USN | | | | | | | | | | |

Internal Assessment Test 2 – December 2024

Sub:	Object Oriented Programming with JAVA					Sub Code:	BCS306A	Branch:	AIDS & CSE(AIDS)
Date:	14/12/2024	Duration:	90 minutes	Max Marks:	50	Sem/Sec:	III -A, B & C		OBE

		Answer any FIVE FULL Questions	MARKS	CO	RBT
1	a	Define package. Explain the steps involved in creating a user-defined package with an example.	5	4	L2

ANWER:

In Java, a package is a namespace that organizes a set of related classes and interfaces. It helps to group related classes together, making the code more modular and manageable. Packages also prevent name conflicts and allow for better access control.

Packages in Java can be:

- Built-in (like java.util, java.io, etc.)
- User-defined (created by developers to organize their own classes)

A user-defined package is a package that you create to organize your own classes.

Steps to Create a User-Defined Package in Java

To create a user-defined package in Java, follow these steps:

1. Create the Package Directory

You need to create a directory structure corresponding to the package name. The directory structure will represent the package.

For example, if you want to create a package called mypackage, you will create a directory named mypackage.

2. Define the Package

In your Java class, define the package at the top of the file using the package keyword, followed by the package name.

3. Write Classes Inside the Package

Create Java classes inside the package directory. These classes will belong to the package.

4. Compile the Classes

Compile the Java classes with the javac command. Make sure to use the correct directory structure, as the javac compiler needs to know where the package is located.

5. Use the Package in Your Program

In another Java file, you can import and use the classes from your user-defined package using the import keyword.

Example: Creating and Using a User-Defined Package in Java

1. Create the Package Directory

First, create a directory called mypackage (the name of the package).

2. Create Classes in the Package

File 1: mypackage/Hello.java

```
// Hello.java
package mypackage;

public class Hello {
    public void greet() {
        System.out.println("Hello from the Hello class!");
    }
}
```

File 2: mypackage/Goodbye.java

```
// Goodbye.java
package mypackage;
```

```
public class Goodbye {
    public void sayGoodbye() {
        System.out.println("Goodbye from the Goodbye class!");
    }
}
```

3. Create a Program to Use the Package

File 3: Main.java (This file will use the mypackage package)

```
// Main.java
import mypackage.Hello; // Import Hello class from mypackage
import mypackage.Goodbye; // Import Goodbye class from mypackage

public class Main {
    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.greet(); // Calls the greet method from Hello class

        Goodbye goodbye = new Goodbye();
        goodbye.sayGoodbye(); // Calls the sayGoodbye method from Goodbye class
    }
}
```

	b	Define an exception. What are the key terms used in exception handling? Explain.	5	4	L2

ANSWER:

An **exception** in Java is an event that disrupts the normal flow of the program's execution. It is an object that wraps an error, and it can occur at runtime when the program encounters some unusual or unexpected situation, such as trying to divide by zero, accessing an array out of bounds, or trying to open a file that doesn't exist. Exceptions in Java are handled using a mechanism called **exception handling**, which allows a program to catch and handle errors without crashing.

When an exception occurs, Java creates an **exception object** which is then passed through the call stack until it is handled by an appropriate **catch block**.

Key Terms Used in Exception Handling

Try Block

- A **try block** is used to wrap code that might throw an exception. If an exception occurs within the try block, it is caught and handled by the corresponding catch block.

Catch Block

- A **catch block** is used to handle exceptions that occur in the try block. It catches exceptions of a specific type.

Finally Block

- The **finally block** is optional and is used to execute important code (like closing a file or releasing resources) regardless of whether an exception was thrown or not. The finally block is always executed after the try and catch blocks, even if there was no exception.

Throw

		• The **throw** keyword is used to explicitly throw an exception. You can throw both checked and unchecked exceptions. **Throws** • The **throws** keyword is used in method signatures to declare that a method may throw one or more exceptions. It tells the caller of the method that they should handle or propagate the exception.			

2	a	Explain the concept of importing packages in Java and provide an example demonstrating the usage of the import statement.
		ANWER:
		In Java, packages are used to group related classes and interfaces together. **Importing a package** allows you to use the classes and interfaces that are defined in that package without needing to specify their fully qualified names (which include the package name).
		By importing a package, you can reference its classes directly, making your code cleaner and more concise. There are two types of import statements:
		1. **Single Class Import**: This imports only one specific class from a package.
		2. **Wildcard Import**: This imports all the classes from a package.
		Single Class Import:
		• To import a specific class from a package, you use the import keyword followed by the fully qualified name of the class.
		• import packageName.ClassName;
		• import java.util.Scanner; // Importing only the Scanner class from java.util package
		Wildcard Import:
		• To import all classes from a package, you use the * symbol as a wildcard.
		import packageName.*;
		import java.util.*; // Importing all classes from the java.util package
		Note that wildcard imports do not work for classes in sub-packages. For example, import java.util.*; will import all classes in java.util, but not in java.util.stream.
		When accessing classes from external libraries or different packages: Importing packages allows you to use external libraries or classes from other parts of the codebase.
			5	4	L2

	b	Demonstrate the working of a nested try block with an example. **ANSWER**: A **nested try block** in Java is a try block placed inside another try block. The inner try block is executed first, and if an exception occurs inside the inner block, it is handled by its corresponding catch block. The outer try block will execute after the inner try block has been processed, and if an exception occurs in the outer try block, it can be handled by its own catch block. **Working of a Nested Try Block** 1. The outer try block is executed first. 2. If an exception occurs in the outer try block, the outer catch block will handle it. 3. If there is an inner try block, it will be executed next. 4. If an exception occurs in the inner try block, the inner catch block will handle it. 5. Finally, a finally block (if present) will always be executed, regardless of whether an exception was thrown or not. **Syntax of a Nested Try Block** try {	5	4	L3

```
    // Outer try block
    try {
        // Inner try block
    } catch (ExceptionType1 e1) {
        // Handle exception in inner try block
    } finally {
        // Code to be executed after inner try-catch
    }
} catch (ExceptionType2 e2) {
    // Handle exception in outer try block
} finally {
    // Code to be executed after outer try-catch
}
```

EXAMPLE:

```
public class NestedTryBlockExample {

    public static void main(String[] args) {
        try {
            // Outer try block
            System.out.println("Outer try block started.");

            try {
                // Inner try block
                System.out.println("Inner try block started.");

                // Simulate an exception in the inner block
                int result = 10 / 0; // Division by zero (ArithmeticException)

                System.out.println("Inner try block ended."); // This line won't be executed
            } catch (ArithmeticException e1) {
                // Handle exception in inner try block
                System.out.println("Inner catch block: ArithmeticException caught in inner try block.");
            }

            System.out.println("Outer try block ended.");

        } catch (Exception e2) {
            // Handle exception in outer try block
            System.out.println("Outer catch block: Exception caught in outer try block.");
        } finally {
            // Finally block for outer try-catch
            System.out.println("Outer finally block executed.");
        }
    }
}
```

3	a	How do you create your own exception class? Explain with a program. **ANSWER:** In Java, you can create your own custom exception by defining a new class that extends either the **Exception** class or the **RuntimeException** class. • **Exception** is for **checked exceptions** (exceptions that must be declared in the method signature using throws). • **RuntimeException** is for **unchecked exceptions** (exceptions that are not required to be declared and occur at runtime).	10	4	L2

Steps to Create Your Own Exception Class:

1. **Define a Class** that extends the Exception (or RuntimeException) class.
2. **Provide Constructors**:
   - A no-argument constructor.
   - A constructor that accepts a custom message.
3. **Throw the Custom Exception** in your code when a specific condition occurs.

Example of a Custom Exception Class

```
// Custom Exception Class
class AgeNotValidException extends Exception {
   // Constructor that accepts a custom message
   public AgeNotValidException(String message) {
      super(message);  // Pass the message to the parent Exception class
   }
}
public class CustomExceptionExample {

   // Method that throws the custom exception if age is less than 18
   public static void validateAge(int age) throws AgeNotValidException {
      if (age < 18) {
         throw new AgeNotValidException("Age is not valid. Age must be 18 or older.");
      } else {
         System.out.println("Age is valid.");
      }
   }

   public static void main(String[] args) {
      try {
         // Test the custom exception with an invalid age
         validateAge(16);  // This will throw the custom exception
      } catch (AgeNotValidException e) {
         // Handle the custom exception
         System.out.println("Caught Exception: " + e.getMessage());
      }
   }
}
```

4	a	Discuss values() and valueOf() methods in Enumerations with suitable examples.	5	5	L2

ANSWER:

In Java, **enum** (short for "enumeration") is a special data type that represents a collection of constants. Enumerations in Java provide some built-in methods, and two important ones are **values()** and **valueOf()**. These methods are used to retrieve information about enum constants.

1. values() Method

The **values()** method is automatically generated by the Java compiler for every enum class. It returns an array of the enum constants in the order they are declared. This method is useful when you need to iterate over all the values of an enum.

Syntax:

public static EnumType[] values()

Example of values() Method:

```
// Enum to represent days of the week
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumValuesExample {
   public static void main(String[] args) {
      // Using values() method to get all enum constants
      Day[] days = Day.values();

      // Iterating through the array of enum constants
      for (Day day : days) {
         System.out.println(day);
      }
   }
}
```

2. valueOf() Method

The **valueOf()** method is a static method that converts a string (representing the name of an enum constant) into the corresponding enum constant. It is useful when you need to convert a string into an enum constant, often in scenarios like user input or data processing where enum values are represented as strings.

Syntax:

public static EnumType valueOf(String name)

Example of valueOf() Method:

```
// Enum to represent days of the week
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumValueOfExample {
   public static void main(String[] args) {
      // Using valueOf() to convert string to enum constant
      String dayName = "WEDNESDAY";
      Day day = Day.valueOf(dayName);  // Converts string to enum constant

      System.out.println("The enum constant is: " + day);

      // Example of invalid input
      try {
```

```
        String invalidDay = "FUNDAY";  // No such constant in enum
        Day invalid = Day.valueOf(invalidDay);  // This will throw IllegalArgumentException
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
  }
}
```

	b	Explain auto-boxing/unboxing in expressions.
		ANWER:
		In Java, **auto-boxing** and **auto-unboxing** are features introduced in **Java 5** that simplify the interaction between primitive types (such as int, char, etc.) and their corresponding wrapper classes (like Integer, Character, etc.).
		1. **Auto-Boxing**: The automatic conversion of a **primitive type** to its corresponding **wrapper class**.
		2. **Auto-Unboxing**: The automatic conversion of a **wrapper class** object to its corresponding **primitive type**.
		1. Auto-Boxing
		Auto-boxing refers to the automatic conversion of a **primitive type** (like int, double, char) into its corresponding **wrapper class object** (like Integer, Double, Character).
		Example of Auto-Boxing:
		public class AutoBoxingExample {
		public static void main(String[] args) {
		int num = 10;
		
		// Auto-boxing: int is automatically converted to Integer object
		Integer integerObj = num;
		
		System.out.println("Integer object: " + integerObj); // Output: 10
		}
		}
		2. Auto-Unboxing
		Auto-unboxing refers to the automatic conversion of a **wrapper class object** (like Integer, Double, Character) back to its corresponding **primitive type** (like int, double, char).
		Example of Auto-Unboxing:
		public class AutoUnboxingExample {
		public static void main(String[] args) {
		Integer integerObj = new Integer(20);
		
		// Auto-unboxing: Integer object is automatically converted to int
		int num = integerObj;
		
		System.out.println("Primitive int: " + num); // Output: 20
		}
		}
			5	5	L2

5	a	What do you mean by a thread? Explain the different ways of creating threads. **ANSWER**: A **thread** in Java is a lightweight process or a single path of execution in a program. A thread allows multiple tasks to run concurrently within a single program, which helps in improving the performance of CPU-bound or I/O-bound tasks. In Java, each thread is an instance of the Thread class, or it can be an implementation of the Runnable interface. Threads allow for **multitasking** and **parallel processing** in Java applications.	10	5	L2

Different Ways of Creating Threads in Java

There are two main ways to create a thread in Java:

1. **By Extending the Thread class**
2. **By Implementing the Runnable interface**

1. By Extending the Thread Class

In this method, you create a custom thread by extending the Thread class and overriding its run() method. The run() method defines the task to be performed by the thread.

Steps to Create a Thread by Extending Thread:

1. **Extend** the Thread class.
2. **Override** the run() method to define the task to be executed by the thread.
3. **Create an instance** of the custom thread class.
4. **Start** the thread by calling the start() method, which internally invokes the run() method.

Example:

```
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed by the thread
        System.out.println("Thread is running!");
    }

    public static void main(String[] args) {
        // Create an instance of MyThread
        MyThread t = new MyThread();

        // Start the thread
        t.start();

        // Main thread continues to execute
        System.out.println("Main thread is running!");
    }
```

} **2. By Implementing the Runnable Interface**

In this approach, you implement the Runnable interface, which requires you to define the run() method. The advantage of using Runnable is that it allows you to extend another class (since Java supports single inheritance, but you can implement multiple interfaces).

Steps to Create a Thread by Implementing Runnable:

1. **Implement** the Runnable interface.
2. **Override** the run() method to define the task to be executed by the thread.
3. **Create an instance** of Thread class, passing the Runnable object as a parameter to the Thread constructor.
4. **Start** the thread by calling the start() method.

Example:

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed by the thread
        System.out.println("Thread is running via Runnable!");
    }

    public static void main(String[] args) {
        // Create an instance of MyRunnable
        MyRunnable runnable = new MyRunnable();

        // Create a thread and pass the Runnable object to the constructor
        Thread t = new Thread(runnable);
```

```
        // Start the thread
        t.start();

        // Main thread continues to execute
        System.out.println("Main thread is running!");
    }
}
```

6	a	What is the need of thread synchronization? Explain with an example how synchronization is implemented in JAVA.	10	5	L3

ANSWER:

In a **multithreading environment**, multiple threads run concurrently and can access shared resources (such as variables, data structures, or files). When more than one thread accesses a shared resource at the same time, it can lead to **data inconsistency** or **race conditions**, where the final outcome depends on the unpredictable order of execution of the threads.

Thread synchronization is a mechanism that ensures that only one thread can access a shared resource at a time. It is crucial for maintaining the **integrity of data** and ensuring that threads do not interfere with each other when accessing shared resources.

Synchronization in Java

Java provides a **synchronized** keyword to ensure that only one thread can execute a particular method or block of code at a time.

How Synchronization Works:

- When a method is marked as synchronized, it is locked by the thread that is executing it. This prevents other threads from executing any synchronized method on the same object.
- The thread must acquire the lock before entering a synchronized method or block. If another thread is already executing a synchronized method, the second thread must wait until the first thread releases the lock.

Types of Synchronization:

1. **Method Synchronization**: Synchronize an entire method.
2. **Block Synchronization**: Synchronize only a specific block of code.

1. Synchronization Using Methods

You can synchronize the entire method by using the synchronized keyword in the method declaration.

Example: Synchronizing the increment() method

```
class Counter {
    private int count = 0;

    // Synchronized method to ensure that only one thread increments the counter at a time
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class SynchronizationExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Thread 1
```

```
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
```

} **Need for Thread Synchronization in Java**

In a **multithreading environment**, multiple threads run concurrently and can access shared resources (such as variables, data structures, or files). When more than one thread accesses a shared resource at the same time, it can lead to **data inconsistency** or **race conditions**, where the final outcome depends on the unpredictable order of execution of the threads.

Thread synchronization is a mechanism that ensures that only one thread can access a shared resource at a time. It is crucial for maintaining the **integrity of data** and ensuring that threads do not interfere with each other when accessing shared resources.

Problems Without Synchronization

If multiple threads are allowed to modify a shared resource concurrently without synchronization, it can lead to unpredictable behavior. This is known as a **race condition**.

Example of a Race Condition:

java

Copy code

```
class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class RaceConditionExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Thread 1
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
```

```
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Explanation:

- The Counter class has a shared resource: the count variable.
- Two threads, t1 and t2, increment the count variable 1000 times each.
- Since the increment() method is not synchronized, there is a chance that both threads will read the value of count at the same time, modify it, and then write the new value back. This can lead to the count variable being incremented incorrectly.

Possible Output (due to race condition):

arduino
Copy code
Final count: 1485 // Expected: 2000, but due to race condition, the result is incorrect.

Solution: Thread Synchronization

To solve this problem, we use **synchronization**. Synchronization ensures that only one thread can access a critical section (the code that modifies the shared resource) at any given time.

Synchronization in Java

Java provides a **synchronized** keyword to ensure that only one thread can execute a particular method or block of code at a time.

How Synchronization Works:

- When a method is marked as synchronized, it is locked by the thread that is executing it. This prevents other threads from executing any synchronized method on the same object.
- The thread must acquire the lock before entering a synchronized method or block. If another thread is already executing a synchronized method, the second thread must wait until the first thread releases the lock.

Types of Synchronization:

1. **Method Synchronization**: Synchronize an entire method.
2. **Block Synchronization**: Synchronize only a specific block of code.

---

1. Synchronization Using Methods

You can synchronize the entire method by using the synchronized keyword in the method declaration.

Example: Synchronizing the increment() method

java
Copy code

```
class Counter {
    private int count = 0;
```

```
    // Synchronized method to ensure that only one thread increments the
counter at a time
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class SynchronizationExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Thread 1
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Explanation:

- The increment() method is marked as synchronized, so only one thread can execute this method at a time.
- When one thread is executing the increment() method, the other thread has to wait until the first thread finishes and releases the lock.

Expected Output:

yaml
Copy code
Final count: 2000

Now, the race condition is avoided, and the final count is as expected.

---

2. Synchronization Using Code Blocks

Instead of synchronizing the entire method, you can synchronize specific blocks of code using the synchronized keyword within a method. This approach is more efficient if only part of the method requires synchronization.

Example: Synchronizing a block of code inside the method

```
class Counter {
    private int count = 0;
```

```
    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}

public class SynchronizationBlockExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Thread 1
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        // Thread 2
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

CI **CCI** **HoD**