**Generics:**

Generics in Java allow you to create classes, interfaces, and methods that operate on types that are specified as parameters. They provide a way to create reusable code that can work with any data type, while still providing compile-time type safety.

1. **Type Parameters**: Generics introduce type parameters, which are placeholders for types that are specified when the generic code is used. Type parameters are enclosed in angle brackets (<>) and are typically single uppercase letters, such as T, E, or K.
2. **Generic Classes**: You can create generic classes by declaring the type parameter in the class definition. This allows you to create classes that can work with different types of data.

**public class Box<T> {**

**private T value;**

**public T getValue() {**

**return value;**

**}**

**public void setValue(T value) {**

**this.value = value;**

**}**

**}**

**Generic Interfaces**: Similarly, you can create generic interfaces by declaring the type parameter in the interface definition.

**public interface List<T> {**

**void add(T element);**

**T get(int index);**

**}**

**Generic Methods:** In addition to generic classes and interfaces, you can also create generic methods within non-generic classes. Generic methods allow you to specify type parameters that are independent of the class's type parameters.

***public class Utils {***

***public static <T> T getElement(T[] array, int index) {***

***return array[index];***

***} }***

***Type Safety:*** Generics provide compile-time type safety, which means that the compiler can detect and report type errors at compile-time rather than at runtime. This helps prevent errors and bugs in your code.

***Type Erasure:*** Java generics use type erasure, which means that generic type information is erased at compile-time and is not available at runtime. This allows backward compatibility with pre-generic code but imposes some limitations on the use of generics, such as the inability to create generic arrays or access the type parameter at runtime.

Generics are widely used in Java for creating collections (e.g., List<T>, Map<K, V>) and for creating reusable code that operates on different types of data. They provide a powerful mechanism for writing flexible and type-safe code in Java.

**Example:**



**Multithreading:** Multithreading is a programming concept that allows a single process to perform multiple tasks concurrently, thus improving performance and responsiveness of applications. Here's a simplified explanation along with examples and some common methods:

Multithreading is a technique that enables a program to perform multiple tasks simultaneously, making use of the available CPU cores efficiently. Each task runs independently as a separate thread of execution within the same process.

**Example 1:** Consider a music player application. While playing music, the application might also need to download new songs, update the user interface, and handle user input. By using multithreading, the music playback can happen in one thread while other tasks like downloading and UI updates occur in separate threads, providing a smooth and responsive user experience.

**Example 2:** Web servers often use multithreading to handle multiple incoming requests concurrently. Each request is processed in a separate thread, allowing the server to serve multiple clients simultaneously without blocking.

**class MyThread extends Thread {**

**public void run() {**

**System.out.println("This is a thread.");**

**}**

**}**

**// Creating and starting a thread**

**MyThread thread = new MyThread();**

**thread.start();**

**Implementing Runnable Interface:**

**class MyRunnable implements Runnable {**

**public void run() {**

**System.out.println("This is a thread implemented using Runnable interface.");**

**}**

**}**

**// Creating and starting a thread using Runnable**

**Thread thread = new Thread(new MyRunnable());**

**thread.start();**

**Thread Synchronization (using synchronized keyword):** Thread synchronization is a technique used to control the access of multiple threads to shared resources in a multithreaded environment. It ensures that only one thread can access a particular resource at a time, preventing data corruption and maintaining consistency. Here's an explanation of thread synchronization:

**Need for Thread Synchronization:**

In a multithreaded environment, multiple threads may access shared resources concurrently. Without synchronization, this concurrent access can lead to race conditions and data inconsistencies. Thread synchronization mechanisms ensure that critical sections of code are executed atomically, avoiding conflicts between threads.

**Locks/Mutexes (Mutual Exclusion):** Using locks, threads can gain exclusive access to a resource. Only the thread holding the lock can execute the critical section of code.

Example: Java's synchronized keyword, Lock interface.

**Semaphores:** Semaphores maintain a count, allowing a certain number of threads to access a resource simultaneously. It can be used to control access to a pool of resources.

Example: Java's Semaphore class.

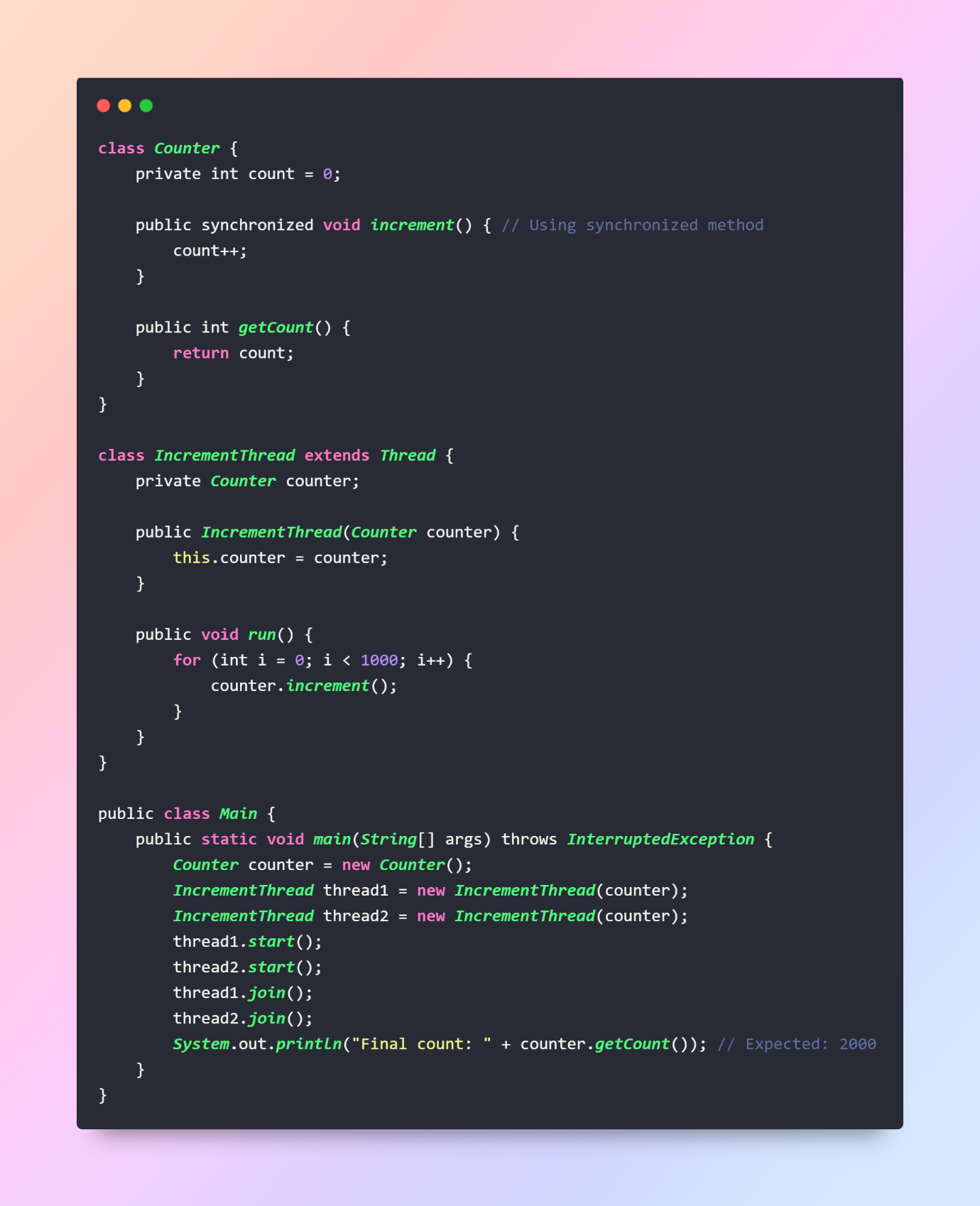
**Monitors:** Monitors combine data and procedures into a single entity. Only one thread can execute a monitor procedure at a time.

Example: Java's wait(), notify(), and notifyAll() methods.

**Resource Sharing:** Resource sharing refers to the situation where multiple threads or processes access and utilize the same resources concurrently. These resources could be data structures, files, memory, or hardware devices.

**Critical section:** A critical section is a part of a program where shared resources are accessed and manipulated by multiple threads or processes. It is a segment of code that must be executed atomically, meaning only one thread can execute it at a time to prevent race conditions and maintain data integrity.

Example Of Synchronisation:



Thread Priorities:

**Thread thread1 = new Thread(() -> {**

**System.out.println("Thread 1 is running.");**

**});**

**thread1.setPriority(Thread.MAX\_PRIORITY); // Set priority to maximum**

JAVA 8 Features: Java 8 introduced several new features and enhancements to the Java programming language, including lambda expressions, the Stream API, default methods in interfaces, and more. Here are definitions along with examples for some key features:

**1. Lambda Expressions:**

Lambda expressions introduce a concise way to express anonymous functions or function literals. They enable you to treat functionality as a method argument or create instances of functional interfaces. Lambda expressions are particularly useful in functional-style programming and enable more readable and maintainable code.

A computer screen shot of a program code

Description automatically generated

A screen shot of a computer program

Description automatically generated

**2. Stream API:**

The Stream API provides a fluent and functional approach to processing collections of objects. Streams allow you to perform complex data manipulation operations (such as filtering, mapping, and reducing) in a declarative manner, making code more concise and readable.

*List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);*

***// Using Stream API to find the sum of squares of even numbers***

***int sumOfSquares = numbers.stream()***

***.filter(n -> n % 2 == 0)***

***.map(n -> n \* n)***

***.reduce(0, Integer::sum);***

***System.out.println("Sum of squares of even numbers: " + sumOfSquares);***

3. Default Methods in Interfaces:

Java 8 introduced the ability to define default methods in interfaces. Default methods enable interfaces to provide concrete method implementations, allowing interface evolution without breaking existing implementations. They are particularly useful for adding new methods to interfaces without forcing existing classes to implement them.

A screenshot of a computer program

Description automatically generated

**4. Method References:**

Method references provide a shorthand notation for lambda expressions that simply call a method. They allow you to refer to methods without invoking them, making code more readable and concise.

**List<String> names = Arrays.asList("Alice", "Bob", "Charlie");**

**// Using lambda expression**

**names.forEach(name -> System.out.println(name));**

**// Using method reference**

**names.forEach(System.out::println);**

Functional Interfaces: Interfaces with exactly one abstract method, used for lambda expressions and method references.

* Functional interfaces are interfaces that have exactly one abstract method.
* They can be annotated with @FunctionalInterface to ensure that they only have one abstract method, though it's not required.
* Functional interfaces are often used in conjunction with lambda expressions.
* For example, the Runnable interface is a functional interface because it has only one abstract method (run()).
* But we can also write default and static method in our functional interface which cannot affect the Function interface implementation.

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod();

}

**Optional:** Optional is a container object that may or may not contain a non-null value.

It helps avoid NullPointerException by providing a way to handle null values more explicitly.

It encourages better programming practices by forcing developers to handle the absence of a value explicitly.

For example, Optional<Integer> optionalValue = Optional.ofNullable(someNullableValue);

**DateTime API:** Provides a comprehensive set of classes for date and time manipulation.

Example:

LocalDate today = LocalDate.now();

LocalDate tomorrow = today.plusDays(1);

**Parallel Streams:** Allows processing streams concurrently to improve performance.

int sum = numbers.parallelStream()

.filter(n -> n % 2 == 0)

.mapToInt(n -> n \* n)

.sum();