MODULE 5

1. What do you mean by a thread? Explain the different ways of creating threads.

- **A thread is a lightweight subprocess** that runs independently within a program.
- Threads enable **multitasking within a program,** allowing multiple operations to run concurrently.
- Threads share the **same memory space** but have their own execution stack and program counter.

Why Use Threads?

- To perform multiple tasks simultaneously.
- Efficient use of CPU by running tasks in parallel.
- Improved application responsiveness (e.g., GUI programs).

Ways to Create Threads in Java

There are **two main** ways to create threads in Java:

1. By Extending the Thread Class

- The Thread class in Java provides methods to create and manage threads.

- **Steps:**

1. Create a class that extends the Thread class.
2. Override the run() method to define the thread's task.
3. Create an object of the class and call the **start()** method to execute the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

```
public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create a thread object
        t1.start(); // Start the thread
    }
}
```

Op;- thread is running

2. By Implementing the Runnable Interface

- The Runnable interface is a functional interface containing the run() method.

- **Steps:**

1. Create a class that implements the Runnable interface.
2. Override the run() method to define the thread's task.
3. Pass an instance of the class to a Thread object and call the **start()** method.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable...");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
            Thread t1 = new Thread(myRunnable); // Pass Runnable object to Thread
        t1.start(); // Start the thread
    }
}
```

Op

Thread is running using Runnable…

2. What is the need of synchronization? Explain with an example how synchronization is implemented in JAVA.

- **Synchronization** in Java is the process of controlling access to **shared resources by multiple threads to prevent data inconsistency.**
- When multiple threads access a shared resource simultaneously, they may interfere with each other, causing issues like race conditions.

- **Need for Synchronization:**

- Ensures data consistency.
- Prevents thread interference.
- Helps achieve thread safety when multiple threads access shared resources.

How Synchronization is Implemented in Java

Java provides the synchronized keyword to achieve synchronization. It can be applied to:

1. **Methods** (synchronized methods).
2. **Blocks** (synchronized blocks).

1. Synchronized Method

- Synchronizing a method ensures that only one thread can execute it at a time.
- It locks the entire object.

2. Synchronized Block

- Synchronizing only a specific block of code rather than the entire method.
- Provides finer control by locking only a particular part of the code or a specific object.

Advantages of Synchronization

1. Ensures thread safety.
2. Prevents data inconsistency and corruption.
3. Helps in achieving predictable outcomes in multithreaded applications.

Disadvantages of Synchronization

1. Slower performance due to locking overhead.
2. May lead to deadlocks if not used carefully.

Example

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

    // Synchronized method to ensure thread safety
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

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

        // Create two threads that increment the counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                counter.increment();
            }
        });

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


        // Print the final count
        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Op:- Final Count: 10

3. Discuss values() and value Of() methods in Enumerations with suitable examples.

values() and **valueOf()** Methods in **Enumerations**

In Java, an **enumeration (enum) is a special data type that contains a fixed set of constants.** The values() and valueOf() methods are two important built-in methods provided by the enum type.

1. values() Method

- **Purpose:** Returns an array of all the constants defined in the enum.
- **Usage:** Used to iterate over all enum constants.
  input : No input required.
  Return Type: Array of enum constants.
  no case sensitive

Syntax:
public static T[] values();

- **T refers to the enum type.**

Example

```
enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

public class EnumValuesExample {
    public static void main(String[] args) {
        System.out.println("Days of the Week:");

        // Using values() to get all constants
        for (Days day : Days.values()) {
            System.out.println(day);
        }
    }
}
```

```
Op:-
Days of the Week:
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

2. valueOf() Method

- **Purpose:** Returns the enum constant with the specified name.
- **Usage:** Used to get an enum constant by its name.
  input : Takes a string (name of the constant).
  return type : Single enum constant.
  it's case sensitive, name must mach as exact

Syntax:
public static T valueOf(String name);

- **name:** The exact name of the enum constant (case-sensitive).

Example :-

```
enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

public class EnumValueOfExample {
    public static void main(String[] args) {
        // Using valueOf() to get a specific constant
        Days day = Days.valueOf("FRIDAY");
        System.out.println("Selected Day: " + day);

        // Incorrect name example (throws exception)
        // Days invalidDay = Days.valueOf("Funday"); // Uncomment to see the error
    }
}
```

Op:- Selected Day: FRIDAY

4. What is multithreading? Write a program to create multiple threads in JAVA.

- **Multithreading** is a feature in Java (and other programming languages) that allows a program to **run multiple threads concurrently**, enabling it to perform several tasks at the same time.
- **A thread is the smallest unit of a CPU's execution.** Java provides **built-in support** for multithreading to perform multiple operations simultaneously, improving the efficiency of a program.

Key Advantages of Multithreading:

- **Better resource utilization:** Allows CPU to be utilized efficiently.
- **Improved performance:** Tasks can run concurrently, reducing overall time.
- **Responsive applications:** Applications with multiple threads remain responsive, even while performing long tasks (e.g., in GUI-based apps).

There are two main ways to create threads in Java:

1. By extending the Thread class.
2. By implementing the Runnable interface.

1. Creating Threads by Extending the Thread Class

Steps:

- Step 1: Create a class that extends the Thread class.
- Step 2: Override the run() method to define the task.
- Step 3: Create thread objects and call start() to begin execution.

Example Program:

```
class MyThread extends Thread {
    public void run() {
        // Task to be performed by the thread
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getId() + " Value: " + i);
            try {
                Thread.sleep(500);  // Sleep for 500ms (0.5 seconds)
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class MultithreadingExample {
    public static void main(String[] args) {
        // Create two threads
```

```
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        // Start the threads
        t1.start();
        t2.start();
    }
}
```

Explanation:

1. **Thread Class:** MyThread extends the Thread class, which provides the method run() that is overridden to define the task.
2. **start():** This method is used to begin the execution of the thread. It calls the run() method in a new thread of execution.
3. **sleep():** Inside the run() method, we use Thread.sleep(500) to pause the execution of each thread for 500 milliseconds (0.5 seconds) between printing the values.

Expected output

```
1 Value: 0
1 Value: 1
2 Value: 0
1 Value: 2
2 Value: 1
1 Value: 3
2 Value: 2
1 Value: 4
2 Value: 3
2 Value: 4
```

5. Explain with an example how inter-thread communication is implemented in JAVA.

Inter-thread communication allows **threads to communicate with each other and coordinate their activities.** In Java, this can be done using the following methods:

- **wait()**
- **notify()**
- **notifyAll()**

These methods are defined in the **Object class** and can be **used to manage the flow of execution between threads.**

Key Concepts

1. **wait():** Causes the current thread to release the lock and enter the waiting state until another thread sends a notification (either notify() or notifyAll()).
2. **notify():** Wakes up one thread that is currently waiting on the object's monitor (lock).
3. **notifyAll():** Wakes up all threads that are waiting on the object's monitor (lock).

Example

producer consumer problem

```
class SharedBuffer {
    private int item = 0;
    private boolean empty = true;

    // Producer thread method
    public synchronized void produce() throws InterruptedException {
        while (!empty) {
            wait();  // Wait if the buffer is full
        }
        item++;
        System.out.println("Produced: " + item);
        empty = false;
        notify();  // Notify consumer
    }

    // Consumer thread method
    public synchronized void consume() throws InterruptedException {
        while (empty) {
```

```
            wait();  // Wait if the buffer is empty
        }
        System.out.println("Consumed: " + item);
        empty = true;
        notify();  // Notify producer
    }
}

public class InterThreadCommunicationExample {
    public static void main(String[] args) throws InterruptedException {
        SharedBuffer buffer = new SharedBuffer();

        // Producer thread
        Thread producer = new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    buffer.produce();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    buffer.consume();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        producer.start();
        consumer.start();

        producer.join();
        consumer.join();
    }
}
```

Explanation:

1. **produce() method:** Waits if the buffer is full, produces an item, and notifies the consumer.
2. **consume() method:** Waits if the buffer is empty, consumes an item, and notifies the producer.

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

Key Points:

- wait() is used to make the thread wait when the condition is not right.
- notify() wakes up the other thread to continue execution.

6. Explain auto-boxing/unboxing in expressions.

Auto-boxing and unboxing are features introduced in Java that **automatically convert between primitive types (e.g., int, char) and wrapper classes (e.g., Integer, Character).** These conversions happen implicitly, without the programmer needing to manually wrap or unwrap the values.

1. Auto-boxing:

Auto-boxing is the automatic conversion of a **primitive type to its corresponding wrapper class.**

Example:

```
int num = 10;
Integer obj = num;  // Auto-boxing: int -> Integer
```

2. Unboxing:

Unboxing is the automatic conversion of a **wrapper class to its corresponding primitive type.**

Example:

```
Integer obj = new Integer(10);
int num = obj;  // Unboxing: Integer -> int
```

Auto-boxing and Unboxing in Expressions:

Java can automatically box and unbox objects as needed in expressions. This is particularly useful in arithmetic operations involving wrapper classes.

Example with Auto-boxing and Unboxing in an Expression:

```
public class AutoBoxingExample {
    public static void main(String[] args) {
        Integer a = 10;  // Auto-boxing: int 10 to Integer
        Integer b = 20;  // Auto-boxing: int 20 to Integer

        // Auto-unboxing for arithmetic operation
        int sum = a + b;  // Unboxing: Integer to int, then addition
```

```
        System.out.println("Sum: " + sum);
    }
}
```

Output: 30

Explanation:

- a and b are Integer objects.
- Java automatically unboxes a and b to primitive int for the addition operation.
- After the operation, the result is stored in the int variable sum.

1. **Auto-boxing:** Converting primitive types (like int, char) to corresponding wrapper classes (like Integer, Character).
2. **Unboxing:** Converting wrapper class objects back to their primitive types.
3. **Implicit in expressions:** Java handles these conversions automatically in arithmetic operations or when assigning values between primitives and wrapper objects.

7. Summarise the type wrappers supported in java

Java provides **wrapper classes for each primitive type.** These classes allow **primitives to be treated as objects,** which is necessary when working with collections like ArrayList that can only hold objects. The wrapper classes are part of the java.lang package and each wrapper class corresponds to a specific primitive type.

List of Wrapper Classes in Java:

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

Key Points about Wrapper Classes:

1. Conversion between Primitives and Objects:

- **Auto-boxing:** Converting a primitive type to its corresponding wrapper class.
- **Unboxing:** Converting a wrapper class object back to its corresponding primitive type.

2. Wrapper Class Methods:

- Each wrapper class provides utility methods such as:
- parseX() methods (e.g., Integer.parseInt() for parsing a string to a primitive int).
- valueOf() methods (e.g., Integer.valueOf() to return an Integer object).
- toString() for converting the wrapper object to a string.

3. Immutable:

- Wrapper classes are immutable, meaning their values cannot be changed once they are created.

4. Autoboxing and Unboxing:

- Java automatically converts between primitive types and their corresponding wrapper objects when needed, such as in collections or during arithmetic operations.

Examples:

- Boolean:
- Boolean trueObj = Boolean.valueOf(true);
- boolean primTrue = trueObj; // Unboxing
- Integer:
- Integer intObj = Integer.valueOf(10);
- int intPrim = intObj; // Unboxing
- Double:
- Double doubleObj = Double.valueOf(10.5);
- double doublePrim = doubleObj; // Unboxing

These wrapper classes are useful for **interacting with collections** (like List, Map) and performing various utility operations on data.

8. Develop a java prog for automatic conversion of wrapper class type into corresponding primitive type that demonstrates unboxing

Java automatically converts the values from the wrapper classes (Integer, Double, Character) to their corresponding primitive types (int, double, char) during assignment, demonstrating unboxing.

```
public class UnboxingExample {
    public static void main(String[] args) {
        // Wrapper class objects
        Integer intObj = 100;
        Double doubleObj = 10.5;
        Character charObj = 'A';

        // Unboxing: Automatic conversion from wrapper class to primitive
type
        int intPrim = intObj;  // Integer -> int
        double doublePrim = doubleObj;  // Double -> double
        char charPrim = charObj;  // Character -> char

        // Displaying the unboxed values
        System.out.println("Unboxed Integer: " + intPrim);
        System.out.println("Unboxed Double: " + doublePrim);
        System.out.println("Unboxed Character: " + charPrim);
    }
}
```

- We have three wrapper class objects: Integer, Double, and Character.
- Unboxing happens automatically when assigning these wrapper objects to corresponding primitive variables (int, double, char).
- The unboxed values are then printed.

Output:

```
Unboxed Integer: 100
Unboxed Double: 10.5
Unboxed Character: A
```

9. What is meant by thread priority? How to assign and get the thread priority?

Thread Priority in Java d**etermines the relative importance of threads during execution.** It is used by the **Thread Scheduler** to decide which thread to execute next. Threads with higher priority may be given more CPU time, though this is not guaranteed, as thread scheduling is handled by the underlying operating system.

Thread Priority Range:

- Priority Value Range:
- Thread.MIN_PRIORITY (1)
- Thread.NORM_PRIORITY (5) – Default priority for threads
- Thread.MAX_PRIORITY (10)

The priority is an integer value between 1 (lowest priority) and 10 (highest priority). By default, threads have normal priority (5).

Assigning Thread Priority:

To assign a priority to a thread, we use the **setPriority()** method.

Syntax:

Thread.setPriority(int priority);

Getting Thread Priority:

To get the priority of a thread, we use the **getPriority()** method.

Syntax:
int priority = thread.getPriority();

Example

```
public class ThreadPriorityExample {
    public static void main(String[] args) {
        // Creating threads
         Thread thread1 = new Thread(() -> System.out.println("Thread 1 is
running"));
         Thread thread2 = new Thread(() -> System.out.println("Thread 2 is
running"));
```

```
        // Assigning priorities
        thread1.setPriority(Thread.MIN_PRIORITY);  // Lowest priority (1)
        thread2.setPriority(Thread.MAX_PRIORITY);  // Highest priority (10)

        // Getting and printing the priorities
        System.out.println("Thread 1 Priority: " + thread1.getPriority());
        System.out.println("Thread 2 Priority: " + thread2.getPriority());

        // Starting threads
        thread1.start();
        thread2.start();
    }
}
```

Explanation:

1. Thread creation: We create two threads (thread1 and thread2).
2. Set priorities:
   - thread1 is assigned the lowest priority (Thread.MIN_PRIORITY).
   - thread2 is assigned the highest priority (Thread.MAX_PRIORITY).
3. Get priorities: We use getPriority() to print the priority of each thread.
4. Thread execution: Both threads are started using start().

Output:

```
Thread 1 Priority: 1
Thread 2 Priority: 10
Thread 2 is running
Thread 1 is running
```

10. creation of thread program

```
public class MyThread implements Runnable {

    // The run method that defines the code to be executed by the thread
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " i is " + i);
            try {
                // Sleep for 500 milliseconds to simulate work
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }

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

        // Create three threads, all executing the same MyThread instance
        Thread t1 = new Thread(myThread);
        Thread t2 = new Thread(myThread);
        Thread t3 = new Thread(myThread);

        // Start all threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Op:-

```
Thread-0 i is 1
Thread-1 i is 1
Thread-2 i is 1
Thread-0 i is 2
Thread-1 i is 2
Thread-2 i is 2
Thread-0 i is 3
Thread-1 i is 3
Thread-2 i is 3
Thread-0 i is 4
```

11. Main thread and child thread prog :-

```
public class Test {
    public static void main(String[] args) {
        // Creating the child thread
        new MyThread();

        try {
            // Main thread executing
            for (int i = 5; i > 0; i--) {
                System.out.println("Running main thread: " + i);
                Thread.sleep(1000);  // Sleep for 1 second
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        // Message when the main thread finishes
        System.out.println("Exiting main thread");
    }
}

class MyThread extends Thread {
    // Constructor to initialize the thread and start it
    MyThread() {
        super("Using thread class");
        System.out.println("Child thread: " + this);
        start();  // Start the thread
    }

    // The run method that will execute in the child thread
    public void run() {
        try {
            // Child thread executing
            for (int i = 5; i > 0; i--) {
                System.out.println("Child thread: " + i);
                Thread.sleep(500);  // Sleep for 0.5 seconds
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        // Message when the child thread finishes
        System.out.println("Exiting child thread");
    }
}
```

12. With example explain isalive() & join() calls in thread

1. isAlive() Method:

The isAlive() method in Java is used to **check if a thread is still running or not**. It **returns true if the thread has been started** and has not yet completed execution, and false if the thread has completed execution or has not been started.

Syntax:

```
public boolean isAlive()
```

Example for isAlive():

```
class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + " is running");
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

public class ThreadAliveExample {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();

        // Checking if the thread is alive
        System.out.println("Is thread alive? " + t1.isAlive()); // true after starting

        t1.join();  // Wait for the thread to finish
        System.out.println("Is thread alive? " + t1.isAlive()); // false after the thread
finishes
    }
}
```

Op:-

```
Thread-0 is running
Thread-0 is running
Thread-0 is running
Thread-0 is running
Thread-0 is running
Is thread alive? true
Is thread alive? false
```

2. join() Method:

The join() method is **used to pause the execution of the current thread** until the thread on which join() was called has finished executing. It's helpful when you need the main thread or any other thread to wait for another thread to complete before continuing its execution.

Syntax:

public final void join() throws InterruptedException

Example

```
class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + " is
running");
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

public class ThreadJoinExample {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();

        // Wait for t1 to finish before continuing
        t1.join();

        // Code here will only run after t1 has finished
        System.out.println("Main thread finished after t1 completes.");
    }
}
```

Op:-

```
Thread-0 is running
Thread-0 is running
Thread-0 is running
```

Thread-0 is running
Thread-0 is running
Main thread finished after t1 completes.

Summary

- **isAlive():** Checks if a thread is still running (true if running, false if finished).
- **join():** Makes the current thread wait for the specified thread to finish execution before proceeding.