color, double width, double height) {
        super(color);
        this.width = width;
        this.height = height;
    }

    @Override
    double calculateArea() {
        return width * height;
    }
}

public class AbstractClassExample {
    public static void main(String[] args) {
        Circle circle = new Circle("Red", 5.0);
        Rectangle rectangle = new Rectangle("Blue", 4.0, 6.0);

        System.out.println("Circle Area: " + circle.calculateArea());
```

```
        circle.displayInfo();

        System.out.println("Rectangle Area: " + rectangle.calculateArea());
        rectangle.displayInfo();
    }
}
```

Output:
Circle Area: 78.53981633974483
Color: Red
Rectangle Area: 24.0
Color: Blue

Explanation:

- We define an abstract class `Shape` with an abstract method `calculateArea()` and a concrete method `displayInfo()`. The abstract class also has a field `color` and a constructor.

- We create two concrete subclasses, `Circle` and `Rectangle`, which inherit from the abstract class `Shape`. Each subclass provides an implementation of the `calculateArea()` method.

- In the `main` method, we create instances of `Circle` and `Rectangle`, demonstrate the calculation of areas by calling the abstract method, and display information using the concrete method.

Module – 3

Q5.a Define interface. Discuss the features of interface and explain them with the help of java program that implements an interface.

A: Interface:
An interface in Java is a programming construct that defines a contract of methods that a class implementing the interface must provide. It acts as a blueprint for classes, specifying a set of methods that they must implement, but it doesn't provide method implementations itself. An interface is declared using the `interface` keyword.

Here are some key features of interfaces:

1. Methods are Abstract: All methods declared in an interface are implicitly `public`, `abstract`, and `default` (if not explicitly marked `static` or `default`). They have no method bodies, and they are meant to be overridden by classes that implement the interface.
2. Multiple Inheritance: Java allows a class to implement multiple interfaces. This provides a way to achieve multiple inheritance, allowing a class to inherit from more than one type.
3. No Instance Variables: Interfaces cannot contain instance variables (fields) or constructors.
4. No Method Implementations: Interfaces don't provide method implementations; it's up to the classes that implement the interface to provide concrete implementations.

Example of interface:

```
interface Shape {
    double area();
    double perimeter();
```

```
}


class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}


class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }

    @Override
    public double perimeter() {
        return 2 * (width + height);
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        Rectangle rectangle = new Rectangle(4.0, 6.0);

        System.out.println("Circle - Area: " + circle.area() + ", Perimeter: " + circle.perimeter());
        System.out.println("Rectangle - Area: " + rectangle.area() + ", Perimeter: " + rectangle.perimeter());
    }
}
```

Output:
Circle - Area: 78.53981633974483, Perimeter: 31.41592653589793
Rectangle - Area: 24.0, Perimeter: 20.0

Explanation:

- We define an interface `Shape` with two abstract methods: `area()` and `perimeter()`.
- We create two classes, `Circle` and `Rectangle`, both of which implement the `Shape` interface. These classes provide concrete implementations of the `area()` and `perimeter()` methods.
- In the `main` method, we create instances of the `Circle` and `Rectangle` classes, calculate and display their areas and perimeters using the methods defined in the interface.

Q5.b List out the differences between abstract class and interface.
A:

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Q6.a What is package in java? List and explain the system packages in java.

A: In Java, a package is a way to organize and structure classes, interfaces, and other Java elements into a hierarchical directory structure. Packages serve several purposes, including:

1. Organization: Packages help organize and categorize related classes and interfaces, making it easier to manage and maintain large codebases.
2. Access Control: Packages allow you to control the visibility and access of classes and their members by defining package-private, protected, and public access levels.
3. Name Avoidance: Packages prevent naming conflicts by ensuring that classes with the same name can coexist in different packages.

Java has several system-defined packages, which are also known as the standard packages or built-in packages. These packages are part of the Java Standard Library and provide a wide range of functionality. Here are some of the key system packages in Java and their brief explanations:

1. `java.lang` Package:
   - This package is automatically imported in every Java program.
   - It contains fundamental classes and exceptions used in Java, such as `Object`, `String`, `System`, and `RuntimeException`.
   - It provides basic functionalities and features like multithreading, object management, and basic data types.

2. `java.util` Package:
   - It contains utility classes and interfaces for data structures, such as lists, sets, maps, and collections.
   - Classes like `ArrayList`, `HashMap`, and `Date` are part of this package.
   - It includes the `Scanner` class for input and the `Random` class for generating random numbers.

3. `java.io` Package:
   - It provides classes for input and output operations, including file handling and stream processing.
   - Classes like `File`, `InputStream`, `OutputStream`, and `Reader` are included.

4. `java.math` Package:
   - It contains classes for arbitrary-precision arithmetic.
   - The `BigInteger` and `BigDecimal` classes are part of this package, which allows precise numeric calculations.

5. `java.net` Package:
   - It offers classes for network programming, including sockets and network communication.
   - Classes like `Socket`, `ServerSocket`, and `URL` are part of this package.

6. `java.awt` and `javax.swing` Packages:
   - These packages are part of Java's Abstract Window Toolkit (AWT) and Swing libraries for creating graphical user interfaces (GUIs).
   - They include classes and components for building GUI applications.

7. `java.sql` Package:
   - It provides classes for database connectivity and SQL operations.
   - Classes like `Connection`, `Statement`, and `ResultSet` are included, making it possible to interact with databases.

8. `java.time` Package:
   - Introduced in Java 8, this package contains classes for date and time handling.
   - It includes classes like `LocalDate`, `LocalTime`, `ZonedDateTime`, and provides comprehensive support for working with dates and times.

9. `java.security` Package:

- It provides classes and interfaces for implementing security features in Java applications.
- Classes for cryptographic operations, secure random numbers, and authentication are included in this package.

Q6.b If user want to group all the similar type of classes and interfaces and keep them in a package ands access them, how it can be done? Explain.

A: To group similar classes and interfaces into a package and access them in Java, you can follow these steps:

1. Create a Package:
   - Decide on a package name that reflects the purpose or category of the classes and interfaces you want to group together.
   - In your project directory, create a subdirectory with the same name as the package. This subdirectory should be under the root directory of your Java project.

2. Organize Classes and Interfaces:
   - Place the source code files (`.java` files) of the classes and interfaces you want to group in the package's subdirectory. Make sure that the package declaration at the top of each source file matches the package name you've chosen.

   For example, if you want to create a package called `myapp.utilities`, the package declaration in your source files should look like this:
   package myapp.utilities;

3. Compile the Code:
   - Compile your Java code using the `javac` compiler. You need to navigate to the root directory of your project and compile the entire project. The compiler will recognize the package structure and generate corresponding `.class` files.

   For example, if your project directory structure looks like this:

```
myproject/
    ├── myapp/
    │   ├── utilities/
    │   │   ├── MyClass1.java
    │   │   ├── MyClass2.java
    │   │   └── ...
    │   └── ...
    └── ...
```

   Navigate to `myproject` and compile your code:

   javac myapp/utilities/MyClass1.java myapp/utilities/MyClass2.java

   The compiled `.class` files will be stored in the package directory.

4. Access Classes and Interfaces:
   - In your Java code, you can access the classes and interfaces within the package by using an `import` statement. Import the classes/interfaces you want to use from the package.

   For example:

   import myapp.utilities.MyClass1;

```
import myapp.utilities.MyClass2;

public class Main {
    public static void main(String[] args) {
        MyClass1 instance1 = new MyClass1();
        MyClass2 instance2 = new MyClass2();

        // Use the instances and their methods
    }
}
```

By specifying the `import` statements, you can access and use the classes and interfaces defined in the package in your code.

This approach allows you to logically group related classes and interfaces into packages, making your code more organized and maintainable. It also helps avoid naming conflicts and improves code reusability.

Module – 4

Q7.a What is the use of multiple catch statement in exception handling? Discuss with a java program.
A: In Java, multiple `catch` statements are used in exception handling to handle different types of exceptions that might occur within a try-catch block. Each `catch` block is responsible for catching and handling a specific type of exception, allowing you to write custom code for different exception scenarios.

The use of multiple `catch` statements ensures that your code can respond appropriately to various types of exceptions. This is important for making your code robust and providing specific error handling for different exceptional situations.

Example: Here is a Java program that demonstrates the use of multiple `catch` statements

```
public class MultipleCatchBlockExample {

        public static void main(String[] args) {

                try{
                        int a[]=new int[5];
                        a[5]=30/0;
                       }
                       catch(ArithmeticException e)
                    {
                                System.out.println("Arithmetic Exception occurs");
                               }
                       catch(ArrayIndexOutOfBoundsException e)
                    {
                                System.out.println("ArrayIndexOutOfBounds Exception occurs");
                               }
                       catch(Exception e)
```

```
                    {
                                System.out.println("Parent Exception occurs");
                               }
                        System.out.println("rest of the code");
        }
}
```

Output:
Arithmetic Exception occurs

Explanation:

- Inside try we had written some code which will give some runtime error.
- There are three catches.
- First one is for Arithmetic Exception
- Second one will detect out of bound in array
- Lastly is the parent class for all Exception.

Q7.b Weather try block can be nested in Java? If yes demonstrate with the help of a java program.
A: Yes, in Java, you can nest try blocks within other try blocks to handle exceptions in a nested manner. This allows you to handle exceptions at different levels of your code, providing more fine-grained error handling.

Example: Here is an example of nesting try blocks in Java

```
public class NestedTryInMainExample {
   public static void main(String[] args) {
      try {
         int[] numbers = {1, 2, 3};
         int result = 0;

         try {
            for (int i = 0; i <= numbers.length; i++) {
               try {
                  result += 10 / (numbers[i] - 2);
               } catch (ArithmeticException innerArithmeticException) {
                  System.out.println("Inner Try: ArithmeticException caught: Division by zero");
               }
            }
         } catch (ArrayIndexOutOfBoundsException innerArrayIndexException) {
            System.out.println("Middle Try: ArrayIndexOutOfBoundsException caught: Array index out of bounds");
         }

         System.out.println("Result: " + result);
      } catch (ArithmeticException outerArithmeticException) {
         System.out.println("Outer Try: ArithmeticException caught: " + outerArithmeticException.getMessage());
      } catch (Exception outerException) {
         System.out.println("Outer Try: General Exception caught: " + outerException.getMessage());
      }

      System.out.println("Program continues...");
   }
}
```

Output:
Inner Try: ArithmeticException caught: Division by zero
Middle Try: ArrayIndexOutOfBoundsException caught: Array index out of bounds
Result: 0
Program continues...

Explanation:

- We have a nested structure of `try` blocks entirely within the `main` method.
- The innermost `try` block handles `ArithmeticException` during division, displaying an error message. The middle `try` block handles `ArrayIndexOutOfBoundsException` when the array index is out of bounds.
- The outer `try` block in the `main` method handles `ArithmeticException` and a more general `Exception`.
- When you run the program, it triggers an `ArithmeticException` during division and an `ArrayIndexOutOfBoundsException`, and each corresponding `catch` block displays the associated error message. Finally, the program continues to execute.

Q8.a Write a java program which uses throws keyword for handling exception.
A: In Java, the `throws` keyword is used to declare that a method might throw a particular type of exception, but the actual exception handling is deferred to the calling method. This means that the method using the `throws` keyword is indicating the possibility of an exception, and it is the responsibility of the caller to handle the exception using a `try-catch` block or propagate it to its caller.

Example: Here is an example of a Java program that uses the `throws` keyword to handle exceptions:

```
import java.io.IOException;
public class Main{

   void m()throws IOException{
      throw new IOException("device error");
   }

   void n()throws IOException{
      m();
   }

   void p(){
      try{
         n();
      }
      catch(Exception e){System.out.println("exception handled");}
   }

   public static void main(String args[]){
      Main obj=new Main();
      obj.p();
      System.out.println("normal flow...");
   }
}
```

Output:

exception handled
normal flow...

Explanation:

The key points to note in this code are:
- The `throws` clause is used to declare that a method might throw a checked exception.
- When a method calls another method that might throw an exception, it can either handle the exception using a `try-catch` block or propagate it to its caller using the `throws` clause.
- In this code, the `IOException` is thrown in the `m()` method and is caught in the `catch` block in the `p()` method.
- The program continues executing after the `catch` block.

Q8.b How to create a custom exception class in Java? Demonstrate using a java program.
A: In Java, you can create a custom exception class by extending the `Exception` class or one of its subclasses (e.g., `RuntimeException`). This allows you to define your own exception types with specific behavior and additional fields if needed.

Example

```
class MyCustomException extends Exception {
   public MyCustomException(String message) {
      super(message);
   }
}

public class CustomExceptionExample {
   public static void main(String[] args) {
      try {
         int result = divide(10, 0);
         System.out.println("Result: " + result);
      } catch (MyCustomException e) {
         System.out.println("Custom Exception caught: " + e.getMessage());
      }
   }

   static int divide(int dividend, int divisor) throws MyCustomException {
      if (divisor == 0) {
         throw new MyCustomException("Custom Exception: Division by zero is not allowed");
      }
      return dividend / divisor;
   }
}
```

Output:
Custom Exception caught: Custom Exception: Division by zero is not allowed

In this program:

- We create a custom exception class `MyCustomException` that extends the built-in `Exception` class. This custom exception class takes a message as an argument in its constructor and passes it to the superclass constructor using `super(message)`.

- The `divide` method checks if the divisor is zero and, if so, throws a `MyCustomException` with a custom error message indicating that division by zero is not allowed.
- In the `main` method, we call the `divide` method with arguments that lead to division by zero.
- The `catch` block in the `main` method catches the `MyCustomException` and prints the custom error message.

By creating a custom exception class, you can define and throw exceptions that are specific to your application's requirements, making your code more organized and easier to understand. This is particularly useful when you want to handle exceptional cases in a way that is meaningful to your application's logic.

Module – 5

Q9.a Write a Java applet program which handles keyboard event.
A: Here's a Java program that creates a simple Swing application to handle keyboard events:

```
import javax.swing.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class KeyboardEventExample {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Keyboard Event Example");

      // Create a text area to display the pressed keys
      JTextArea textArea = new JTextArea(10, 30);
      textArea.setEditable(false);
      frame.add(textArea);

      // Add a key listener to the text area
      textArea.addKeyListener(new KeyListener() {
         @Override
         public void keyTyped(KeyEvent e) {
            // Key typed event (key press and release)
            char keyChar = e.getKeyChar();
            textArea.append("Key Typed: " + keyChar + "\n");
         }

         @Override
         public void keyPressed(KeyEvent e) {
            // Key pressed event
            int keyCode = e.getKeyCode();
            textArea.append("Key Pressed: " + KeyEvent.getKeyText(keyCode) + "\n");
         }

         @Override
         public void keyReleased(KeyEvent e) {
            // Key released event
            int keyCode = e.getKeyCode();
            textArea.append("Key Released: " + KeyEvent.getKeyText(keyCode) + "\n");
         }
```

```
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

In this program:

- We create a Swing `JFrame` to serve as the main window.
- A `JTextArea` component is added to the frame to display the keyboard events.
- We add a `KeyListener` to the text area to handle key events, including `keyTyped`, `keyPressed`, and `keyReleased`. Each event displays information about the key that triggered it in the text area.
- The program is designed to run as a standalone Java application, not as an applet. It creates a graphical user interface (GUI) window to handle keyboard events using Swing components.

To run this program, make sure you have the Java Development Kit (JDK) installed on your computer. Compile and run the program using the following commands:

```
javac KeyboardEventExample.java
java KeyboardEventExample
```

Q9.b Explain the method involve in life cycle of an applet.

A: In Java, the life cycle of an applet is managed by a set of methods provided by the `Applet` class and is divided into several stages. Here are the key methods that are involved in the life cycle of an applet:

1. `init()` Method:
   - The `init()` method is called when an applet is first created. This method is responsible for initializing the applet and is typically used for one-time setup tasks.
   - It is called after the applet's constructor and before any other methods are invoked.
   - Common tasks in the `init()` method include setting up user interface components, initializing variables, and performing necessary setup.

2. `start()` Method:
   - The `start()` method is called after the `init()` method and is used to start the execution of the applet.
   - It is called when the applet is first displayed on the web page or when the user revisits the page after minimizing it.
   - You can use the `start()` method to initiate any activities or animations that need to run while the applet is active.

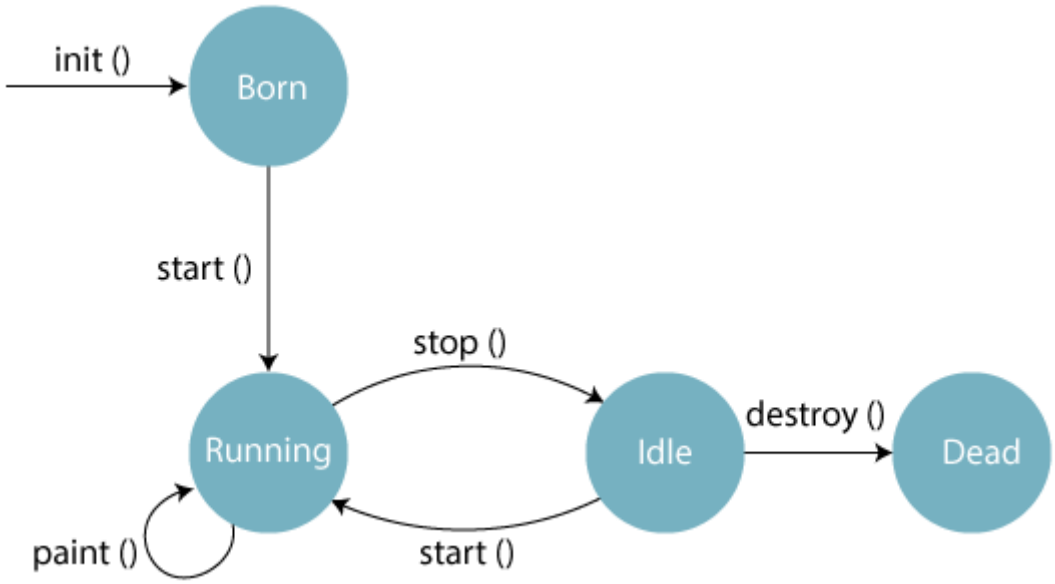
3. `paint()` Method:
   - The `paint()` method is responsible for rendering the applet's graphical content on the screen.
   - It is automatically called whenever the applet needs to be redrawn, such as when it's first displayed, when the user resizes the browser window, or when another window temporarily obscures the applet.
   - You should override this method to specify how your applet should be drawn.

4. `stop()` Method:
   - The `stop()` method is called when the applet is no longer in view, typically because the user has navigated away from the web page containing the applet.
   - You can use this method to suspend any ongoing activities or animations.
   - It is important to release any resources or stop any threads that were started in the `start()` method to prevent unnecessary resource consumption.

5. `destroy()` Method:
   - The `destroy()` method is called when the applet is about to be removed from memory.
   - You can use this method to perform cleanup tasks, such as releasing resources, closing files, and stopping threads that were initiated during the applet's execution.

6. `stop()` and `destroy()` Sequencing:
   - The `stop()` method is called when the user navigates away from the applet's page or minimizes the browser, while the `destroy()` method is called when the applet is about to be unloaded.
   - It is important to ensure that any resources started in the `start()` method are properly stopped in the `stop()` method and released in the `destroy()` method to avoid memory leaks.

The life cycle of an applet can be summarized as follows:
- `init()`: Initialize the applet.
- `start()`: Start execution.
- `paint()`: Draw content on the screen.
- `stop()`: Suspend execution.
- `destroy()`: Cleanup and resource release.



Q10.a How JButton class is used in swings? Explain

A: In Swing, the `JButton` class is used to create and manage button components, which are interactive elements that users can click to trigger actions in a graphical user interface (GUI) application. `JButton` is part of the Java Abstract Window Toolkit (AWT) and Swing library and is commonly used to create push buttons, toggle buttons, and radio buttons. Here's an explanation of how to use the `JButton` class in Swing:

1. Import the Required Packages:

```
import javax.swing.JButton;

import javax.swing.JFrame;

import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
```

2. Create a `JButton` Instance:

```
JButton myButton = new JButton("Click Me");
```

3. Add the Button to a Container:

```
JFrame frame = new JFrame("Button Example");
frame.add(myButton);
```

4. Register Action Listeners:

```
myButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      //Code
   }
});
```

5. Set Layout and Display the GUI:

```
frame.setLayout(new FlowLayout());
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(300, 150);
frame.setVisible(true);
```

6. Handle Button Clicks:

When the user clicks the button, the action defined in the `actionPerformed` method of the action listener is executed. In this example, it displays a message dialog with the "Button clicked!" message.

Example:

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class JButtonExample {
   public static void main(String[] args) {
```

```
        JFrame frame = new JFrame("Button Example");
        JButton myButton = new JButton("Click Me");

        myButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Button clicked!");
            }
        });

        frame.add(myButton);
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 150);
        frame.setVisible(true);
    }
}
```

This example creates a simple Swing GUI with a `JButton` that displays a message when clicked. The `JButton` class provides a wide range of customization options for button appearance and behavior. You can set icons, tooltips, and other properties to tailor the button to your application's needs.

Q10.b Write a Java program to display a frame using JFrame class.

A: To create a basic Java program that displays a frame using the `JFrame` class, follow these steps:

1. Import the necessary packages:

```
import javax.swing.JFrame;
```

2. Create a `JFrame` instance:

```
JFrame frame = new JFrame("My First Frame");
```

3. Set the default close operation for the frame to ensure the application terminates when the frame is closed:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4. Set the size of the frame (optional):

```
frame.setSize(400, 300);
```

5. Make the frame visible:

```
frame.setVisible(true);
```

Here's a complete Java program that creates and displays a simple frame:

```
import javax.swing.JFrame;

public class SimpleFrameExample {
    public static void main(String[] args) {

        JFrame frame = new JFrame("My First Frame");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setSize(400, 300);

        frame.setVisible(true);
    }
}
```

When you run this program, it will create a JFrame with the title "My First Frame" and a default size of 400x300 pixels. You can resize, minimize, maximize, and close the frame as needed. The program will terminate when you close the frame because we've set the default close operation to `JFrame.EXIT_ON_CLOSE`.

--- END OF DOCUMENT ---