Internal Assessment Test 2- DEC. 2024

SCHEME & SOLUTION

Sub:	Object Oriented Programming with Java	Sub Code:	BCS306A	Branch:	ISE

	Answer any FIVE FULL questions	MARKS	CO	RBT
1	➢ Define Package. Explain with an Example the steps Involved in Creating User Defined Packages.	2+4+4 [10]	CO4	L1

In Java, a **package** is a namespace that organizes a set of related classes and interfaces. It provides a way to group related classes, interfaces, and sub-packages into a single namespace, which helps in avoiding name conflicts and improves modularity. A package can be used to bundle similar classes or functionality together, making the code more maintainable and easier to organize.

There are two types of packages in Java:

1. **Built-in Packages**: These are the packages provided by the Java API, like java.util, java.io, java.lang, etc.
2. **User-defined Packages**: These are packages that are created by users to organize their classes and other resources.

Steps to Create a User-Defined Package:

The steps involved in creating and using a user-defined package in Java are as follows:

Step 1: Create a Package

To create a package, you use the package keyword followed by the package name at the beginning of the Java source file. For example, we can create a package called com.example.mypackage.

java

Copy code

```
// File: MyClass.java
package com.example.mypackage;  // Define the package

public class MyClass {
    public void display() {
        System.out.println("This is a user-defined package.");
    }
}
```

Step 2: Save the Java File

Save the MyClass.java file in the appropriate directory that matches the package name. For example, for the package com.example.mypackage, the directory structure should be:

markdown

Copy code

```
com/
  example/
    mypackage/
      MyClass.java
```

Step 3: Compile the Package

To compile the Java file, navigate to the root directory (com in this case), and use the javac command:

bash

Copy code

```
javac com/example/mypackage/MyClass.java
```

This will create the compiled bytecode file MyClass.class inside the com/example/mypackage/ directory.

Step 4: Use the User-Defined Package

Once the package is created, you can import it into other Java programs and use the classes defined within that package. For example, in another class, you can import and use MyClass.

java

Copy code

```
// File: TestPackage.java
import com.example.mypackage.MyClass;  // Import the class from the user-defined package

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();  // Calling the method from MyClass
    }
}
```

Step 5: Compile and Run the Program

First, compile the TestPackage.java file:

bash

Copy code

```
javac TestPackage.java
```

Then, run the program using the java command:

bash

Copy code

```
java TestPackage
```

The output will be:

kotlin

Copy code

```
This is a user-defined package.
```

Full Example of Creating and Using a User-Defined Package

1. **Step 1: Create MyClass.java (User-Defined Package)**

java

Copy code

```
// File: com/example/mypackage/MyClass.java
package com.example.mypackage;

public class MyClass {
    public void display() {
        System.out.println("This is a user-defined package.");
    }
}
```

2. **Step 2: Create TestPackage.java (Main Program to Use the Package)**

java

Copy code

```
// File: TestPackage.java
import com.example.mypackage.MyClass;   // Importing the user-defined package

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();  // Calling the method from MyClass
    }
}
```

3. **Step 3: Directory Structure**

markdown

Copy code

```
com/
  example/
    mypackage/
      MyClass.java
TestPackage.java
```

4. **Step 4: Compile and Run**

bash

Copy code

```
javac com/example/mypackage/MyClass.java  // Compile MyClass
javac TestPackage.java  // Compile TestPackage

java TestPackage  // Run the program
```

2 Explain Different Types of Exception and ways of Handling an Exception with an Example.

2+3+5 [10] CO4 L2

Types of Exceptions in Java and Ways to Handle Exceptions

In Java, **exceptions** are events that disrupt the normal flow of the program's execution. These are objects that describe runtime errors. Exceptions are of two main categories:

1. **Checked Exceptions**
2. **Unchecked Exceptions**

1. Checked Exceptions

Checked exceptions are exceptions that are checked at compile time. The compiler ensures that these exceptions are either handled using a try-catch block or declared using the throws keyword in the method signature. If you fail to handle these exceptions, your program will not compile.

- Examples of checked exceptions:
  - IOException (for input-output errors)
  - SQLException (for database-related errors)
  - ClassNotFoundException (when a class is not found)

2. Unchecked Exceptions

Unchecked exceptions are exceptions that are not checked at compile time but at runtime. These are subclasses of RuntimeException. You are not forced to handle or declare them, but it's still good practice to do so.

- Examples of unchecked exceptions:
  - ArithmeticException (for divide-by-zero errors)

  - NullPointerException (when an object is null and you try to access its methods/fields)
  - ArrayIndexOutOfBoundsException (when an invalid index is used to access an array)

Exception Hierarchy in Java

- Throwable is the superclass of all errors and exceptions in Java.
  - Error (for serious system errors like OutOfMemoryError or StackOverflowError)
  - Exception (for all exceptions)
    - RuntimeException (for unchecked exceptions)
    - IOException, SQLException, etc. (for checked exceptions)

Ways to Handle Exceptions in Java

Exceptions in Java can be handled using the following approaches:

1. **Using Try-Catch Block**
2. **Using Throws Clause**
3. **Using Finally Block**

1. Using Try-Catch Block

The try block contains the code that might throw an exception, and the catch block handles the exception. You can have multiple catch blocks to handle different types of exceptions.

Syntax:

java

Copy code

```
try {
   // Code that may throw an exception
} catch (ExceptionType1 e1) {
   // Handle exception of type ExceptionType1
} catch (ExceptionType2 e2) {
   // Handle exception of type ExceptionType2
}
```

Example:

java

Copy code

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

Output:

vbnet

Copy code

```
Error: Cannot divide by zero.
```

2. Using Throws Clause

The throws keyword is used to declare that a method may throw one or more exceptions. This allows the caller of the method to handle the exception. It is typically used for checked exceptions.

Syntax:

java

Copy code

```
public void method() throws ExceptionType1, ExceptionType2 {
    // Code that may throw exceptions
}
```

Example:

java

Copy code

```
import java.io.*;

public class ThrowsExample {
    // Method that may throw IOException
    public static void readFile(String filename) throws IOException {
        FileReader file = new FileReader(filename);  // May throw
FileNotFoundException
        BufferedReader fileInput = new BufferedReader(file);
        fileInput.readLine();
        fileInput.close();
    }

    public static void main(String[] args) {
        try {
            readFile("non_existent_file.txt");
        } catch (IOException e) {
            System.out.println("File not found or read error: " + e.getMessage());
        }
    }
}
```

Output:

arduino

Copy code

```
File not found or read error: non_existent_file.txt (No such file or directory)
```

In the above code, the readFile method declares that it may throw an IOException. The caller (in main) handles the exception with a try-catch block.

3. Using Finally Block

The finally block is used to execute code that should run regardless of whether an exception occurs or not. It's commonly used for cleanup activities, like closing file streams, database connections, etc.

Syntax:

java

Copy code

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle exception
} finally {
    // Code that will execute regardless of whether an exception occurred or not
}
```

Example:

java

Copy code

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Trying to divide...");
            int result = 10 / 2;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("This block will always execute.");
        }
    }
}
```

Output:

vbnet

Copy code

```
Trying to divide...
Result: 5
This block will always execute.
```

If an exception had occurred (like dividing by 0), the finally block would still have executed.

Example of Multiple Catch Blocks

You can have multiple catch blocks to handle different types of exceptions. Each catch block can handle a specific exception.

java

Copy code

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[2];
            arr[3] = 5;  // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds.");
        } catch (Exception e) {
            System.out.println("An unexpected error occurred.");
        }
    }
}
```

Output:

sql

Copy code

```
Array index out of bounds.
```

In this case, the catch block for ArrayIndexOutOfBoundsException is executed because that is the exception thrown by the code.

Summary of Exception Handling

- **Checked Exceptions**: These are exceptions that must be either caught or declared using the throws keyword. Examples include IOException, SQLException.

- **Unchecked Exceptions**: These are exceptions that do not need to be explicitly handled. They inherit from RuntimeException. Examples include ArithmeticException, NullPointerException.
- **Handling Exceptions**:
  - **Try-Catch Block**: Used to catch and handle exceptions.
  - **Throws Clause**: Used to declare exceptions that a method might throw, making it the responsibility of the caller to handle the exceptions.
  - **Finally Block**: Used for cleanup code, always executed regardless of whether an exception occurred.

By using exception handling properly, you can make your Java programs more robust and user-friendly, ensuring they behave predictably even when errors occur.

3	Explain different ways of Creating Thread with a neat Programming Example.	5+5 [10]	CO5	L2

In Java, a **thread** is a lightweight process that allows for concurrent execution of two or more parts of a program. Each thread has its own execution path, but they share resources like memory. Java provides two primary ways to create and manage threads:

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

Both approaches are commonly used in Java applications to perform tasks asynchronously or concurrently.

1. Creating a Thread by Extending the Thread Class

In this approach, you extend the Thread class and override its run() method. The run() method contains the code that will be executed by the thread when it starts.

Steps:

- Create a subclass of the Thread class.
- Override the run() method with the code that should execute in the thread.
- Create an instance of the subclass and invoke its start() method to begin execution of the thread.

Example:

java

Copy code

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

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

        // Start the thread
        thread.start();
        // Main thread continues executing
        System.out.println("Main thread is running");
    }
}
```

Explanation:

- The MyThread class extends the Thread class and overrides the run() method.
- In the main() method, we create an instance of MyThread and call the start() method to begin the execution of the thread.
- The start() method internally invokes the run() method in a new thread of execution.

Output:

arduino

Copy code

```
Thread is running by extending the Thread class
Main thread is running
```

Note: The output may vary because of thread scheduling. The "Thread is running by extending the Thread class" message might appear before or after the "Main thread is running" message depending on how the threads are scheduled.

2. Creating a Thread by Implementing the Runnable Interface

The second way to create a thread is by implementing the Runnable interface, which is more flexible. This approach allows you to separate the task (the code to be executed) from the thread management, as Runnable is a functional interface and can be used with different types of thread management.

Steps:

- Implement the Runnable interface by providing an implementation for the run() method.
- Pass the Runnable instance to a Thread object.
- Start the thread using the start() method.

Example:

java

Copy code

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

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

        // Create a thread and pass the Runnable object
        Thread thread = new Thread(myRunnable);
        // Start the thread
        thread.start();

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

Explanation:

- The MyRunnable class implements the Runnable interface and provides the code to be executed in the run() method.
- In the main() method, we create a Thread object and pass the Runnable instance to it.
- Calling the start() method begins the execution of the thread, which in turn calls the run() method.

Output:

kotlin

Copy code

```
Thread is running by implementing the Runnable interface
Main thread is running
```

Again, the order of the messages may vary depending on how the threads are scheduled.

3. Creating a Thread Using Lambda Expression (Java 8 and above)

Since Java 8, you can use **lambda expressions** to implement the Runnable interface more concisely. This eliminates the need for a separate class or an anonymous class to implement the run() method.

Example:

java

Copy code

```
public class LambdaThreadExample {
    public static void main(String[] args) {
        // Using a lambda expression to create a thread
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running using a lambda expression");
        });

        // Start the thread
        thread.start();

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

Explanation:

- The lambda expression () -> {} provides the implementation for the run() method of the Runnable interface.
- The lambda expression is passed directly to the Thread constructor to create the thread.

Output:

arduino

Copy code

```
Thread is running using a lambda expression
Main thread is running
```

4. Creating a Thread Using Anonymous Class

Another concise way to create a thread is by using an **anonymous class** to implement the Runnable interface. This is a common approach when you want to avoid creating a separate class just for implementing Runnable.

Example:

java

Copy code

```
public class AnonymousThreadExample {
    public static void main(String[] args) {
        // Create a thread using an anonymous class that implements Runnable
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Thread is running using an anonymous class");
            }
        });

        // Start the thread
        thread.start();

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

Explanation:

- An anonymous class is used to implement the Runnable interface directly inside the Thread constructor.
- The run() method of the anonymous class contains the task to be executed in the new thread.

Output:

arduino

Copy code

```
Thread is running using an anonymous class
Main thread is running
```

Key Differences Between the Two Methods:

Aspect	**Extending Thread Class**	**Implementing Runnable Interface**
Inheritance	Inherits from Thread class	Implements Runnable interface
Flexibility	Less flexible (can only extend one class)	More flexible (can implement multiple interfaces)
Use Case	Best for simple tasks	Best when you want to separate logic (task) from thread management
Multiple Threads	Can only extend one class, limiting multi-threading tasks	Can be used by multiple threads with a single instance of Runnable

4	Explain the concept of Autoboxing and Unboxing with an Example.	5+5 [10]	CO5	L2

Autoboxing and unboxing are two related concepts in Java that deal with the automatic conversion between primitive data types and their corresponding wrapper classes.

1. Autoboxing

Autoboxing is the automatic conversion that the Java compiler makes between primitive types and their corresponding **wrapper classes**. For example, a primitive int can be automatically converted to an Integer object by the compiler.

- **Primitive types** are simple data types like int, char, double, etc.
- **Wrapper classes** are objects that wrap the primitive data types (e.g., Integer for int, Character for char, Double for double).

2. Unboxing

Unboxing is the reverse process of autoboxing. It refers to the automatic conversion of an **object** (a wrapper class) to its corresponding **primitive type**. For example, an Integer object can be automatically converted to a primitive int.

Why Autoboxing and Unboxing are Important

Autoboxing and unboxing simplify code, making it easier to work with collections like ArrayList that can only hold objects. Without autoboxing, you would need to manually convert between primitive types and wrapper objects, making code more complex.

Example of Autoboxing and Unboxing

Autoboxing Example:

java

Copy code

```
public class AutoboxingExample {
    public static void main(String[] args) {
        // Autoboxing: Converting primitive int to Integer object
        int primitiveInt = 10;
        Integer wrappedInt = primitiveInt;  // This is autoboxing

        System.out.println("Primitive int: " + primitiveInt);
        System.out.println("Wrapped Integer: " + wrappedInt);
    }
}
```

Explanation:

- In the above example, the primitive int is automatically converted (autoboxed) into an Integer object when assigned to the wrappedInt variable.

Output:

sql

Copy code

```
Primitive int: 10
Wrapped Integer: 10
```

Unboxing Example:

java

Copy code

```
public class UnboxingExample {
    public static void main(String[] args) {
        // Autoboxing: Integer object is automatically converted to primitive int
        Integer wrappedInt = new Integer(100);  // Integer object
        int primitiveInt = wrappedInt;  // This is unboxing
        System.out.println("Wrapped Integer: " + wrappedInt);
        System.out.println("Primitive int: " + primitiveInt);
    }
}
```

Explanation:

- In this example, the Integer object wrappedInt is automatically converted (unboxed) into a primitive int when assigned to the primitiveInt variable.

Output:

sql

Copy code

```
Wrapped Integer: 100
Primitive int: 100
```

Autoboxing and Unboxing in Collections:

Autoboxing and unboxing are particularly useful when working with collections like ArrayList. Since collections can only hold objects (not primitive types), autoboxing and unboxing allow you to work with primitive types in collections seamlessly.

Example with ArrayList:

java

Copy code

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Autoboxing: primitive int is automatically boxed into Integer
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);  // Autoboxing: int to Integer

        // Unboxing: Integer is automatically converted back to primitive int
        int num = list.get(0);  // Unboxing: Integer to int

        System.out.println("Value from ArrayList: " + num);
    }
}
```

Explanation:

- In this example, when we add a primitive int value to the ArrayList<Integer>, Java automatically performs autoboxing (converting int to Integer).
- Similarly, when we retrieve the value from the list, Java automatically performs unboxing (converting Integer to int).

Output:

csharp

Copy code

```
Value from ArrayList: 10
```

Detailed Breakdown of Autoboxing and Unboxing

1. **Autoboxing**: Happens automatically when you assign a primitive value to a wrapper class object.
   - For example, when you assign an int to an Integer object, Java will automatically box the primitive value into the Integer object.

2. **Unboxing**: Happens automatically when you assign a wrapper class object to a primitive variable.
    - For example, when you assign an Integer object to an int, Java will automatically unbox the Integer into the primitive int.

Java Wrapper Classes for Primitives

Primitive Type	**Wrapper Class**
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

5	Define: a) isAlive() b)join() c)setPriority() d)isDaemon() e)sleep()	2 x 5 [10]	CO5	L1

In Java, threads are a fundamental part of concurrent programming, and there are several useful methods to control thread behavior. Let's define and explain the following thread-related methods:

a) isAlive() Method

- **Definition**: The isAlive() method is used to check if a thread is still alive (i.e., it has been started and has not yet completed its execution).
- **Syntax**:

java

Copy code

```
boolean isAlive();
```

- **Returns**:
    - true if the thread has been started and has not yet died.
    - false if the thread has not been started or has already finished execution.
- **Example**:

java

Copy code

```
public class ThreadAliveExample {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        thread.start();
        System.out.println("Thread is alive? " + thread.isAlive()); // Before sleeping
        thread.join();  // Wait for the thread to finish
        System.out.println("Thread is alive? " + thread.isAlive()); // After completion
    }
}
```

Output:

csharp

Copy code

```
Thread is alive? true
Thread is alive? false
```

---

b) join() Method

- **Definition**: The join() method allows the current thread (the thread calling join()) to wait until the thread on which join() is called has finished executing. This is useful when you want one thread to wait for another to finish before proceeding.
- **Syntax**:

java

Copy code

```
void join() throws InterruptedException;
void join(long millis) throws InterruptedException;
void join(long millis, int nanos) throws InterruptedException;
```

- **Description**:
  - join() makes the current thread wait indefinitely for the other thread to complete.
  - join(long millis) makes the current thread wait for a specific period (in milliseconds).
  - join(long millis, int nanos) allows waiting for a specified period in milliseconds and nanoseconds.
- **Example**:

java

Copy code

```
public class ThreadJoinExample {
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(() -> {
            try {
                Thread.sleep(2000);
                System.out.println("Thread 1 completed.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("Thread 2 completed.");
        });

        thread1.start();
        thread2.start();

        thread1.join(); // Wait for thread1 to finish
        thread2.join(); // Wait for thread2 to finish

        System.out.println("Both threads completed.");
    }
}
```

Output:

mathematica

Copy code

```
Thread 2 completed.
Thread 1 completed.
Both threads completed.
```

---

c) setPriority() Method

- **Definition**: The setPriority() method is used to set the priority of a thread. Java provides 10 levels of thread priority, from Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10), with the default priority being Thread.NORM_PRIORITY (5).
- **Syntax**:

java

Copy code

```
void setPriority(int priority);
```

- **Parameter**: priority is an integer value that should be between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY (10).
- **Example**:

java

Copy code

```
public class ThreadPriorityExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> System.out.println("Thread 1 with higher
priority"));
        Thread thread2 = new Thread(() -> System.out.println("Thread 2 with lower
priority"));

        thread1.setPriority(Thread.MAX_PRIORITY);  // Set thread1 priority to
maximum (10)
```

```
        thread2.setPriority(Thread.MIN_PRIORITY);  // Set thread2 priority to
minimum (1)

        thread1.start();
        thread2.start();
    }
}
```

Output (Note: The actual execution order depends on the thread scheduler, which may not strictly follow priorities):

csharp

Copy code

```
Thread 1 with higher priority
Thread 2 with lower priority
```

d) isDaemon() Method

- **Definition**: The isDaemon() method is used to check whether a thread is a **daemon thread**. Daemon threads are low-priority background threads that are automatically terminated when all non-daemon threads have finished executing.
- **Syntax**:

java

Copy code

```
boolean isDaemon();
```

- **Returns**:
  - true if the thread is a daemon thread.
  - false if the thread is not a daemon thread.
- **Example**:

java

Copy code

```
public class DaemonThreadExample {
    public static void main(String[] args) throws InterruptedException {
        Thread daemonThread = new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    System.out.println("Daemon thread running...");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        daemonThread.setDaemon(true);  // Set as a daemon thread
        daemonThread.start();

        System.out.println("Is daemon thread? " + daemonThread.isDaemon()); // Check if it's a daemon thread

        Thread.sleep(5000);  // Main thread sleeps for 5 seconds
        System.out.println("Main thread finished.");
    }
}
```

Output:

arduino

Copy code

```
Is daemon thread? true
Daemon thread running...
Daemon thread running...
Daemon thread running...
Main thread finished.
```

After the main thread finishes execution, the daemon thread is automatically terminated.

---

e) sleep() Method

- **Definition**: The sleep() method is used to pause the execution of the current thread for a specified period. The thread remains inactive (sleeps) for the given time in milliseconds (and optionally in nanoseconds).
- **Syntax**:

java

Copy code

```
static void sleep(long millis) throws InterruptedException;
static void sleep(long millis, int nanos) throws InterruptedException;
```

- **Parameters**:
    - millis: The number of milliseconds to sleep.
    - nanos: The number of nanoseconds to sleep (optional).
- **Example**:

java

Copy code

```
public class ThreadSleepExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            try {
                System.out.println("Thread started.");
                Thread.sleep(2000);  // Sleep for 2 seconds
                System.out.println("Thread resumed after 2 seconds.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        thread.start();
    }
}
```

Output:

mathematica

Copy code

```
Thread started.
(After 2 seconds)
Thread resumed after 2 seconds.
```

Explanation:

- The sleep() method causes the current thread to pause for the specified duration (in this case, 2 seconds).
- After the sleep time is over, the thread resumes execution from where it left off.

Summary of Methods

Method	Description
isAlive()	Returns true if the thread is alive (i.e., it has been started and not yet finished).
join()	Makes the current thread wait until the thread on which join() is called finishes.
setPriority()	Sets the priority of a thread (between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY).
isDaemon()	Checks if the thread is a daemon thread (which will terminate when all non-daemon threads finish).
sleep()	Pauses the execution of the current thread for a specified amount of time.

These methods provide fine control over thread execution, synchronization, and management in a multi-threaded Java program.

6	Implement a java program to create an interface resizable with methods resizeWidth(int width) and resizeHeight(int height) that allow an object to be resized. Create a class Rectangle that implements the resizable interface and implement the resize methods.	3+3+4 [10]	CO4	L3

```
interface Resizable {
    void resizeWidth(int width);
    void resizeHeight(int height);
  }
```

```
   class Rectangle implements Resizable {
    private int width;
    private int height;

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

    @Override
    public void resizeWidth(int width) {
        this.width = width;
    }

    @Override
    public void resizeHeight(int height) {
        this.height = height;
    }

    @Override
    public String toString() {
        return "Rectangle (width: " + width + ", height: " + height + ")";
    }
  }

public class ResizableDemo {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(10, 20);
        System.out.println("Original Rectangle: " + rectangle);

        // Resize the rectangle
        rectangle.resizeWidth(15);
        rectangle.resizeHeight(25);
        System.out.println("Resized Rectangle: " + rectangle);
    }
  }
```

Output:

```
Original Rectangle: Rectangle (width: 10, height: 20)
Resized Rectangle: Rectangle (width: 15, height: 25)
```