

Threads

- DO NOT DISTRIBUTE –

- DO NOT COPY – ALL RIGHTS RESERVED TO THE AUTHOR –

Introduction

A **thread** is the smallest unit of execution within a process. It is more “lightweight” than a full process because multiple threads within the same process can share memory and other resources. Using multiple threads lets you run tasks in parallel, improving efficiency and responsiveness. For example, in a web browser, one thread might handle rendering a page while another listens for user input. If one thread is blocked (e.g., waiting for data), the other can continue operating.

Multitasking in Operating Systems

1. Process-Based Multitasking (Multiprocessing)

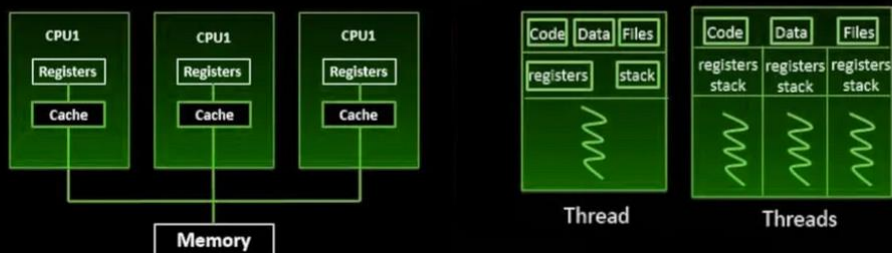
- Each process is heavyweight, with its own memory space.
- Switching from one process to another is relatively slow because it involves updating registers, memory maps, etc.

2. Thread-Based Multitasking (Multithreading)

- Threads share a single process’s memory.
- Switching among threads is faster, and communication costs between threads are lower.



Multiprocessing vs Multithreading



Thread Life Cycle

Java threads typically move through these states:

1. New

The thread is created but not yet started (`new Thread(...)` but `start()` not called).

2. Runnable

The thread is ready to run and waiting for CPU time. It enters this state after you call `start()`.

3. Running

The thread is actively executing code. After its CPU time slice ends, it may go back to **Runnable**.

4. Blocked/Waiting

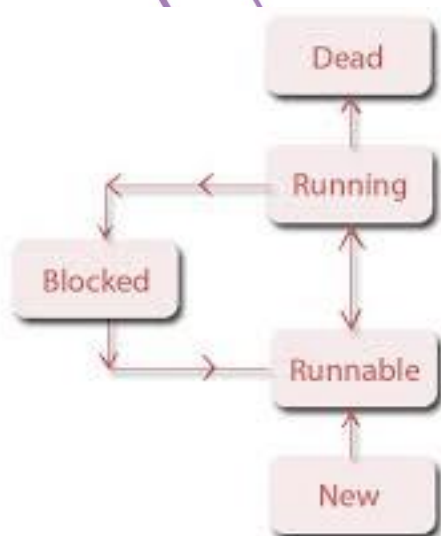
The thread is inactive because it's waiting for a resource (e.g., I/O or a lock). Once that resource is available, the thread returns to **Runnable**.

5. Timed Waiting

The thread is inactive for a specified time (e.g., using `Thread.sleep(milliseconds)` or a timed wait on a lock). This prevents indefinite starvation.

6. Terminated

The thread finishes its task (normal termination) or is stopped by an unhandled exception (abnormal termination). Once terminated, it cannot be restarted.



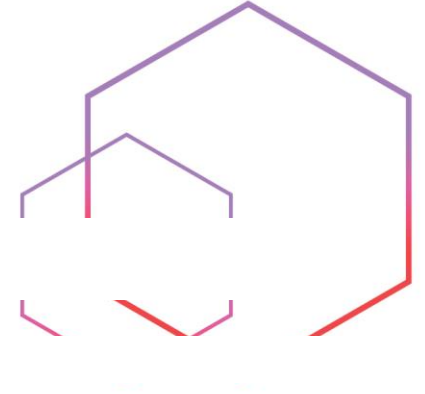
Lifecycle of a thread

Threads in Java



```
}  
}
```

- Override `run()` with the code you want the thread to execute.
- Call `start()`, not `run()`, to create a new thread of execution.



2. Implementing the `Runnable` Interface

```
class MyTask implements Runnable {  
    public void run() {  
        // Code to run in this thread  
    }  
}  
  
public class ThreadExample2 {  
    public static void main(String[] args) {  
        Thread t = new Thread(new MyTask());  
        t.start();  
    }  
}
```

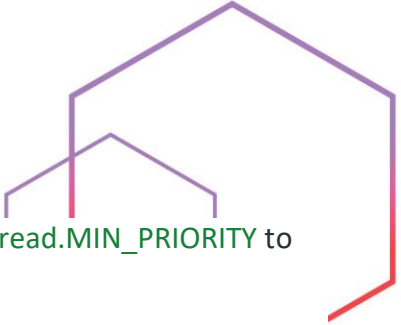
- This approach allows more flexibility (e.g., your class can extend another class while still being runnable).

Summary

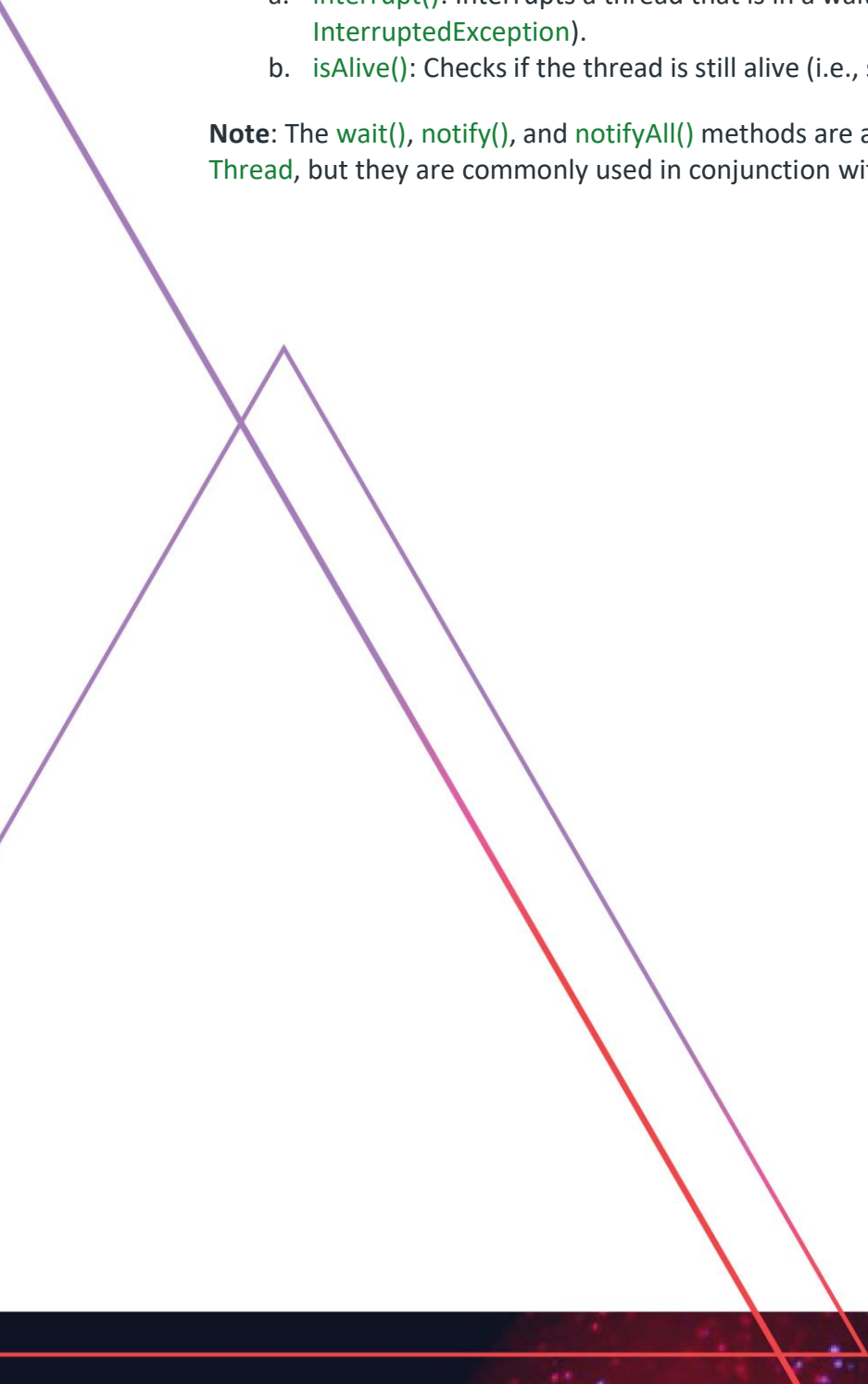
1. **Thread vs. Process:** Threads share the same memory space, leading to faster context switching and easier data sharing.
2. **Thread Life Cycle:** New → Runnable → Running → Blocked/Waiting/Timed Waiting → Terminated.
3. **Creation Methods:** Extend `Thread` or implement `Runnable`. Always call `start()` to spawn a new thread, rather than `run()` which just calls the method on the current thread.
4. **Why Use Threads?:**
 - a. Parallel execution (increasing efficiency).
 - b. Better responsiveness (a blocked thread doesn't halt the entire application).

List of Common Methods in the Thread Class

1. **Constructors**
 - a. `Thread()`: Creates a new thread without specifying a `Runnable` or a name.
 - b. `Thread(Runnable target)`: Creates a new thread with the specified `Runnable` object.
 - c. `Thread(Runnable target, String name)`: Creates a new thread with a `Runnable` and a given name.
2. **Starting & Running**
 - a. `start ()`: Creates a new thread of execution and calls the thread's `run()` method.
 - b. `run()`: The entry point for the thread's logic (typically overridden in a subclass or in the `Runnable`).
3. **Naming**
 - a. `getName()`: Returns the name of the thread.
 - b. `setName(String name)`: Sets the name of the thread.
4. **Priority**

- 
- a. `getPriority()`: Returns the priority of the thread (int from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`).
 - b. `setPriority(int newPriority)`: Sets the priority of the thread.
5. **Current Thread (Static)**
- a. `currentThread()`: Returns a reference to the currently executing thread.
6. **Joining/Waiting for Another Thread**
- a. `join()`: Causes the current thread to wait until the thread on which `join()` is called finishes.
 - b. `join(long millis)`: Waits at most `millis` time for the thread to finish.
7. **Sleeping**
- a. `sleep(long millis)`: Pauses (puts the current thread into a timed waiting state) for the specified time.
8. **Interruptions & Status**
- a. `interrupt()`: Interrupts a thread that is in a waiting or sleeping state (throws `InterruptedException`).
 - b. `isAlive()`: Checks if the thread is still alive (i.e., started but not yet terminated).

Note: The `wait()`, `notify()`, and `notifyAll()` methods are actually defined in `Object`, not in `Thread`, but they are commonly used in conjunction with threads and synchronization.



Project Demonstrating Thread Methods

Below is a small example that shows how to use some of these methods: creating threads, naming/renaming them, setting priorities, joining, and sleeping.

1. Thread Class (Custom Worker)

```
class MyWorker extends Thread {
    public MyWorker(String name) {
        super(name); // Set initial thread name
    }
    @Override
    public void run() {
        System.out.println("Thread started: " + this.getName()
            + ", Priority: " + this.getPriority());
        try {
            // Sleep for a short time to simulate work
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Thread " + this.getName() + " was interrupted.");
        }
        System.out.println("Thread finishing: " + this.getName());
    }
}

public class ThreadMethodsDemo {
    public static void main(String[] args) {
        // 1. Create a thread with a given name
        MyWorker worker1 = new MyWorker("Worker-1");

        // 2. Create a second thread without a name, then rename it
        MyWorker worker2 = new MyWorker("TempName");
        worker2.setName("Worker-2");

        // 3. Set priorities
        worker1.setPriority(Thread.MIN_PRIORITY); // 1
        worker2.setPriority(Thread.MAX_PRIORITY); // 10

        // 4. Start the threads
        System.out.println("Starting threads...");
        worker1.start();
        worker2.start();

        // 5. Demonstrate currentThread()
        Thread current = Thread.currentThread();
        System.out.println("Current thread is: "
            + current.getName()
            + " (priority "
            + current.getPriority() + ")");
    }
}
```

// 6. Wait (join) for both threads to finish

```
try {  
    System.out.println("Main thread waiting for Worker-1 and Worker-2 to finish...");  
    worker1.join();  
    worker2.join();  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

// 7. Check status

```
System.out.println("Is Worker-1 alive? " + worker1.isAlive());  
System.out.println("Is Worker-2 alive? " + worker2.isAlive());
```

// 8. Sleep the main thread for demonstration

```
System.out.println("Main thread sleeping for 500ms...");  
try {  
    Thread.sleep(500);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

```
System.out.println("Main thread finished.");
```

```
}
```

<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>