Assignment 19.

Part A:

1. How to Create Your Own Package

A package in Java is used to group related classes and interfaces together. It helps in organizing code, avoiding naming conflicts, and providing access protection.

Why We Use Packages:

• To organize classes logically
• To avoid class name conflicts
• To provide controlled access
• To improve code maintainability

Syntax:

```
package mypackage;
```

Example:

```
package mypackage;

public class Student {
   public void display() {
      System.out.println("Hello from Student class");
   }
}
```

Steps in Eclipse:

1. Right click on src folder
2. Select New → Package
3. Give package name
4. Create class inside package

2. What is a Marker Interface?

A Marker Interface is an empty interface (without methods). It is used to provide special instructions to the JVM.

Example:

```
import java.io.Serializable;

public class Student implements Serializable {
   int id;
}
```

Common Marker Interfaces:
• Serializable
• Cloneable
• RandomAccess

3. Serialization and Deserialization

Serialization is the process of converting an object into a byte stream so it can be saved to a file or sent over a network.

Deserialization is the process of converting a byte stream back into an object.

Example:

```
import java.io.*;

class Student implements Serializable {
   int id = 101;
}

FileOutputStream fos = new FileOutputStream("data.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(s);

FileInputStream fis = new FileInputStream("data.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
Student s2 = (Student) ois.readObject();
```

4. What is Transient Keyword?

The transient keyword is used to prevent a variable from being serialized.

Example:

```
class Student implements Serializable {
   int id = 101;
   transient String password = "abc123";
}
```

After deserialization, the transient variable becomes null because it was not saved.

Real-time Use Cases:
• Password
• OTP
• Security tokens
• Temporary session data

5. Difference Between Interface and Abstract Class

Feature	Interface	Abstract Class
Methods	Only abstract methods (before Java 8)	Abstract + Concrete methods
Variables	public static final	Can have normal variables
Constructor	No	Yes
Multiple Inheritance	Supported	Not Supported
Access Modifiers	public only	private, protected allowed

Real-Time Example:

Interface Example (Vehicle Rules):

```
interface Vehicle {
    void start();
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts with key");
    }
}
```

Abstract Class Example (Bank Account):

```
abstract class BankAccount {
    int accountNumber;

    void deposit() {
        System.out.println("Deposit money");
    }

    abstract void withdraw();
}
```

Key Difference:
- Interface defines WHAT to do (contract)
- Abstract class defines WHAT + partial HOW (blueprint)

Part B:

1. Program to Handle Divide-by-Zero Exception

This program demonstrates how to handle ArithmeticException when dividing a number by zero.

```
public class DivideByZeroExample {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int result = a / b;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero");
        }
    }
}
```

2. Program to Handle Multiple Exceptions

This program demonstrates handling multiple exceptions using multiple catch blocks.

```
public class MultipleExceptionExample {
    public static void main(String[] args) {
        try {
            int arr[] = new int[5];
            arr[10] = 50; // ArrayIndexOutOfBoundsException
            int num = 10 / 0; // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception occurred");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out Of Bounds Exception occurred");
        } catch (Exception e) {
            System.out.println("General Exception occurred");
        }
    }
}
```

3. Program to Create Your Own Custom Exception

This program demonstrates how to create and use a custom exception.

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

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            int age = 15;
            if (age < 18) {
                throw new InvalidAgeException("Age must be 18 or above");
            }
        } catch (InvalidAgeException e) {
            System.out.println("Custom Exception Caught: " + e.getMessage());
        }
    }
}
```

4. Program to Demonstrate try-catch-finally Block

This program demonstrates the use of finally block which always executes.

```
public class TryCatchFinallyExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught");
        } finally {
            System.out.println("Finally block always executes");
        }
    }
}
```

5. Program to Demonstrate Nested Try Blocks

This program demonstrates nested try blocks where one try block is inside another.

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            int arr[] = new int[5];
            try {
                arr[10] = 50;
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Array index out of bounds");
            }
            int num = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Outer catch: Arithmetic exception");
        }
    }
}
```