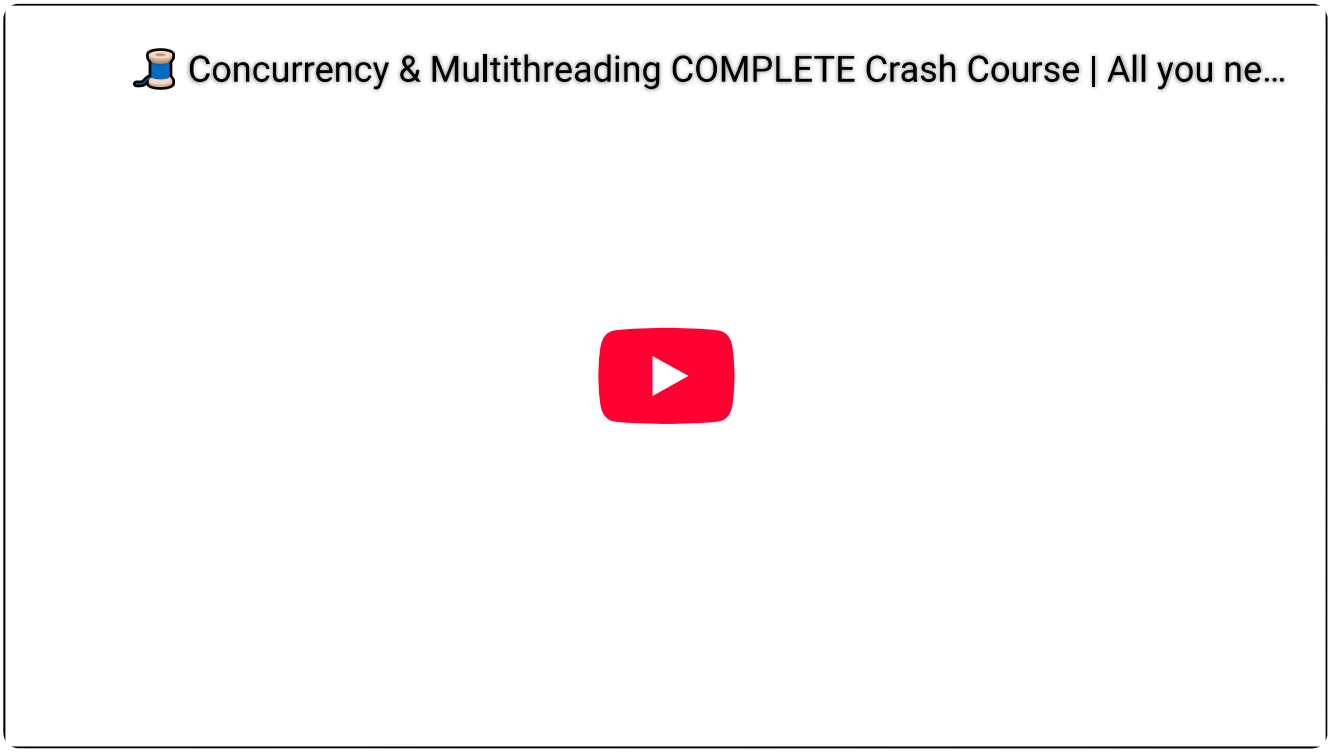
Threads - Thread class and Runnable Interface



Topic Tags:

system design lld

🐆⬛ **Github Codes Link**: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Java Threads 🧵🔄

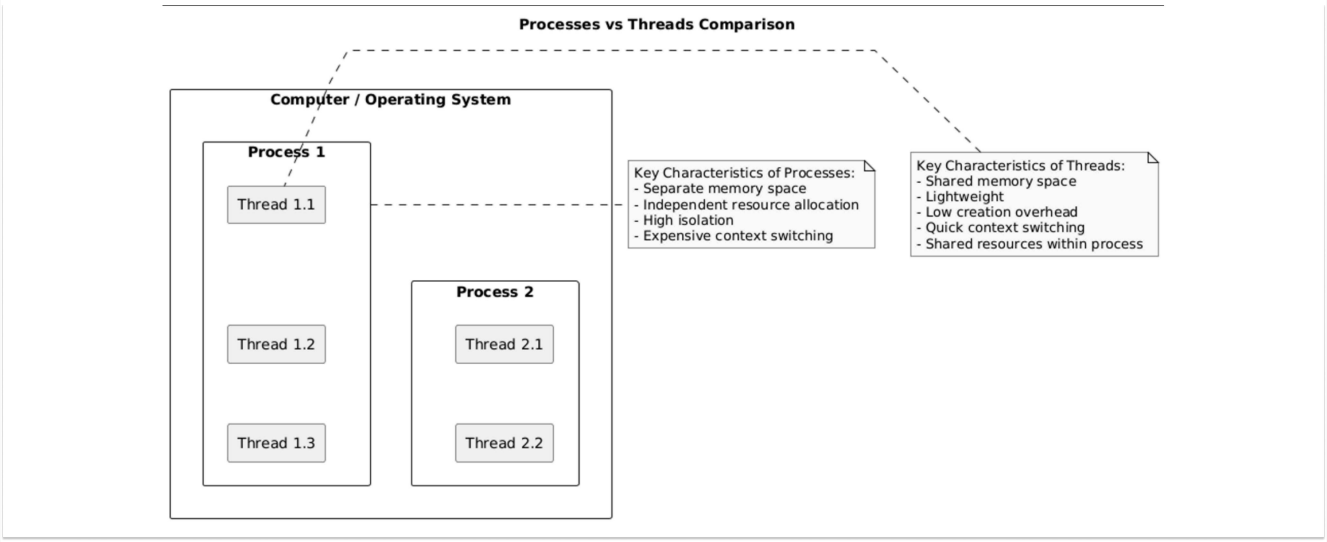
Threads are fundamental units of execution in Java that allow programs to perform multiple tasks concurrently. They enable developers to create responsive applications, utilize multi-core processors efficiently, and improve overall application performance. 🚀💻

Processes vs Threads: Understanding Concurrent Execution 🖥️🔄

Processes and threads are both fundamental concepts in concurrent computing, but they have distinct characteristics and use cases. Let's explore their similarities and differences to provide a comprehensive understanding.

Comparison Table: Processes vs Threads :

Characteristic	Process	Thread
Definition	An independent program with its own memory space	A lightweight unit of execution within a process
Memory	Separate memory space	Shared memory within the same process
Resource Allocation	Fully independent resource allocation	Shares resources with other threads in the same process
Overhead	High (creating a new process is resource-intensive)	Low (threads are lightweight and quick to create)
Communication	Inter-process communication requires complex mechanisms	Easy communication through shared memory
Isolation	High isolation between processes	Low isolation (threads can directly affect each other)
Switching Cost	Expensive context switching	Relatively inexpensive context switching
Failure Impact	One process crash doesn't typically affect others	Thread failure can potentially crash the entire process



Key Differences Explained 🔍

1. **Memory Management**

- Processes have separate memory spaces, providing strong isolation.
- Threads within the same process share memory, allowing efficient data sharing but requiring careful synchronization.

2.Resource Consumption

- Creating a new process is computationally expensive and requires significant system resources.
- Threads are lightweight and can be created and destroyed quickly with minimal overhead.

3.Communication

- Processes typically communicate through complex mechanisms like pipes, sockets, or message queues.
- Threads can communicate directly by sharing memory, making inter-thread communication more straightforward and faster.

4.Fault Tolerance

- Process crashes are isolated and don't necessarily affect other processes.
- A thread crash can potentially bring down the entire process and all its threads.

When to Use Processes vs Threads 🤔

Use Processes When:

- You need strong isolation between different parts of an application
- Running completely independent tasks
- Leveraging multiple CPU cores for separate computational tasks

Real Life Examples :

1. 🪟 **Web Browsers**: Modern browsers like Chrome run each tab as a separate process to ensure one crashing tab doesn’t affect others.
2. 📸 **Image Processing Pipelines**: Applications like Photoshop or video editors use multiple processes to handle large computations separately.
3. 🎮 **Game Engines**: Physics simulations and AI computations in games run in separate processes to utilize multiple CPU cores efficiently.

Use Threads When:

- You need to perform multiple tasks within the same application
- Tasks need to share common data quickly
- You want to improve responsiveness and performance of a single application

Real Life Examples :

1. 📱 **Mobile Apps**: A messaging app like WhatsApp uses threads to handle UI updates and background network requests simultaneously.
2. 🛒 **E-commerce Platforms**: Websites like Amazon use threads to allow multiple users to browse, add to cart, and checkout simultaneously while sharing inventory data.
3. 🎵 **Music Streaming Services**: Apps like Spotify use threads to keep the UI responsive while continuously buffering audio in the background.

Key Features of Threads 🔑

Concurrent Execution:

Multiple threads can run simultaneously, allowing programs to perform multiple tasks at once. 🔄,
📌 **Example :**
In a web browser, one thread can handle user interactions (scrolling, clicking), while another thread loads a
web page in the background. This prevents the UI from freezing while content is still loading.

Resource Sharing:

Threads within the same process share memory and resources, making communication between threads efficient. 🤝
📌 **Example** (for Resource Sharing):
In a text editor like Microsoft Word, multiple threads handle different tasks—one thread checks spelling and grammar, another auto-saves the document, while another processes user input. Since all threads share the same document data, resource sharing ensures efficiency without redundant memory usage.

Lightweight:

Threads require fewer resources compared to creating multiple processes. □
📌 **Example** (for Lightweight):
In a multiplayer online game, multiple threads manage player movements, background music, and network communication. Since creating a new process for each task would be costly, using threads keeps the game smooth and responsive while consuming fewer resources.

Why specifically Threads and not processes ?

Threads share memory space, making communication between them faster compared to processes. This ensures smooth and responsive performance without unnecessary duplication of resources.

Creating Threads in Java 🛠️

In Java, there are two primary ways to create and work with threads:

1. Extending the Thread Class 🧬

The Thread class provides the foundation for creating and managing threads in Java. By extending this class, you can override the run() method to define the code that will be executed in a separate thread.

Java

```java
class MyThread extends Thread {

    // Override the run method to define thread behavior
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread " + Thread.currentThread().getId()
            try {
                Thread.sleep(500); // Pause execution for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread(); // Create thread instance
        MyThread thread2 = new MyThread(); // Create another thread instan

        thread1.start(); // Start the first thread
        thread2.start(); // Start the second thread
    }
}
```

Output :

```
Thread id: 11 is running: 0
Thread id: 12 is running: 0
Thread id: 11 is running: 1
Thread id: 12 is running: 1
Thread id: 11 is running: 2
Thread id: 12 is running: 2
Thread id: 11 is running: 3
Thread id: 12 is running: 3
Thread id: 11 is running: 4
Thread id: 12 is running: 4
```

2. Implementing the Runnable Interface 🔄

The Runnable interface provides a more flexible approach to creating threads. It separates the task from the thread itself, promoting better object-oriented design and allowing a class to extend another class while still being runnable in a separate thread.

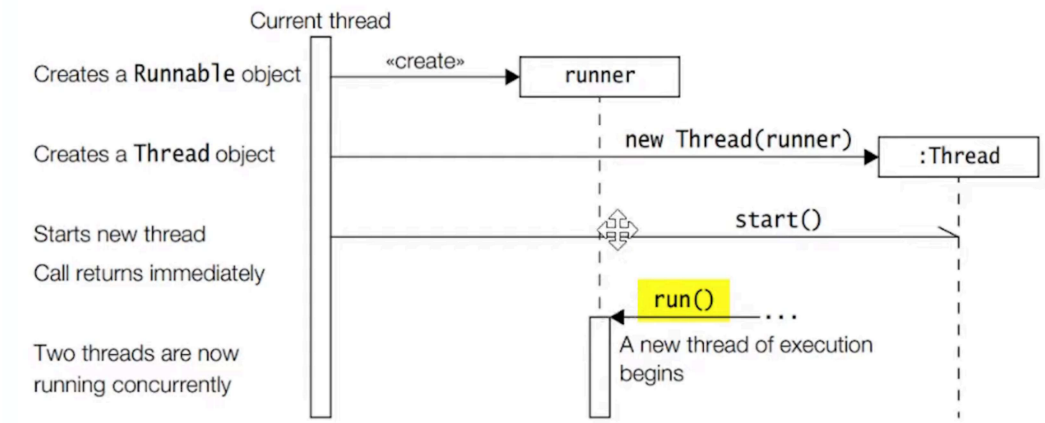
Java

```java
class MyRunnable implements Runnable {

    // Implement the run method from Runnable interface
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable " + Thread.currentThread().getId(
            try {
                Thread.sleep(500); // Pause execution for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable(); // Create runnable instanc

        Thread thread1 = new Thread(runnable); // Create thread with runna
        Thread thread2 = new Thread(runnable); // Create another thread wi

```

```
24          thread1.start(); // Start the first thread
25          thread2.start(); // Start the second thread
26      }
27  }
```

Output :

```
Runnable id: 11 is running: 0
Runnable id: 12 is running: 0
Runnable id: 11 is running: 1
Runnable id: 12 is running: 1
Runnable id: 11 is running: 2
Runnable id: 12 is running: 2
Runnable id: 11 is running: 3
Runnable id: 12 is running: 3
Runnable id: 11 is running: 4
Runnable id: 12 is running: 4
```



Thread vs Runnable: Which to Choose? 🤔

Feature	Thread	Runnable	Callable
Return Result	No	No	Yes
Throw Exceptions	No (only unchecked)	No (only unchecked)	Yes (checked & unchecked)
Creation Method	`new Thread().start()`	`new Thread(runnable).start()`	`executorService.submit(callable)`
Return Type	void	void	Generic type <V>
Method to Implement	`run()`	`run()`	`call()`
Introduced in Java	JDK 1.0	JDK 1.0	JDK 5
Works with	-	Thread	ExecutorService & Future

Extending Thread Class ⚖️

Advantages:

- Simpler to implement for beginners
- Direct access to Thread methods

Disadvantages:

- Limits inheritance (Java doesn't support multiple inheritance)
- Each task requires a new thread instance

Implementing Runnable Interface ⚖️

Advantages:

- Better object-oriented design
- Allows class to extend other classes
- Same Runnable instance can be shared across multiple threads
- More flexible for executor frameworks

Disadvantages:

- Slightly more code to write
- Indirect access to Thread methods

Using the Callable Interface 📞

The Callable interface, introduced in Java 5 as part of the concurrency utilities, provides a more powerful alternative to Runnable. Unlike Runnable, Callable can return results and throw checked exceptions.

Key Features of Callable 🔑

- Return Values: Callable tasks can return results, unlike Runnable tasks which return void.
- Exception Handling: Callable's call() method can throw checked exceptions, while Runnable's run() method cannot.
- Future Objects: Callable works with Future objects to retrieve results after task completion.

Checked vs Unchecked Exceptions in Java :

Checked Exceptions :

- Checked exceptions are exceptions that must be either declared in the method signature using throws or handled using try-catch.
- They are checked at compile-time.
- Examples:
- IOException (e.g., file not found)
- SQLException (e.g., database connection failure)
- InterruptedException (e.g., thread interruption)

Unchecked Exceptions :

- Unchecked exceptions are runtime errors that do not require explicit handling.
- They are checked at runtime, meaning they occur due to logical errors in the program.
- Examples:
- NullPointerException (e.g., calling a method on null)
- ArrayIndexOutOfBoundsException (e.g., accessing an invalid array index)
- ArithmeticException (e.g., division by zero)

Key Difference:

Checked exceptions enforce error handling at compile-time, while unchecked exceptions indicate programming mistakes that occur at runtime. 🚀

How Callable Works ?

The Callable interface works with the ExecutorService framework rather than directly extending Thread. While we'll explore the ExecutorService framework in detail in later parts

of our course, below is a complete example showing how to create and manage threads using Callable:

Implementing Callable Interface 💻

Java

```java
class MyCallable implements Callable<String> {

    private final String name;

    public MyCallable(String name) {
        this.name = name;
    }

    // Implement the call method from Callable interface
    @Override
    public String call() throws Exception {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < 5; i++) {
            result.append("Callable ").append(name)
                  .append(" is running: ").append(i).append("n");
            Thread.sleep(500); // Pause execution for 500 milliseconds
        }
        return result.toString(); // Return the result as a String
    }
}


public class CallableExample {
    public static void main(String[] args) {
        // Create ExecutorService with a fixed thread pool
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Create Callable instances
        Callable<String> callable1 = new MyCallable("Task 1");
        Callable<String> callable2 = new MyCallable("Task 2");

        try {
            // Submit Callable tasks to the executor and get Future object
```

```
34              Future<String> future1 = executor.submit(callable1);
35              Future<String> future2 = executor.submit(callable2);
36
37              // Get results from Future objects
38              System.out.println("Result from first task:");
39              System.out.println(future1.get()); // Blocks until the task co
40
41              System.out.println("Result from second task:");
42              System.out.println(future2.get()); // Blocks until the task co
43
44          } catch (InterruptedException | ExecutionException e) {
45              System.out.println("Task execution interrupted: " + e.getMessa
46          } finally {
47              // Shutdown the executor
48              executor.shutdown();
49          }
50      }
51  }
```

Output:

```
Result from first task:
Callable Task 1 is running: 0
Callable Task 1 is running: 1
Callable Task 1 is running: 2
Callable Task 1 is running: 3
Callable Task 1 is running: 4


Result from second task:
Callable Task 2 is running: 0
Callable Task 2 is running: 1
Callable Task 2 is running: 2
Callable Task 2 is running: 3
Callable Task 2 is running: 4
```

Runnable Cannot Throw Checked Exceptions like callable as :

- The run() method in Runnable does not allow checked exceptions to be thrown.
- If an exception needs to be handled, it must be caught inside the run() method itself.

Callable vs. Thread vs. Runnable Comparison :

Best Practices for Thread Implementation

1. **Use Runnable over Thread extension** when possible for better design principles.
2. **Keep synchronization minimal** to avoid performance bottlenecks.
3. **Handle interruptions properly** to ensure graceful thread termination.
4. **Avoid thread starvation** by balancing priorities and resource allocation.
5. **Use higher-level concurrency utilities** from java.util.concurrent package for complex scenarios.

Interview Questions

1. What is the difference between start() and run() methods?
Answer: The start() method begins thread execution and calls the run() method, while the run() method simply contains the code to be executed. Directly calling run() won't create a new thread; it will execute in the current thread.

Java

```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().get
    }
}


public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();  // Starts a new thread


        MyThread t2 = new MyThread();
        t2.run();   // Runs in the main thread
    }
}
```

2. Can we call the start() method twice on the same Thread object?

Answer: No, calling start() twice on the same Thread object will throw an IllegalThreadStateException. A thread that has completed execution cannot be restarted. ⚠️

Java

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

    public static void main(String[] args) {
        TestThread t = new TestThread();
        t.start(); // Works fine
        t.start(); // Throws IllegalThreadStateException
    }
}
```

3. What is thread safety and how can it be achieved?

Answer: Thread safety refers to code that functions correctly during simultaneous execution by multiple threads. It can be achieved through synchronization, immutable objects, concurrent collections, atomic variables, and thread-local variables. 🔒

4. What happens if an exception occurs in a thread's run method?

Answer: If an uncaught exception occurs in a thread's run() method, the thread terminates. The exception doesn't propagate to the parent thread and doesn't affect other threads. ❌

Java

```java
class MyThread extends Thread {
    public void run() {
        try {
            throw new RuntimeException("Exception in thread");
        } catch (Exception e) {
            System.out.println("Caught exception in thread: " + e.getMessa
        }
    }
}

```

```
11 public class Main {
12     public static void main(String[] args) {
13         MyThread t = new MyThread();
14         t.start(); // Starts a separate thread
15         System.out.println("Main thread is running");
16     }
17 }
```

Explanation:

- If you call t.run(); directly, it behaves like a normal method call and executes in the main thread, so the exception would be

thrown in the main thread.

- By calling t.start();, the thread runs separately, and if an exception occurs in run(), it terminates that thread without

affecting the main thread.

- The main thread continues execution and prints "Main thread is running" even if the child thread throws an exception.

Output :

```
Main thread is running
Exception in thread "Thread-0" java.lang.RuntimeException: Exception in thread
    at MyThread.run(Main.java:4)
    at java.base/java.lang.Thread.run(Thread.java:833)
```

5. What's the difference between sleep() and wait()?

Answer: sleep() causes the current thread to pause for a specified time without releasing locks. wait() causes the current thread
to wait until another thread invokes notify() or notifyAll() on the same object, and it releases the lock on the object. 😴

Example 1 : sleep() Example :

Code :

Java

```java
class SleepExample {
    public static void main(String[] args) {
        System.out.println("Thread is going to sleep...");
        try {
            Thread.sleep(2000); // Sleep for 2 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread woke up after sleeping.");
    }
}
```

Output :

```
Thread is going to sleep...
(Thread pauses for 2 seconds)
Thread woke up after sleeping.
```

- **Simulation Explanation:**

1. The main thread prints "Thread is going to sleep...".
2. It pauses execution for 2 seconds (sleep(2000)).
3. After 2 seconds, execution resumes normally, printing "Thread woke up after sleeping."
4. sleep() does NOT release any locks, meaning other threads can't access synchronized resources during sleep.

- **What Happens to the Resource a Thread Was Holding when the sleep() method is called?**
- When a thread calls sleep(), it **pauses execution** for the specified time.
- However, it **does NOT release any locks** it was holding.
- Other threads **cannot access synchronized resources** held by the sleeping thread.

Example 2: wait() – Pausing and Waiting for a Notification :
Code :

Java

```
class SharedResource {
    synchronized void waitExample() {
        System.out.println(Thread.currentThread().getName() + " is waiting
        try {
            wait(); // Releases the lock and waits
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " resumed af
    }


    synchronized void notifyExample() {
        System.out.println("Notifying a waiting thread...");
        notify(); // Wakes up one waiting thread
    }
}


public class WaitNotifyExample {
    public static void main(String[] args) {
        SharedResource shared = new SharedResource();


        // Thread 1 (Waits)
        Thread t1 = new Thread(() -> shared.waitExample(), "Thread-1");

        // Thread 2 (Notifies after 2 seconds)
        Thread t2 = new Thread(() -> {
            try {
                Thread.sleep(2000); // Ensure Thread-1 goes to wait state
                shared.notifyExample();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "Thread-2");


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

```
41      }
42  }
```

Output :

```
Thread-1 is waiting...
Notifying a waiting thread...
Thread-1 resumed after notify.
```

Simulation Explanation:

1. Thread-1 calls wait() and enters the waiting state, releasing the lock.
2. Thread-2 starts and sleeps for 2 seconds to simulate delay.
3. After 2 seconds, Thread-2 calls notify(), waking up Thread-1.
4. Thread-1 resumes execution after wait() and prints "Thread-1 resumed after notify."

- **What Happens to the Resource a Thread Was Holding when the wait() method is called?**

1. When a thread calls wait(), it releases the lock on the synchronized object it was holding.
2. Other threads can now acquire the lock and continue execution.
3. The waiting thread remains idle until another thread calls notify() or notifyAll().

- **What Happens to the Idle Thread Once notify() or notifyAll() is Called?**

When notify() or notifyAll() is called, the **waiting thread does not immediately start running**. Instead, it follows these steps:

1. When another thread calls notify(), one waiting thread is **moved to the Blocked (or Runnable) State**, but it does **not** start execution immediately.
2. The notified thread cannot resume execution until it successfully **acquires the lock** on the synchronized object.
3. If multiple threads are waiting, only one gets notified by notify(), while notifyAll() wakes up all waiting threads (but they still compete for the lock).
4. Once the thread reacquires the lock, it continues execution from where it called wait().

- **What If We Use notifyAll()?**

If we replace notify(); with notifyAll();, all **waiting threads** will be notified, but only **one** will acquire the lock first as **they will**
compete for the lock, and execution depends on the thread scheduler.

6. What is the Callable interface, and how does it differ from Runnable?
Answer: Callable is a functional interface introduced in Java 5 as part of the concurrency utilities. The key differences from Runnable are:

- Callable's call() method can return a result (it's a parameterized type), while Runnable's run() method returns void
- Callable's call() method can throw checked exceptions, while Runnable's run() method cannot
- Callable works with Future objects to handle the results asynchronously

7. Can you use Callable with standard Thread objects?
Answer: No, you cannot directly use Callable with the Thread class. Callable is designed to work with the **ExecutorService** framework. Thread class only accepts Runnable objects. However, you can adapt a Callable to work with Thread by creating a Runnable that executes the Callable and stores its result:

Java

```java
class MyRunnable implements Runnable {
    private Callable<Integer> callable;
    public MyRunnable(Callable<Integer> callable) {
        this.callable = callable;
    }

    public void run() {
        try {
            System.out.println(callable.call());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Conclusion 🎯

Threads are powerful tools for creating concurrent applications in Java. Understanding the Thread class and Runnable interface is essential for effective multi-threaded programming. By choosing the right approach based on your application's needs, you can write efficient, responsive, and robust Java applications. With proper thread management and synchronization, you can fully harness the power of modern multi-core processors and create applications that perform multiple tasks simultaneously. 🚀🧵