OOPs in JAVA
3rd SEM Exam – Important Questions

MODULE – 5

1. Define Enumerations. Give an example.

Ans:

Enumerations:

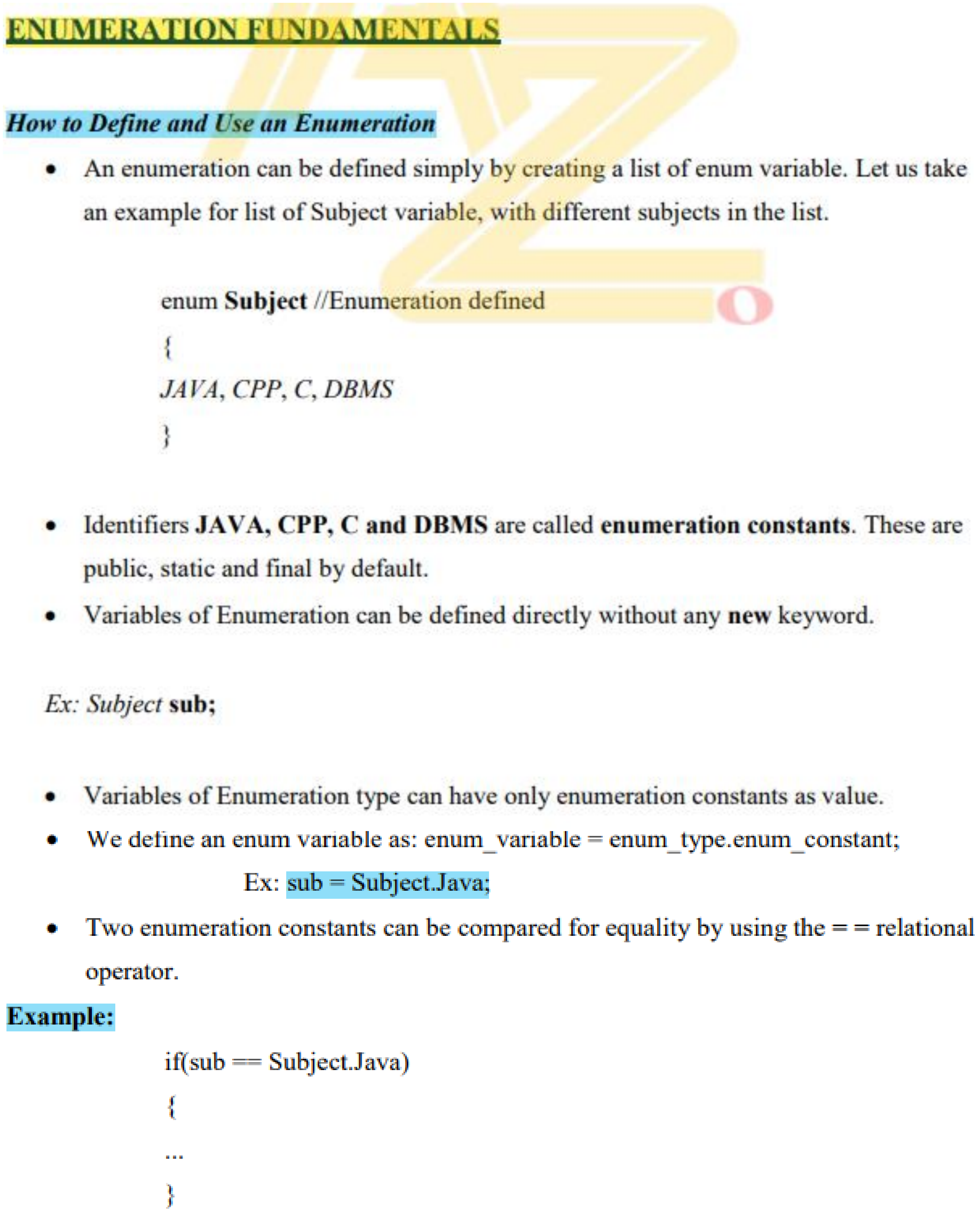
1. **Definition:** Enumerations (enums) in Java are special data types used to define a collection of constants. They provide a way to represent a fixed set of predefined values.
2. **Purpose:** Enumerations improve code readability, maintainability, and type safety by defining a set of named constants that can be used throughout the codebase.
3. **Declaration:** Enums are declared using the `enum` keyword followed by the name of the enumeration and a list of constant values enclosed in braces `{}`.
4. **Example:** Below is an example of an enumeration named `DayOfWeek`, representing the days of the week.

```java
// Definition of an enumeration representing days of the week
enum DayOfWeek {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}

public class Main {
    public static void main(String[] args) {
        // Accessing enum constants
        DayOfWeek day1 = DayOfWeek.MONDAY;
        DayOfWeek day2 = DayOfWeek.WEDNESDAY;

        // Using enum constants
        System.out.println("Day 1: " + day1);
        System.out.println("Day 2: " + day2);
    }
}
```

In this example:

- We define an enumeration named `DayOfWeek` containing seven constants representing each day of the week.
- In the `main` method, we create variables `day1` and `day2` of type `DayOfWeek` and assign them constants `MONDAY` and `WEDNESDAY`, respectively.
- We then print the values of `day1` and `day2`, which will output "Day 1: MONDAY" and "Day 2: WEDNESDAY" to the console.

(or)



ENUMERATION FUNDAMENTALS

How to Define and Use an Enumeration

- An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject //Enumeration defined
{
JAVA, CPP, C, DBMS
}
```

- Identifiers **JAVA, CPP, C and DBMS** are called **enumeration constants**. These are public, static and final by default.
- Variables of Enumeration can be defined directly without any **new** keyword.

Ex: Subject **sub;**

- Variables of Enumeration type can have only enumeration constants as value.
- We define an enum variable as: enum_variable = enum_type.enum_constant;

  Ex: sub = Subject.Java;
- Two enumeration constants can be compared for equality by using the = = relational operator.

Example:

```
if(sub == Subject.Java)
{
...
}
```

Program 1: Example of Enumeration

```
enum WeekDays
{
        sun, mon, tues, wed, thurs, fri, sat
}
class Test
{
        public static void main(String args[])
        {
                WeekDays wk; //wk is an enumeration variable of type WeekDays
                wk = WeekDays.sun;
                //wk can be assigned only the constants defined under enum type Weekdays
                System.out.println("Today is "+wk);
        }
}
```

Output : Today is sun

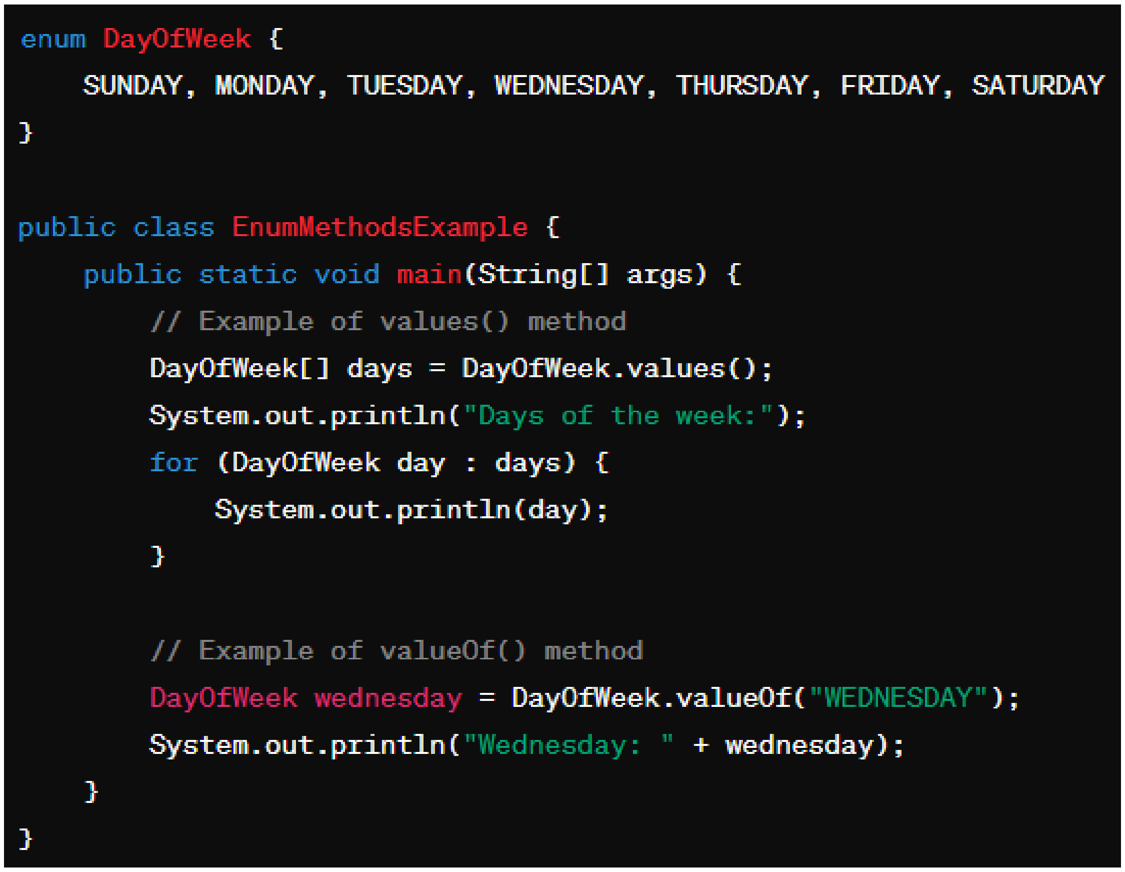
2. Discuss values() and value Of() methods in Enumerations with suitable examples.

Ans:

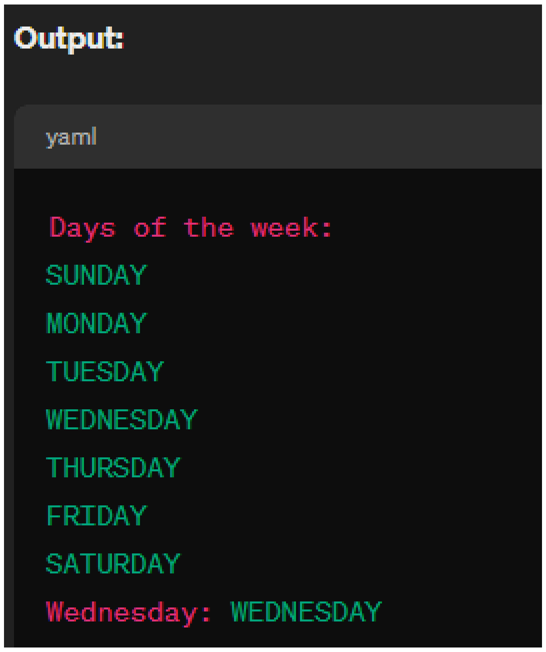
values() and valueOf() Methods in Enumerations:

1. **values() Method:**
    - The `values()` method returns an array containing all the constants defined within an enumeration.
    - It allows iterating over all the constants in the enumeration or accessing them by index.
    - This method is useful for tasks such as iterating through all enum constants or performing operations on each constant.
2. **valueOf() Method:**
    - The `valueOf()` method is used to obtain the enum constant corresponding to the specified string representation.
    - It parses the string and returns the enum constant with the matching name.
    - This method is helpful when converting user input or configuration settings represented as strings into enum constants.

Example:



```java
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumMethodsExample {
    public static void main(String[] args) {
        // Example of values() method
        DayOfWeek[] days = DayOfWeek.values();
        System.out.println("Days of the week:");
        for (DayOfWeek day : days) {
            System.out.println(day);
        }

        // Example of valueOf() method
        DayOfWeek wednesday = DayOfWeek.valueOf("WEDNESDAY");
        System.out.println("Wednesday: " + wednesday);
    }
}
```



Output:

yaml

```
Days of the week:
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
Wednesday: WEDNESDAY
```

In this example:

- We use the `values()` method to obtain an array of all constants in the `DayOfWeek` enumeration and print them.
- We then use the `valueOf()` method to obtain the enum constant representing Wednesday ("WEDNESDAY") and print it.

values() and valueOf() Methods

- The java compiler internally adds the values() method when it creates an enum.
- The values() method returns an array containing all the values of the enum.
- Its general form is,

  public **static** *enum-type[ ]* **values()**
- valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method.
- It's general form is,

  public **static** *enum-type* **valueOf** (String *str*)

3. "Enumerations in Java are class types"-justify this statement with appropriate examples.

Ans:

The statement "Enumerations in Java are class types" implies that enum types are treated as classes in Java, offering features and capabilities similar to traditional classes. Here's how we can justify this statement with appropriate examples:

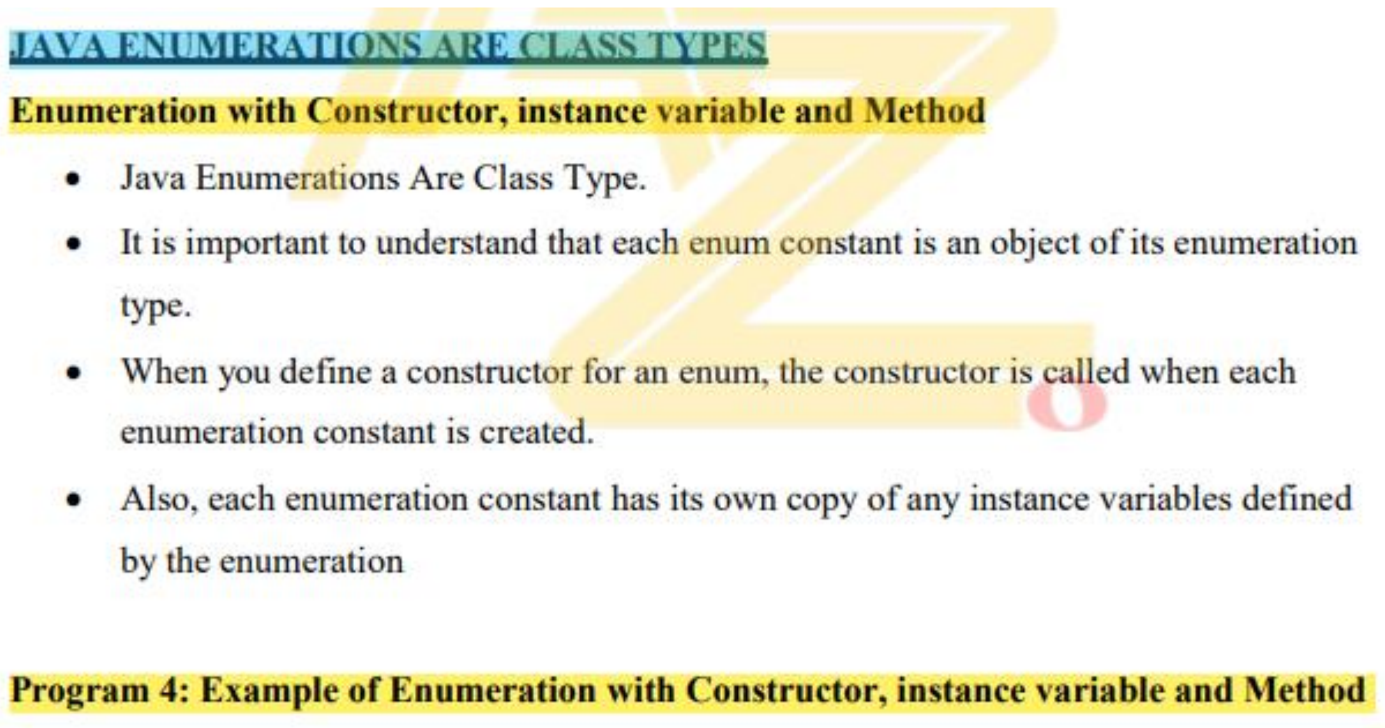
1. **Defining Enumerations as Class Types:**
   - Enumerations in Java are defined using the `enum` keyword, which signifies the creation of a new data type, similar to class declaration syntax.
   - Enumerations can have constructors, methods, and fields, similar to classes. This allows enum types to encapsulate behavior and state within themselves.
2. **Example Demonstrating Enum as a Class Type:**

```java
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    private String abbreviation;

    // Constructor
    DayOfWeek() {
        this.abbreviation = name().substring(0, 3); // Abbreviation of day
    }

    // Method
    public String getAbbreviation() {
        return abbreviation;
    }
}
```

3. **Justification:**
    - In the above example, `DayOfWeek` is an enum type, defined using the `enum` keyword.
    - It contains a constructor (`DayOfWeek()`) and a method (`getAbbreviation()`), demonstrating behavior encapsulation similar to class types.
    - Enum constants (`SUNDAY`, `MONDAY`, etc.) act as instances of the `DayOfWeek` enum type, analogous to objects of a class.
    - The enum constants can access methods and fields defined within the enum type, just like objects accessing methods and fields of a class instance.
4. **Usage Similarity with Class Types:**
    - Enumerations are used to represent a fixed set of predefined constants, providing type safety and readability.
    - They can be used in switch statements, method parameters, and return types, just like class types.
    - Enumerations support inheritance, interfaces, and annotations, further emphasizing their class-like behavior and capabilities.

In summary, the statement "Enumerations in Java are class types" is justified by the fact that enum types are defined similarly to classes, can encapsulate behavior and state, and offer features analogous to class types in Java.

(or)

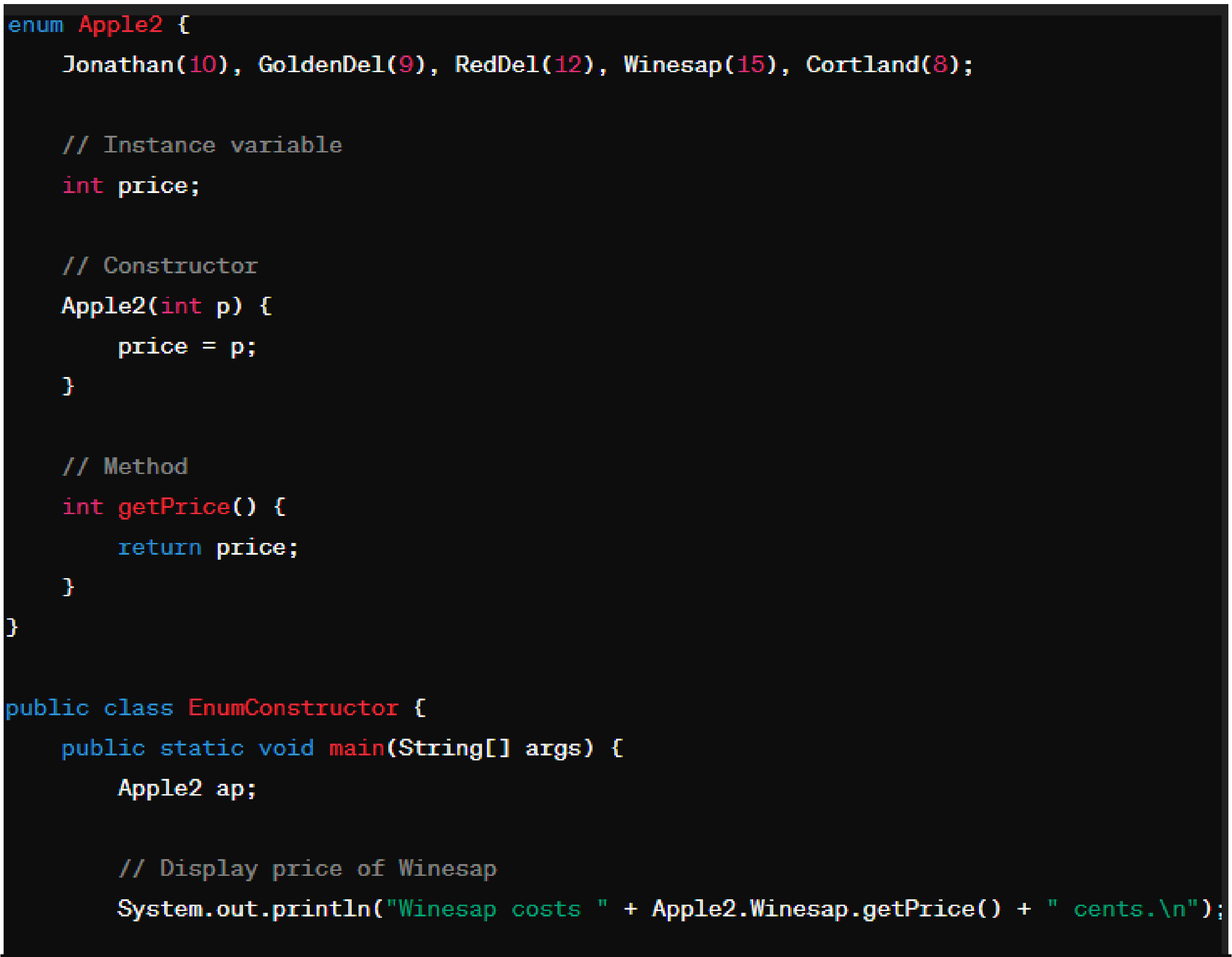


JAVA ENUMERATIONS ARE CLASS TYPES

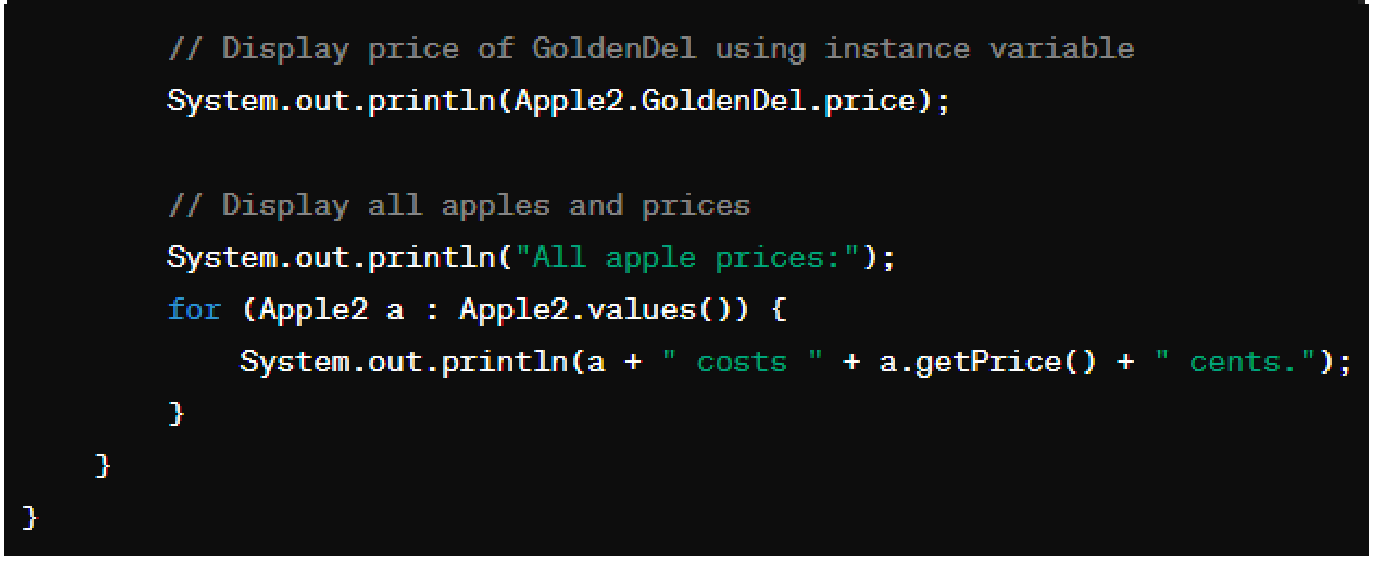
Enumeration with Constructor, instance variable and Method

- Java Enumerations Are Class Type.
- It is important to understand that each enum constant is an object of its enumeration type.
- When you define a constructor for an enum, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration

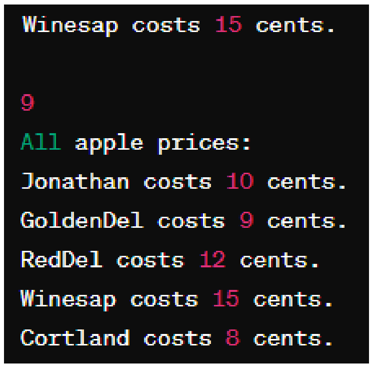
Program 4: Example of Enumeration with Constructor, instance variable and Method



```java
enum Apple2 {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    // Instance variable
    int price;

    // Constructor
    Apple2(int p) {
        price = p;
    }

    // Method
    int getPrice() {
        return price;
    }
}

public class EnumConstructor {
    public static void main(String[] args) {
        Apple2 ap;

        // Display price of Winesap
        System.out.println("Winesap costs " + Apple2.Winesap.getPrice() + " cents.\n");
```



```java
        // Display price of GoldenDel using instance variable
        System.out.println(Apple2.GoldenDel.price);

        // Display all apples and prices
        System.out.println("All apple prices:");
        for (Apple2 a : Apple2.values()) {
            System.out.println(a + " costs " + a.getPrice() + " cents.");
        }
    }
}
```



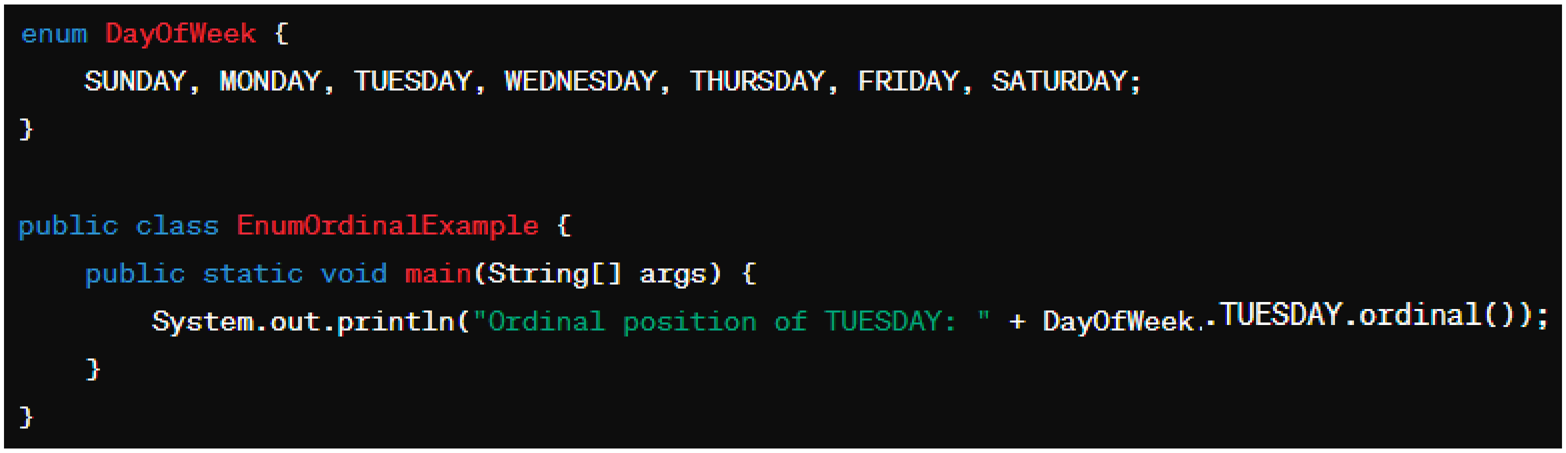
```
Winesap costs 15 cents.

9
All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

4. Write a note on ordinal() and compare To() methods.

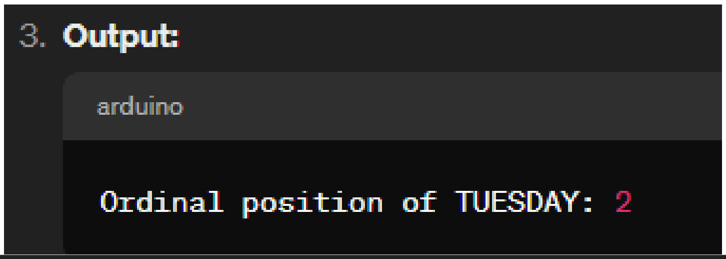
Ans:

ordinal() Method:

1. **Explanation:**
    - The `ordinal()` method returns the ordinal position of an enum constant within its enumeration declaration.
    - Enum constants are assigned ordinal values starting from zero for the first constant, incrementing by one for each subsequent constant.
    - This method provides a convenient way to obtain the numerical order of enum constants.



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

public class EnumOrdinalExample {
    public static void main(String[] args) {
        System.out.println("Ordinal position of TUESDAY: " + DayOfWeek.TUESDAY.ordinal());
    }
}
```



3. **Output:**

arduino

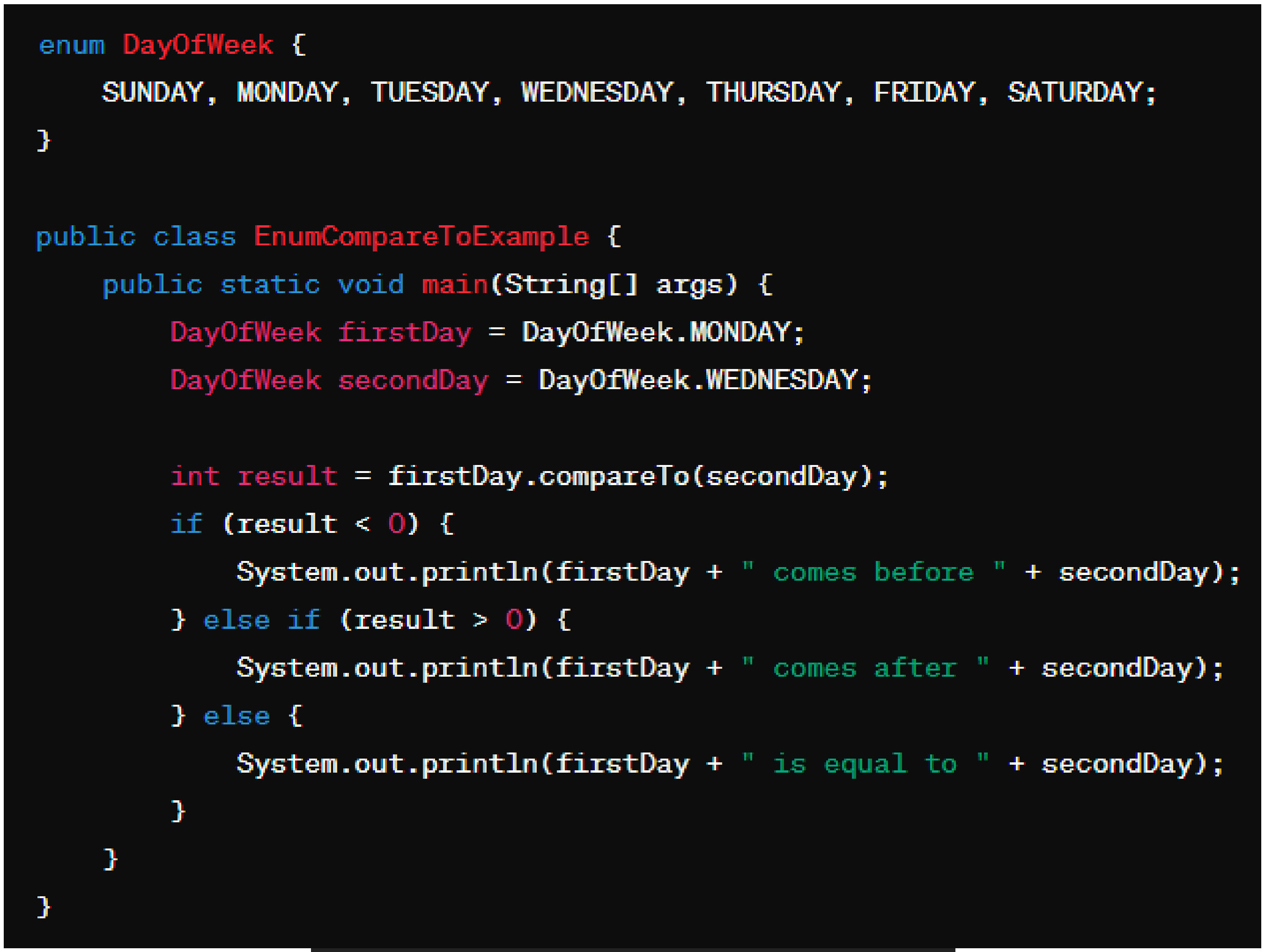
```
Ordinal position of TUESDAY: 2
```

In this example, the `ordinal()` method is used to retrieve the ordinal position of the `TUESDAY` constant, which is 2.

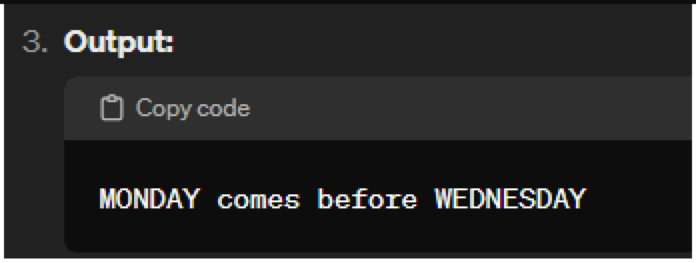
compareTo() Method:

1. **Explanation:**
    - The `compareTo()` method compares two enum constants based on their ordinal values.
    - It returns a negative integer if the invoking constant is less than the argument constant, zero if they are equal, and a positive integer if the invoking constant is greater.
    - This method enables sorting enum constants in their natural order based on their ordinal values.
2. **Example:**

In this example, the `compareTo()` method is used to compare `MONDAY` with `WEDNESDAY`, indicating that `MONDAY` comes before `WEDNESDAY`. In this example, the `compareTo()` method is used to compare `MONDAY` with `WEDNESDAY`, indicating that `MONDAY` comes before `WEDNESDAY`.



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

public class EnumCompareToExample {
    public static void main(String[] args) {
        DayOfWeek firstDay = DayOfWeek.MONDAY;
        DayOfWeek secondDay = DayOfWeek.WEDNESDAY;

        int result = firstDay.compareTo(secondDay);
        if (result < 0) {
            System.out.println(firstDay + " comes before " + secondDay);
        } else if (result > 0) {
            System.out.println(firstDay + " comes after " + secondDay);
        } else {
            System.out.println(firstDay + " is equal to " + secondDay);
        }
    }
}
```



3. **Output:**

```
MONDAY comes before WEDNESDAY
```

5. What are wrapper classes? Explain with examples.

Ans:

Wrapper Classes in Java:

1. **Explanation:**
    - Wrapper classes in Java are used to convert primitive data types into objects (i.e., to wrap primitives into objects).
    - They provide a way to represent primitive data types as objects, allowing them to be used in Java collections, generics, and other scenarios where objects are required.
    - Each primitive data type has a corresponding wrapper class in Java.
2. **Example:**

```java
// Example of wrapper classes
public class WrapperExample {
    public static void main(String[] args) {
        // Primitive data types
        int num = 10;
        double value = 3.14;

        // Wrapping primitive data types into objects
        Integer integerObj = Integer.valueOf(num); // Integer wrapper class
        Double doubleObj = Double.valueOf(value); // Double wrapper class

        // Printing wrapped objects
        System.out.println("Integer object: " + integerObj);
        System.out.println("Double object: " + doubleObj);
    }
}
```

3. Output

```
Integer object: 10
Double object: 3.14
```

4. Explanation (Continued):

- In the example, we have primitive data types `int` and `double`.
- We use the wrapper classes `Integer` and `Double` to wrap these primitive data types into objects.
- The `valueOf()` method is used to convert primitive data types to their corresponding wrapper objects.
- The resulting wrapper objects (`Integer` and `Double`) can then be used like any other objects in Java.

5. Wrapper Classes in Java:

- `Byte`, `Short`, `Integer`, `Long`: For representing integral values.
- `Float`, `Double`: For representing floating-point values.
- `Character`: For representing characters.
- `Boolean`: For representing boolean values.

6. Usage:

- Wrapper classes are commonly used in scenarios where primitive data types need to be treated as objects, such as when working with collections like `ArrayList`, `HashMap`, or when using generics.

Wrapper classes provide a convenient way to work with primitive data types in Java, allowing them to be treated as objects and enabling additional functionalities such as methods and compatibility with object-oriented features of Java.

Type Wrappers

- Java provides type wrapper classes that encapsulate primitive types within objects.
- Type wrapper classes include `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character`, and `Boolean`.
- These classes offer methods that allow integration of primitive types into Java's object hierarchy.
- Java's autoboxing feature automatically converts primitive types to their corresponding wrapper classes when necessary, and vice versa.
- This simplifies the process of working with both primitive types and objects, as conversions are handled implicitly by the compiler.

Overall, type wrappers in Java allow primitive types to be used in situations where objects are required, providing a bridge between the world of primitive types and the object-oriented nature of Java. Autoboxing further simplifies the interaction between primitive types and objects by handling conversions automatically.

6. Explain auto boxing /unboxing in expressions.

Ans:

Auto Boxing and Unboxing in Java:

1. **Auto Boxing:**
   - Auto boxing is the automatic conversion of primitive data types into their corresponding wrapper classes when necessary.
   - It allows primitive data types to be used as objects without explicitly creating instances of wrapper classes.
2. **Example of Auto Boxing:**

```
// Auto boxing example
Integer numObj = 10; // Auto boxing: int to Integer
Double valueObj = 3.14; // Auto boxing: double to Double
```

3. **Explanation (Auto Boxing):**
   - In the example, primitive data types (`int` and `double`) are assigned directly to objects of their corresponding wrapper classes (`Integer` and `Double`).
   - Java automatically converts the primitive values into objects of the wrapper classes using auto boxing.
4. **Unboxing:**
   - Unboxing is the automatic conversion of wrapper class objects into their corresponding primitive data types when necessary.
   - It allows objects of wrapper classes to be used as primitive data types without explicitly extracting their values.
5. **Example of Unboxing:**

```java
// Unboxing example
int num = numObj; // Unboxing: Integer to int
double value = valueObj; // Unboxing: Double to double
```

6. **Explanation (Unboxing):**
   - In the example, objects of wrapper classes (`Integer` and `Double`) are assigned directly to primitive data types (`int` and `double`).
   - Java automatically extracts the values from the wrapper class objects and assigns them to the corresponding primitive data types using unboxing.
7. **Usage:**
   - Auto boxing and unboxing provide syntactic sugar, making code more readable and concise by eliminating the need for manual conversions between primitive data types and wrapper classes.
   - They are commonly used when working with collections, generics, and other scenarios where primitive data types and objects need to interact seamlessly.

Auto boxing and unboxing simplify the process of working with primitive data types and wrapper classes in Java, enhancing code readability and reducing boilerplate code.

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

Ans:

1. **Explanation:**
   - Multithreading is the ability of a CPU or a single core in a computer to provide multiple threads of execution concurrently.
   - In Java, multithreading allows a program to perform multiple tasks simultaneously by dividing its execution into multiple threads.
   - Each thread represents a separate flow of execution within the same program.
2. **Example Program - Creating Multiple Threads:**

```java
// Example program to create multiple threads in Java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread " + Thread.currentThread().getId() + " is running
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        int numThreads = 5; // Number of threads to create

        // Creating and starting multiple threads
        for (int i = 0; i < numThreads; i++) {
            MyThread thread = new MyThread();
            thread.start(); // Starting the thread
        }
    }
}
```

3. Output (Sample):

```
Thread 11 is running.
Thread 12 is running.
Thread 13 is running.
Thread 14 is running.
Thread 15 is running.
```

4.Explanation (Program):

- In the program, we define a class `MyThread` that extends the `Thread` class and overrides the `run()` method.
- Inside the `run()` method, each thread prints a message indicating its ID using `Thread.currentThread().getId()`.
- In the `main()` method, we create and start multiple instances of `MyThread` to create multiple threads.
- Each thread executes concurrently, and their IDs are printed sequentially.

5.Usage:

- Multithreading is used in scenarios where tasks can be performed concurrently, such as in server applications, GUI programming, simulations, etc.
- It helps improve program performance by utilizing available CPU resources more efficiently and increasing responsiveness in applications.

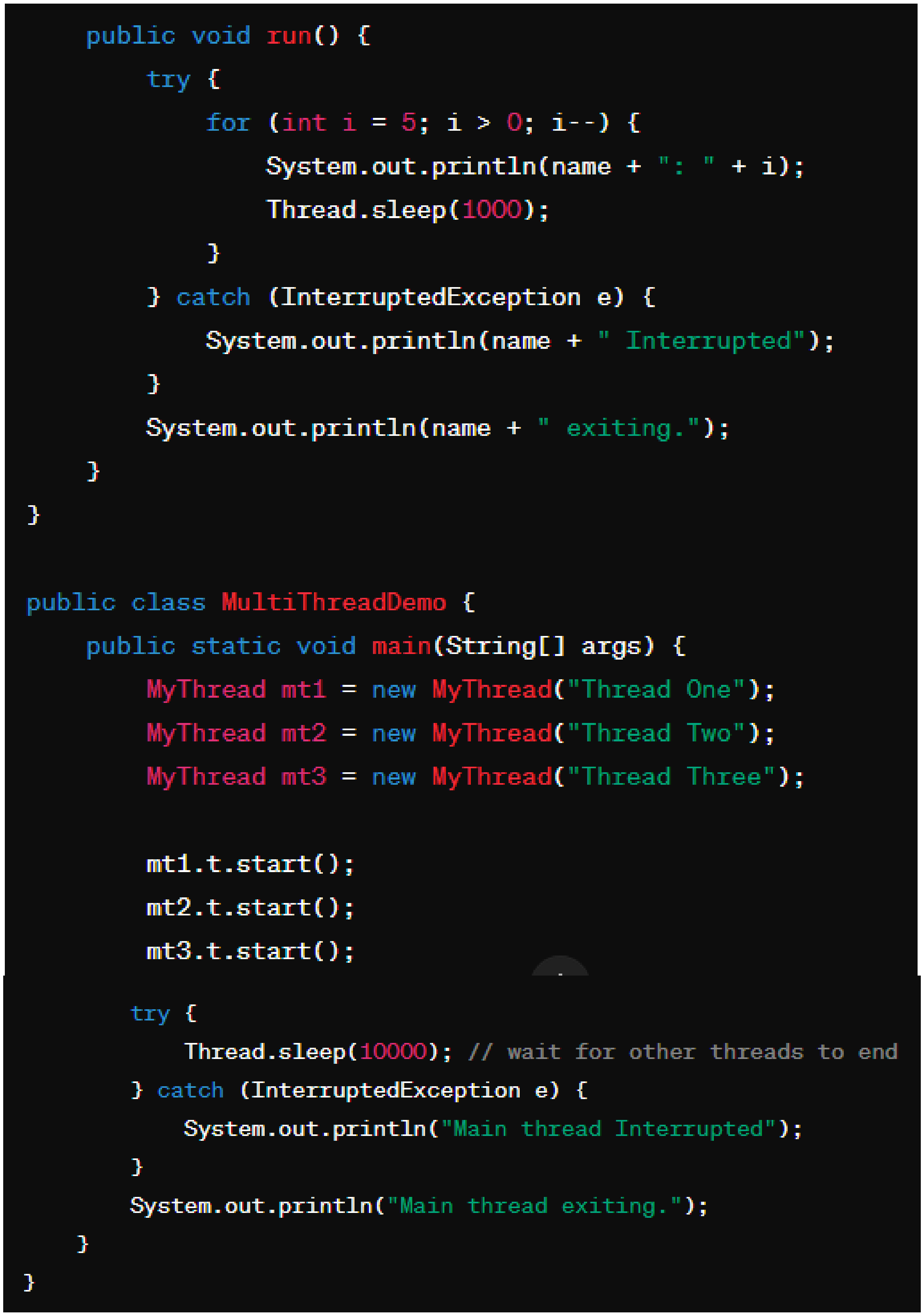
Multithreading in Java allows programs to perform multiple tasks simultaneously, enhancing performance and efficiency in various applications.

(or)

Multithreaded Programming

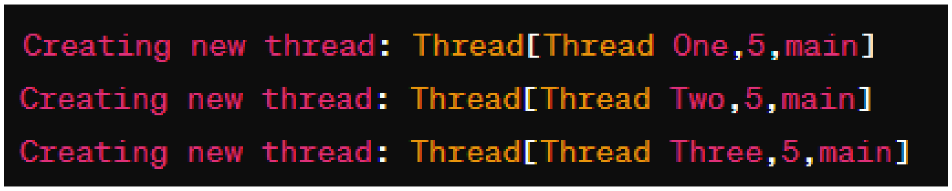
- Multithreading in Java allows for concurrent execution of multiple parts of a program, known as threads.
- This contrasts with process-based multitasking, where each program is a separate unit, whereas threads share the same address space and belong to the same process.
- Multithreading is advantageous due to its lower overhead compared to process-based multitasking.
- It helps in maximizing processing power by minimizing idle time, especially in interactive and networked environments where tasks like data transmission and user input are slower compared to CPU processing speed.
- Multithreading enables more efficient use of available resources and smoother program execution by allowing other threads to run while one is waiting.

```java
// Simplified version of creating multiple threads
class MyThread implements Runnable {
    String name;
    Thread t;

    MyThread(String threadName) {
        name = threadName;
        t = new Thread(this, name);
        System.out.println("Creating new thread: " + t);
    }
```

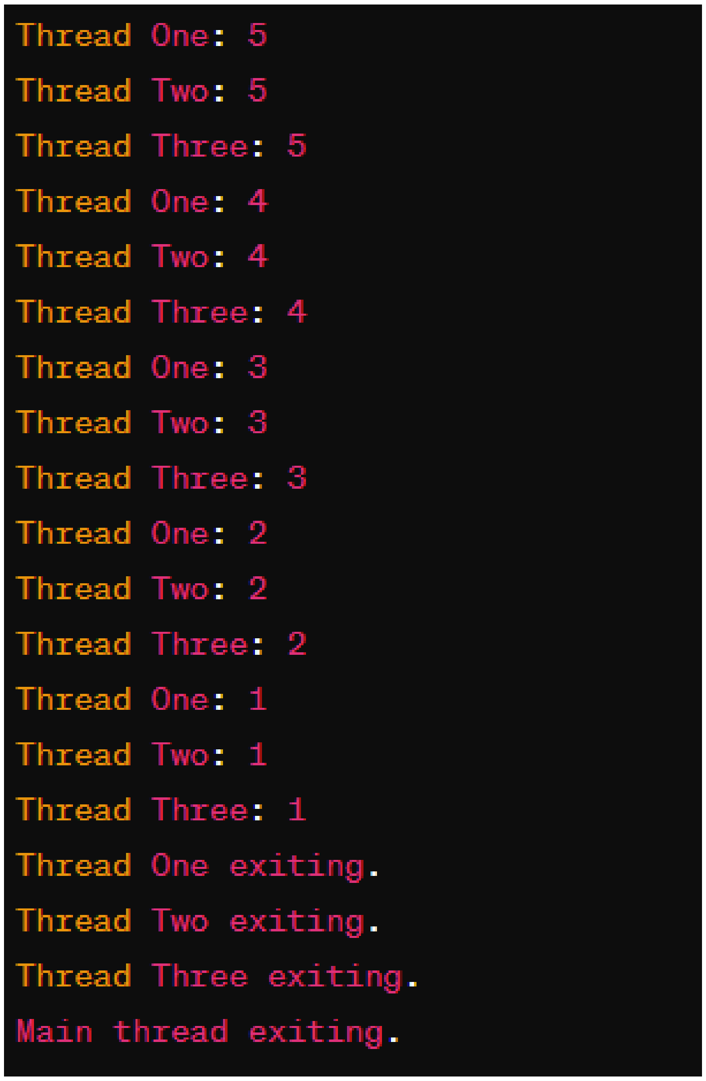


```java
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        MyThread mt1 = new MyThread("Thread One");
        MyThread mt2 = new MyThread("Thread Two");
        MyThread mt3 = new MyThread("Thread Three");

        mt1.t.start();
        mt2.t.start();
        mt3.t.start();

        try {
            Thread.sleep(10000); // wait for other threads to end
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Output:



```
Creating new thread: Thread[Thread One,5,main]
Creating new thread: Thread[Thread Two,5,main]
Creating new thread: Thread[Thread Three,5,main]
```



```
Thread One: 5
Thread Two: 5
Thread Three: 5
Thread One: 4
Thread Two: 4
Thread Three: 4
Thread One: 3
Thread Two: 3
Thread Three: 3
Thread One: 2
Thread Two: 2
Thread Three: 2
Thread One: 1
Thread Two: 1
Thread Three: 1
Thread One exiting.
Thread Two exiting.
Thread Three exiting.
Main thread exiting.
```

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

Ans:

Thread in Java:

1. **Definition:**
   - A thread in Java represents a separate path of execution within a program.
   - Threads allow for concurrent execution, enabling programs to perform multiple tasks simultaneously.
2. **Creating Threads:**
   - Java provides two main approaches for creating threads:
     - Implementing the `Runnable` interface.
     - Extending the `Thread` class.

Implementing the Runnable Interface:

1. **Explanation:**
   - Define a class that implements the `Runnable` interface.
   - Implement the `run()` method to define the thread's behavior.
2. **Example:**

```
class MyRunnable implements Runnable {
    public void run() {
        // Define thread behavior
    }
}

// Create an instance of the class implementing Runnable
MyRunnable myRunnable = new MyRunnable();
// Create a Thread object passing the instance of Runnable
Thread thread = new Thread(myRunnable);
// Start the thread
thread.start();
```

Extending the Thread Class:

1. **Explanation:**
   - Create a subclass that extends the `Thread` class.
   - Override the `run()` method to specify the thread's behavior.
2. **Example:**

```
class MyThread extends Thread {
    public void run() {
        // Define thread behavior
    }
}

// Create an instance of the Thread subclass
MyThread myThread = new MyThread();
// Start the thread
myThread.start();
```

(or)

Implementing Runnable:

1. **Definition:** Runnable interface abstracts a unit of executable code.
2. **run() Method:** Implementing classes need to provide a `run()` method, which serves as the entry point for the new thread.
3. **Thread Instantiation:** After implementing Runnable, instantiate a Thread object using the constructor `Thread(Runnable threadOb, String threadName)`.
4. **Starting the Thread:** Call the `start()` method on the Thread object to begin execution of the new thread.

5. **Execution Context:** Code within `run()` operates like any other method, allowing for method calls, variable declarations, etc.

Extending Thread:

1. **Definition:** Another way to create a thread is by extending the Thread class.
2. **Override run():** The extending class must override the `run()` method to define the thread's behavior.
3. **Starting the Thread:** Invoke `start()` method to commence execution of the thread.
4. **run() Execution:** Code within the overridden `run()` method specifies what the thread should execute.
5. **Simplicity:** Extending Thread offers a straightforward approach, but limits the flexibility to extend other classes.

These simplified points encapsulate the essence of creating threads in Java using both approaches. They provide a concise understanding suitable for exam preparations.

9. Write a JAVA program to create two threads, one displays "computer science" and another displays "information science" five times.

Ans:

```java
class MessageThread extends Thread {
    private String message;

    public MessageThread(String message) {
        this.message = message;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(message);
            try {
                Thread.sleep(1000); // Pause for 1 second between each message
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class ThreadExample {
    public static void main(String[] args) {
        // Create two threads with different messages
        MessageThread thread1 = new MessageThread("Computer Science");
        MessageThread thread2 = new MessageThread("Information Science");

        // Start both threads
        thread1.start();
        thread2.start();
    }
}
```

In this program:

- We define a `MessageThread` class that extends `Thread`.
- Each `MessageThread` object is initialized with a specific message.
- The `run()` method of `MessageThread` is overridden to display the message five times, with a 1-second pause between each display.
- In the `main()` method, we create two `MessageThread` objects with different messages and start both threads using the `start()` method.

10. With syntax, explain use Of is Alive( ) and join( ) methods.

Ans:

1. **isAlive() Method:**
    - `isAlive()` is a method in Java used to check whether a thread is alive or not.
    - A thread is considered alive if it has been started and has not yet completed execution or terminated.
    - It returns a boolean value, `true` if the thread is alive, `false` otherwise.

Syntax of `isAlive()` Method:

```java
boolean isAlive()
```

Example of `isAlive()` Method:

```java
Thread thread = new Thread(() -> {
    // Thread execution logic
});
thread.start(); // Start the thread
boolean alive = thread.isAlive(); // Check if the thread is alive
```

In this example, `isAlive()` is used to check if a thread is alive after starting it.

Explanation of `join()` Method:

2. **join() Method:**
    - `join()` is a method in Java that allows one thread to wait for the completion of another thread.
    - When a thread calls `join()` on another thread, it waits until the specified thread terminates.
    - It is useful for coordinating the execution of multiple threads.

Syntax of `join()` Method:

```
void join() throws InterruptedException
```

Example of `join()` Method:

```
Thread thread = new Thread(() -> {
    // Thread execution logic
});
thread.start(); // Start the thread
try {
    thread.join(); // Wait for the thread to finish execution
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

In this example, `join()` is used to wait for the completion of a thread's execution after starting it.

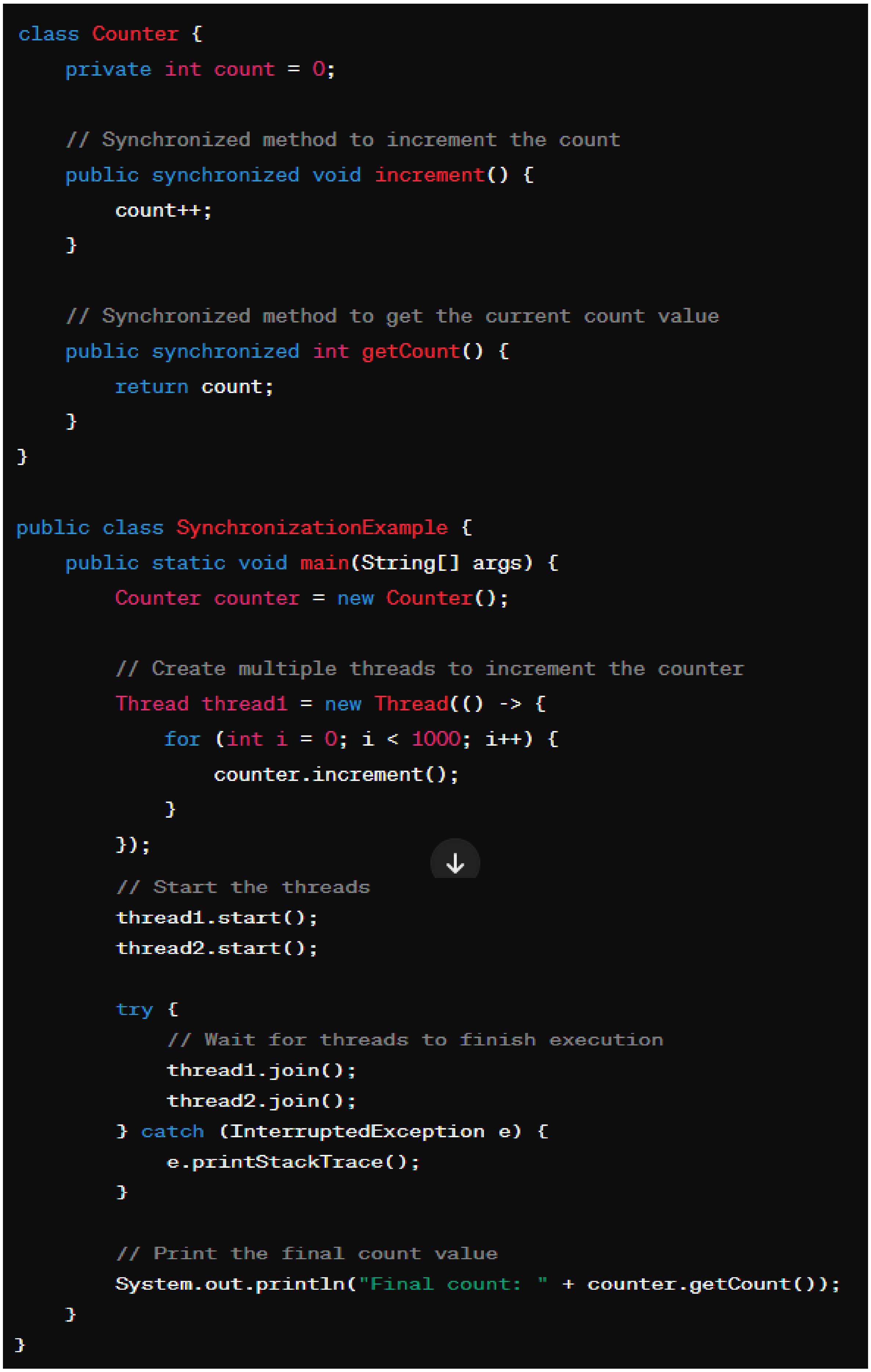
11. What is the need of synchronization? Explain with an example how synchronization is implemented in JAVA.

Ans:

Need for Synchronization:

1. **Ensuring Thread Safety:**
    - In a multithreaded environment, multiple threads may access shared resources concurrently, leading to data inconsistency and errors.
    - Synchronization is needed to ensure that only one thread can access the shared resource at any given time, thus preventing race conditions and maintaining data integrity.
2. **Preventing Concurrent Access:**
    - Synchronization prevents multiple threads from accessing critical sections of code simultaneously.
    - It ensures that the state of shared objects remains consistent and prevents conflicts arising from concurrent access.

Example of Synchronization in Java:



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

    // Synchronized method to increment the count
    public synchronized void increment() {
        count++;
    }

    // Synchronized method to get the current count value
    public synchronized int getCount() {
        return count;
    }
}

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

        // Create multiple threads to increment the counter
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        // Start the threads
        thread1.start();
        thread2.start();

        try {
            // Wait for threads to finish execution
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

In this example:

- We have a `Counter` class with synchronized methods to increment the count and retrieve the current count value.
- Multiple threads are created to concurrently increment the counter.
- Synchronization ensures that only one thread can access the `increment()` and `getCount()` methods at a time, preventing race conditions.
- As a result, the final count printed is the correct sum of increments performed by both threads.

(or)

Synchronizing Threads

o Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

o Java Synchronization is better option where we want to allow only one thread to access the shared resource

Why use Synchronization

The synchronization is mainly used

o To prevent thread interference.

o To prevent consistency problem.

Thread Synchronization

- Synchronized method.
- Synchronized block.

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

General Syntax :

```
synchronized(object)
{
//statement to be synchronized
}
```

Synchronization

➢ When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.

Need for Synchronization:

- Synchronization is necessary to ensure thread safety, especially when multiple threads access shared resources concurrently.
- Without synchronization, concurrent access to shared resources can lead to data corruption, race conditions, and other inconsistencies.

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

Ans:

Inter Thread Communication in Java:

Inter thread communication enables threads to synchronize and coordinate their execution in Java. It's facilitated through methods like `wait()`, `notify()`, and `notifyAll()` from the `Object` class.

- **wait():** Pauses the current thread until another thread invokes `notify()` or `notifyAll()` for the same object. Releases the object's lock during the wait.
- **notify():** Wakes up a single waiting thread on the object. If multiple threads are waiting, one is chosen arbitrarily.
- **notifyAll():** Wakes up all threads waiting on the object. Allows them to compete for the lock.

These methods are used with synchronized blocks or methods to ensure proper synchronization between threads, facilitating communication and coordination.

Example of Inter Thread Communication in Java:

```java
class Customer {
    int amount = 10000;

    synchronized void withdraw(int amount) {
        System.out.println("Going to withdraw...");
        if (this.amount < amount) {
            System.out.println("Less balance; waiting for deposit...");
            try {
                wait(); // Wait if balance is insufficient
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.amount -= amount;
        System.out.println("Withdraw completed...");
    }
    synchronized void deposit(int amount) {
        System.out.println("Going to deposit...");
        this.amount += amount;
        System.out.println("Deposit completed...");
        notify(); // Notify waiting thread that deposit is done
    }
}

class Test {
    public static void main(String args[]) {
        final Customer c = new Customer();

        // Thread to withdraw amount
        new Thread() {
            public void run() {
                c.withdraw(15000);
            }
        }.start();

        // Thread to deposit amount
        new Thread() {
            public void run() {
                c.deposit(10000);
            }
        }.start();
    }
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...
```

Explanation:

- In this example, we have a `Customer` class with `withdraw()` and `deposit()` methods, which are synchronized to ensure thread safety.
- The `withdraw()` method checks if the balance is sufficient for withdrawal. If not, it waits for the `deposit()` method to deposit funds.
- The `deposit()` method adds funds to the balance and notifies the waiting thread that the deposit is completed.
- In the `Test` class, two threads are created: one for withdrawal and one for deposit.
- The withdrawal thread attempts to withdraw an amount greater than the balance, so it waits for the deposit thread to add funds.
- After the deposit is completed, the withdrawal thread resumes and completes the withdrawal process.

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

Ans:

Thread Priority in Java:

Thread priority determines the preference given to a thread for CPU time when multiple threads are competing. Thread priorities are represented by integers ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest. By default, threads inherit the priority of their parent thread.

- **Assigning Priority:** You can set the priority of a thread using the `setPriority(int priority)` method, where `priority` is an integer value between 1 and 10. Higher values indicate higher priority.
- **Getting Priority:** The priority of a thread can be obtained using the `getPriority()` method.

It's important to note that thread priority is merely a hint to the scheduler and does not guarantee the execution order. The final decision lies with the JVM and underlying operating system scheduler.

```
// Assigning priority to a thread
Thread thread = new Thread();
thread.setPriority(Thread.MAX_PRIORITY); // Sets the priority to the highest

// Getting priority of a thread
int priority = thread.getPriority();
```

(or)

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, a thread's priority is used to decide when to switch from one running thread to the next.
- This is called a *context switch.*
- The rules that determine when a context switch takes place are simple:
- *1) A thread can voluntarily relinquish control :* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *2) A thread can be preempted by a higher-priority thread :* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. This is called *preemptive multitasking*

- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.
- For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion.
- For other types of operating systems, threads of equal priority must voluntarily yield control to their peers.

(or)

Thread Priorities

Thread Priorities:

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads over a given period of time.
- Higher-priority threads can preempt lower-priority ones, meaning they can interrupt the execution of lower-priority threads.

Equal Priority Threads:

- In theory, threads of equal priority should get equal access to the CPU.
- However, Java is designed to work in various environments, and the actual behavior may differ depending on the operating system and multitasking implementation.
- To ensure fairness, threads that share the same priority should yield control occasionally, especially in nonpreemptive environments.

Setting Thread Priority:

- Use the **setPriority()** method to set a thread's priority.
- Syntax: **void setPriority(int level)**
- The **level** parameter specifies the new priority setting for the thread, and it must be within the range of **MIN_PRIORITY** and **MAX_PRIORITY**, currently 1 and 10, respectively.
- To return a thread to default priority, use **NORM_PRIORITY**, which is currently 5.

Getting Thread Priority:

- Use the **getPriority()** method to obtain the current priority setting of a thread.
- Syntax: **int getPriority()**

Implementation Considerations:

- Implementations of Java may have different behaviors when it comes to scheduling and thread priorities.
- To ensure predictable and cross-platform behavior, it's advisable to use threads that voluntarily give up CPU time.

14. Explain how to achieve suspending, resuming and stopping threads with an example program.

Ans:

Suspending, Resuming, and Stopping Threads in Java:

1. **Suspending Threads:**
   - Suspending a thread temporarily halts its execution without terminating it.
   - You can suspend a thread using the `suspend()` method.
2. **Resuming Threads:**
   - Resuming a suspended thread allows it to continue its execution.
   - Resuming is achieved using the `resume()` method.
3. **Stopping Threads:**
   - Stopping a thread terminates its execution.
   - It's recommended to use a flag or condition to control the termination of threads rather than abruptly stopping them, as stopping threads abruptly can lead to resource leaks or inconsistent program state.
   - You can implement stopping by setting a flag and checking it periodically in the thread's execution loop.

(or)

Suspending, Resuming, and Stopping Threads

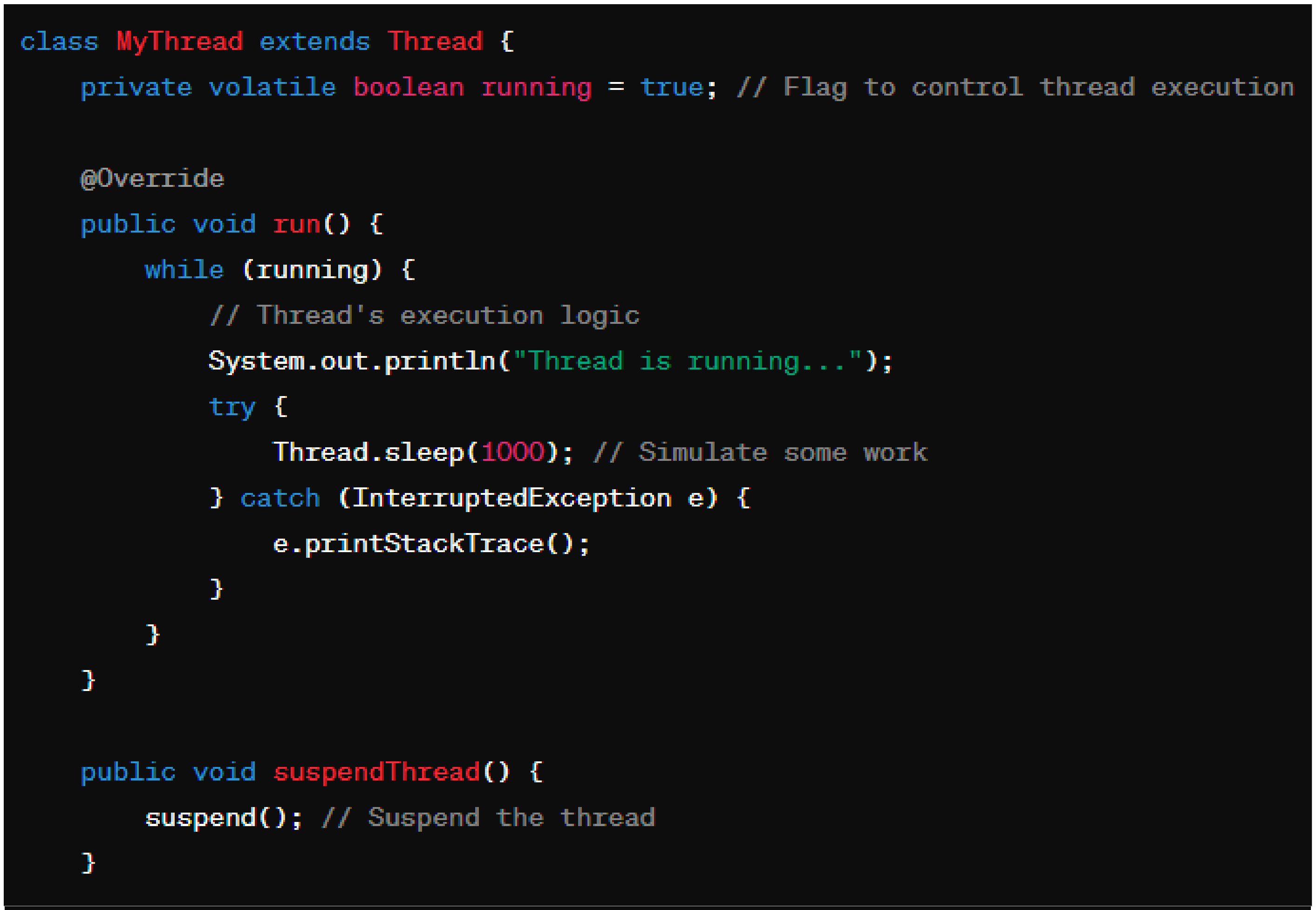
Deprecated Methods:

- In early versions of Java (prior to Java 2), thread suspension, resumption, and termination were managed using **suspend()**, **resume()**, and **stop()** methods defined by the **Thread** class.
- However, these methods were deprecated due to potential issues and risks they posed, such as causing system failures and leaving critical data structures in corrupted states.

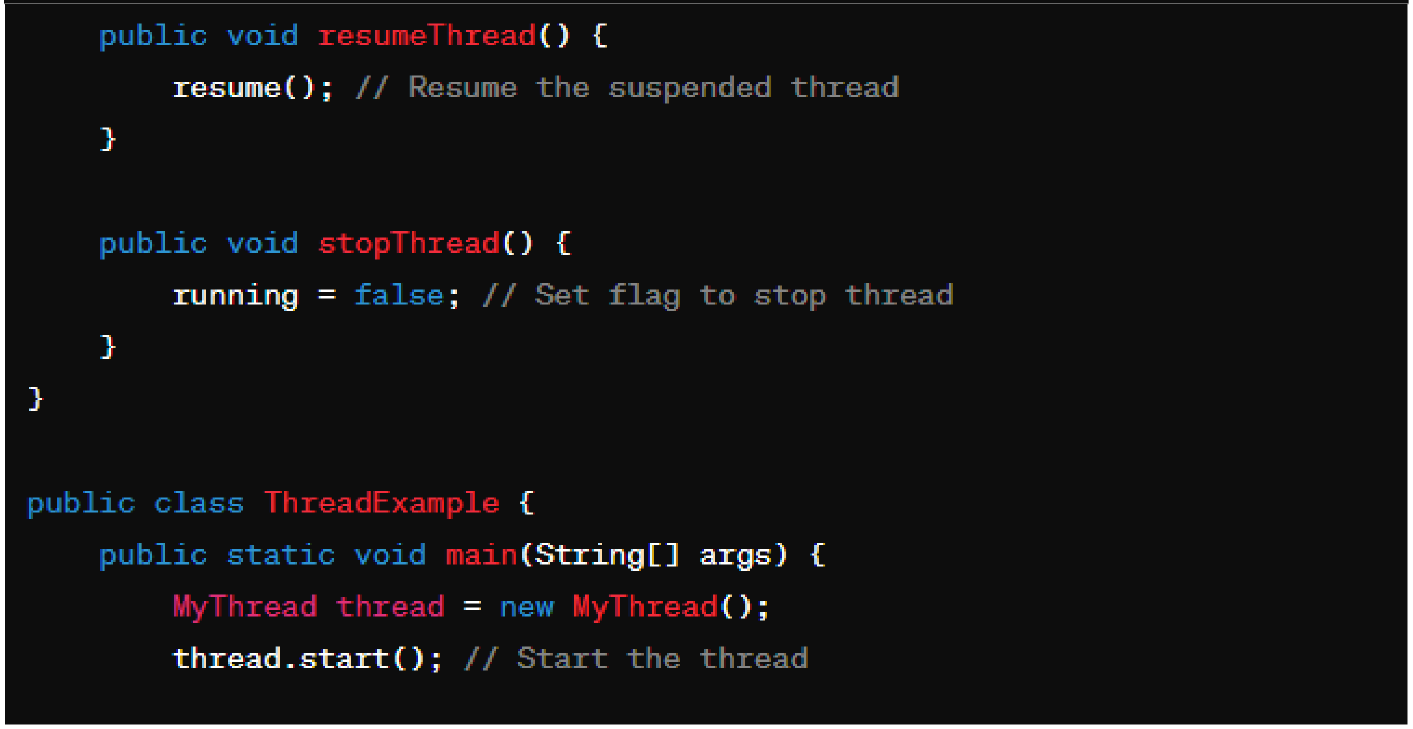
Reasons for Deprecation:

- **suspend()**: Can cause serious system failures, as it doesn't release locks on critical data structures, potentially leading to deadlock.
- **resume()**: Deprecated as it requires **suspend()** to work properly.
- **stop()**: Can cause system failures by leaving critical data structures in corrupted states.

Example Program:

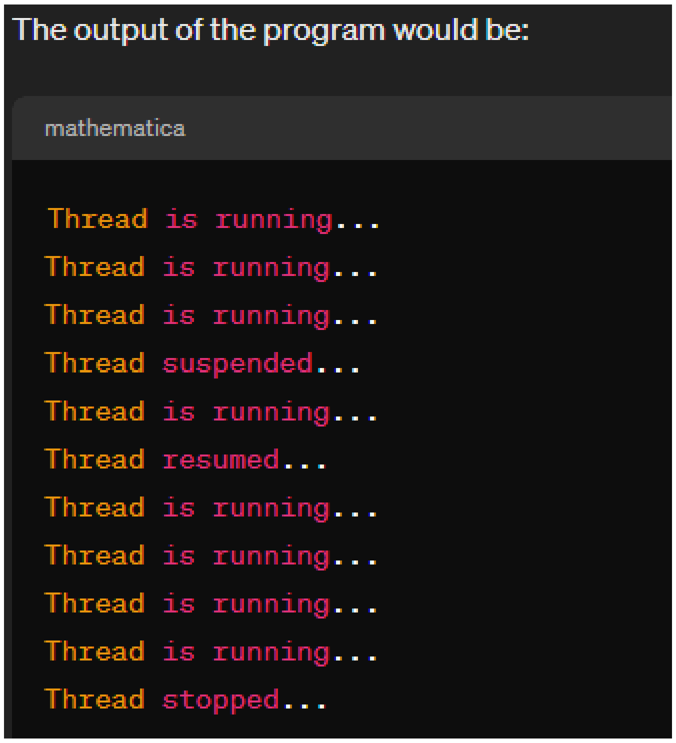


```java
class MyThread extends Thread {
    private volatile boolean running = true; // Flag to control thread execution

    @Override
    public void run() {
        while (running) {
            // Thread's execution logic
            System.out.println("Thread is running...");
            try {
                Thread.sleep(1000); // Simulate some work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void suspendThread() {
        suspend(); // Suspend the thread
    }
```



```java
    public void resumeThread() {
        resume(); // Resume the suspended thread
    }

    public void stopThread() {
        running = false; // Set flag to stop thread
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Start the thread
        // Suspend the thread after 3 seconds
        try {
            Thread.sleep(3000);
            thread.suspendThread();
            System.out.println("Thread suspended...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Resume the thread after 2 seconds
        try {
            Thread.sleep(2000);
            thread.resumeThread();
            System.out.println("Thread resumed...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Stop the thread after 5 seconds
        try {
            Thread.sleep(5000);
            thread.stopThread();
            System.out.println("Thread stopped...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

This program demonstrates suspending, resuming, and stopping a thread using the `suspendThread()`, `resumeThread()`, and `stopThread()` methods, respectively.


The output of the program would be:

mathematica

```
Thread is running...
Thread is running...
Thread is running...
Thread suspended...
Thread is running...
Thread resumed...
Thread is running...
Thread is running...
Thread is running...
Thread is running...
Thread stopped...
```

15. Explain Deadlock ?

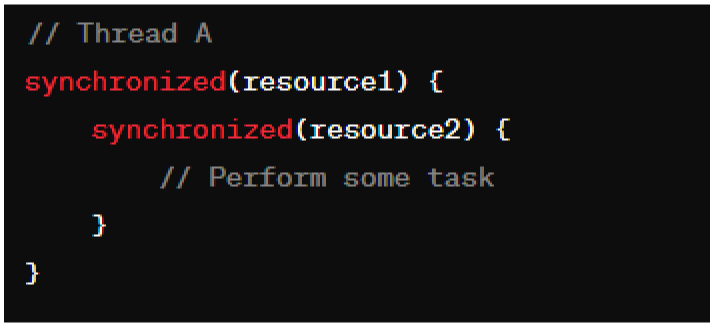
Ans:

Deadlock is a situation in multithreading where two or more threads are blocked forever, waiting for each other to release resources. This situation arises when two or more threads wait for resources that are held by each other, resulting in a cyclic dependency and none of the threads being able to proceed.

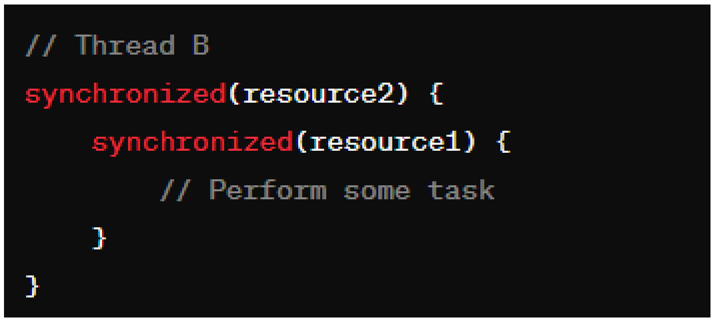
1. Deadlock occurs when two or more threads are waiting indefinitely for resources held by each other.
2. Each thread waits for the other thread to release a resource before it releases its own resource, leading to a stalemate situation.

Example: Consider two threads, Thread A and Thread B, each requiring two resources to complete their task. If Thread A holds Resource 1 and requests Resource 2, while Thread B holds Resource 2 and requests Resource 1, a deadlock can occur:

In this scenario, both threads are waiting for the other to release the resource it holds, resulting in a deadlock situation where neither thread can progress further.



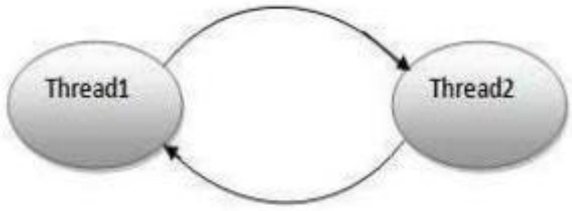
```
// Thread A
synchronized(resource1) {
    synchronized(resource2) {
        // Perform some task
    }
}
```



```
// Thread B
synchronized(resource2) {
    synchronized(resource1) {
        // Perform some task
    }
}
```

Deadlock

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- Deadlocks can occur in Java when the synchronized keyword causes the executing thread to block while waiting to get the lock, associated with the specified object.
- Since the thread might already hold locks associated with other objects, two threads could each be waiting for the other to release a lock. In such case, they will end up waiting forever.