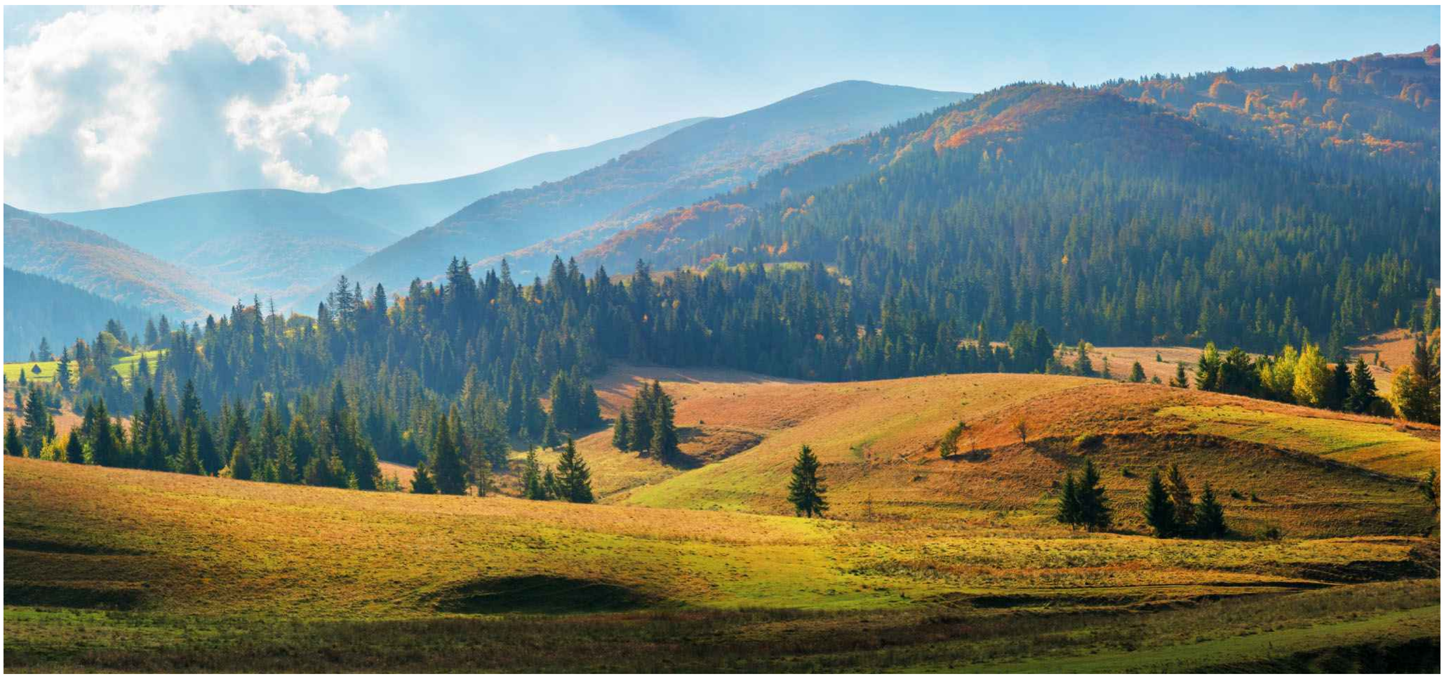
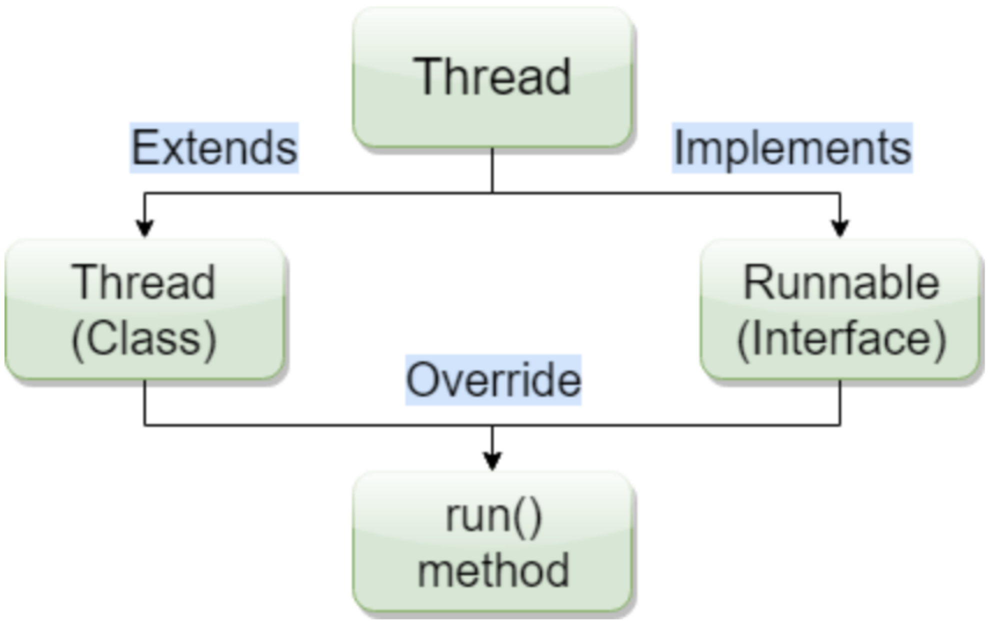


34-Thread Implementation

Author: Vincent Lau

Learning Objectives

> Understand the ways to implement threads in Java programming



Introduction

A `thread` **in Java is the smallest unit of execution within a program**. Threads allow a program to perform **multiple tasks** concurrently, enabling better **utilization** of resources in **multi-core processors**. Java provides built-in support for working with threads through the `java.lang.Thread` class.

Initialize Thread

There are two ways to create and start threads in Java:

Extending the Thread Class

You can create a new class that extends the `Thread` class and override its `run()` method to define the code that the thread will execute.

```java
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Start the thread's execution
    }
}
```

Implementing the Runnable Interface

By Class

You can create a class that implements the `Runnable` interface and provides the thread's logic in the `run()` method.

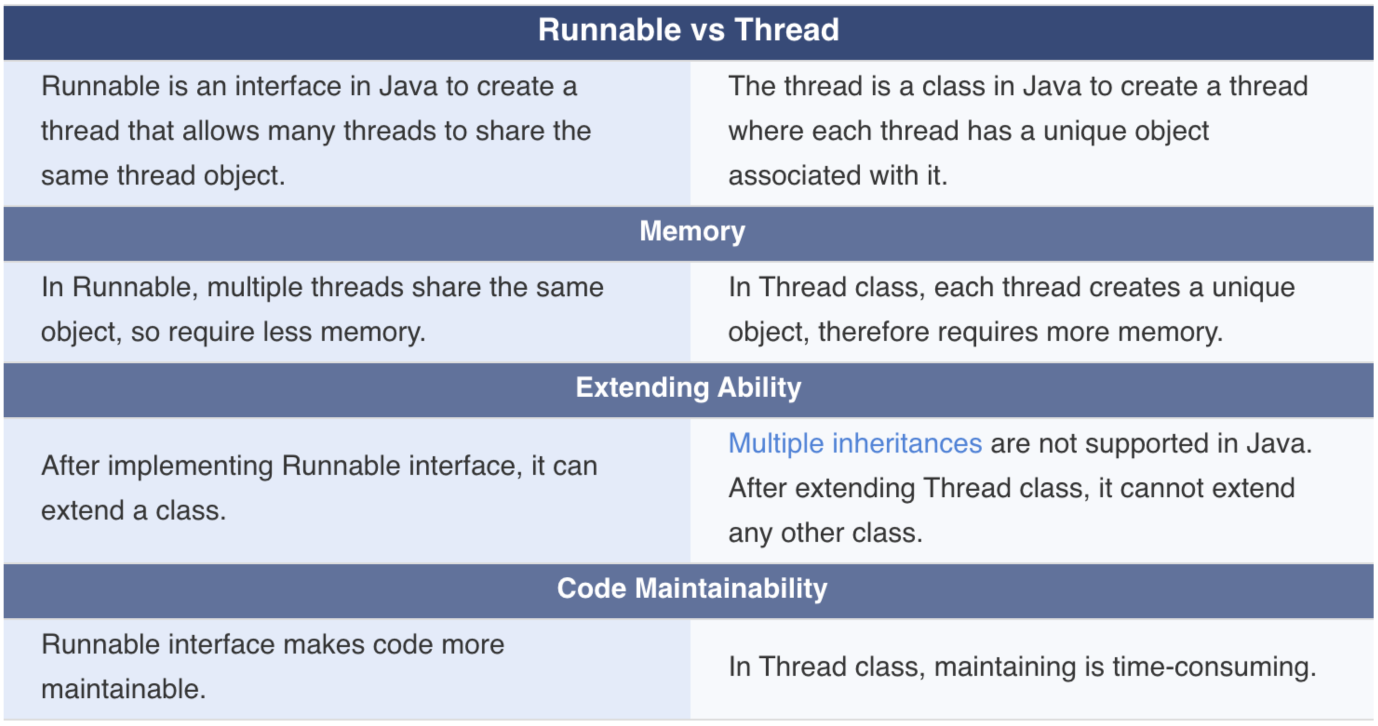
```java
class Task implements Runnable {

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        Task task = new Task();
        Thread thread = new Thread(task);
        thread.start(); // Start the thread's execution
    }
}
```

By Lambda

You can create the runnable object directly by Lambda expression that implements the `Runnable` interface and provides the thread's logic in the `run()` method.

```java
public class ThreadExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread: " + i);
            }
        };
        Thread thread = new Thread(task);
        thread.start(); // Start the thread's execution
    }
}
```

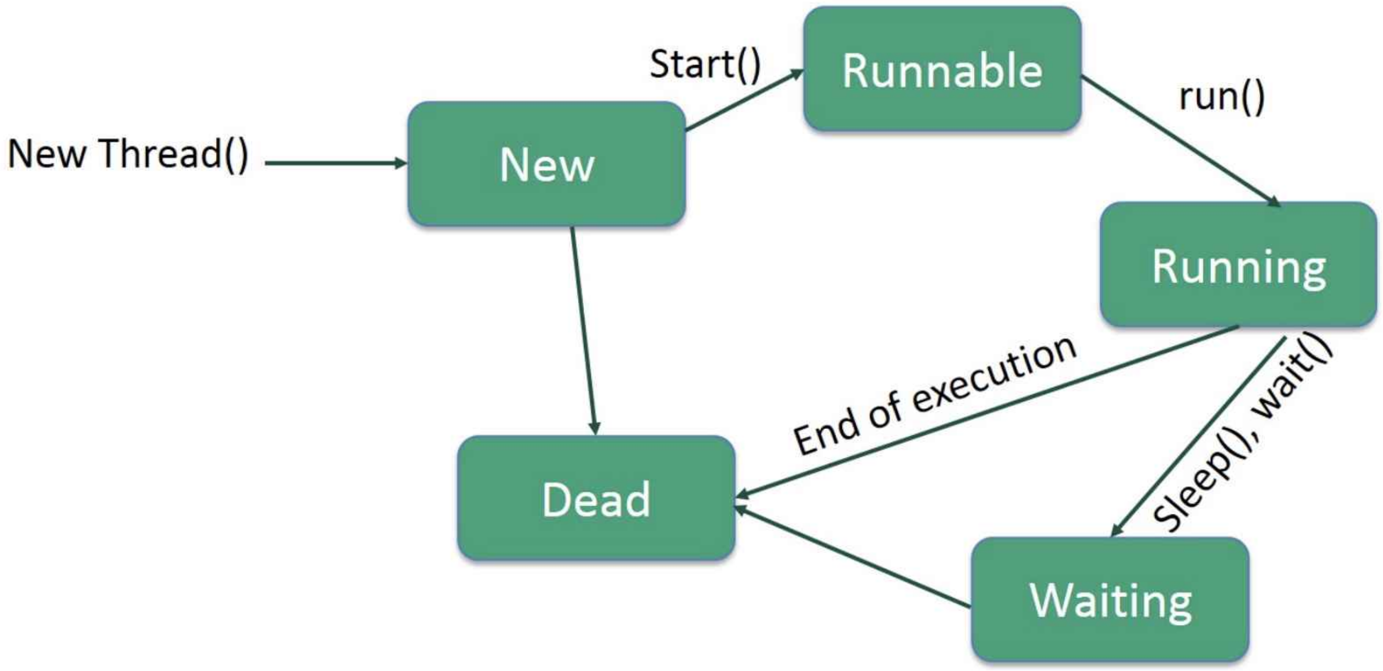
Differences



Runnable vs Thread	
Runnable is an interface in Java to create a thread that allows many threads to share the same thread object.	The thread is a class in Java to create a thread where each thread has a unique object associated with it.
Memory	
In Runnable, multiple threads share the same object, so require less memory.	In Thread class, each thread creates a unique object, therefore requires more memory.
Extending Ability	
After implementing Runnable interface, it can extend a class.	Multiple inheritances are not supported in Java. After extending Thread class, it cannot extend any other class.
Code Maintainability	
Runnable interface makes code more maintainable.	In Thread class, maintaining is time-consuming.

Thread States

Threads in Java go through different states during their lifecycle, including `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, and `TERMINATED`.



Thread Synchronization

When multiple threads access shared resources, synchronization is required to avoid race conditions and data corruption. Java provides synchronization mechanisms like the `synchronized` keyword and `Object`'s `wait()` and `notify()` methods. They are typically used when one thread needs to wait for a specific condition to be met before proceeding, and another thread needs to signal when that condition has been fulfilled.

Code Example

```java
class SharedResource {
    private boolean hasData = false; // No data by default

    public synchronized void produce() throws InterruptedException {
        while (hasData) {
            wait(); // Wait for the consumer to consume
        }

        System.out.println("Producing data...");
        Thread.sleep(1000); // Simulate some processing
        hasData = true;

        notify(); // Notify the waiting consumer
    }

    public synchronized void consume() throws InterruptedException {
        while (!hasData) {
            wait(); // Wait for the producer to produce
        }

        System.out.println("Consuming data...");
        Thread.sleep(1000); // Simulate some processing
        hasData = false;

        notify(); // Notify the waiting producer
    }
}

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

        Thread producerThread = new Thread(() -> {
            try {
                sharedResource.produce();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                sharedResource.consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        producerThread.start();
        consumerThread.start();
    }
}
// Output
// Producing data...
// Consuming data...
```

1. We have a `SharedResource` class that contains a boolean flag `hasData` indicating whether data is available for consumption.
2. The `produce()` method is called by the producer thread. It enters a synchronized block, checks if data is already available, and if so, waits for the consumer to consume.
3. If data is not available, the producer simulates data production and then sets `hasData` to `true` before notifying the consumer thread.
4. The `consume()` method is called by the consumer thread. It enters a synchronized block, checks if data is available, and if not, waits for the producer to produce.
5. If data is available, the consumer simulates data consumption, sets `hasData` to `false`, and then notifies the producer thread.
6. The producer and consumer threads are started separately.

This example demonstrates how `wait()` and `notify()` can be used to coordinate the activities of two threads. Keep in mind that proper synchronization and handling of exceptions are important in real-world scenarios.

Thread Priority

Threads can have priority levels that influence their execution order. Higher-priority threads are more likely to be executed before lower-priority threads, but priority is not guaranteed to determine exact execution order.

```java
Thread thread1 = new Thread(() -> {
    // Thread logic
});
thread1.setPriority(Thread.MIN_PRIORITY);
```

```java

Thread thread2 = new Thread(() -> {
    // Thread logic
});
thread2.setPriority(Thread.MAX_PRIORITY);
```

Thread Joining

The `join()` method is used to wait for a thread to complete its execution before the current thread continues. It throws InterruptedException.

```java
Thread thread1 = new Thread(() -> {
    // Thread 1 logic
});

Thread thread2 = new Thread(() -> {
    // Thread 2 logic
});

thread1.start();
thread2.start();

try {
    thread1.join(); // Wait for thread1 to complete
    thread2.join(); // Wait for thread2 to complete
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Thread Sleeping

You can use the `Thread.sleep()` method to make a thread sleep for a specified amount of time (ms).

```java
try {
    Thread.sleep(1000); // Sleep for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Code Example

Introducing getId(), getName(), getPriority(), sleep() methods.

```java
public class ThreadInfoPrinter {

    public void threadInfo() {
        Thread currentThread = Thread.currentThread();
        long threadId = currentThread.getId();
        String threadName = currentThread.getName();
        int threadPriority = currentThread.getPriority();

        System.out.println("Thread ID: " + threadId);
        System.out.println("Thread Name: " + threadName);
        System.out.println("Thread Priority: " + threadPriority);

        for (int i = 1; i <= 5; i++) {
            System.out.println(currentThread.getName() + ": " + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ThreadInfoPrinter printer = new ThreadInfoPrinter();

        Thread thread1 = new Thread(() -> {
            printer.threadInfo(); // same object, different thread
        });
        Thread thread2 = new Thread(() -> {
            printer.threadInfo(); // same object, different thread
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Summary

Threads in Java allow for concurrent execution of tasks and are crucial for taking advantage of multi-core processors. You can create threads by extending the `Thread` class or implementing the `Runnable` interface. Thread synchronization, priority, joining, and sleeping are important concepts when working with threads. Remember to manage resources and avoid race conditions when multiple threads access shared data.