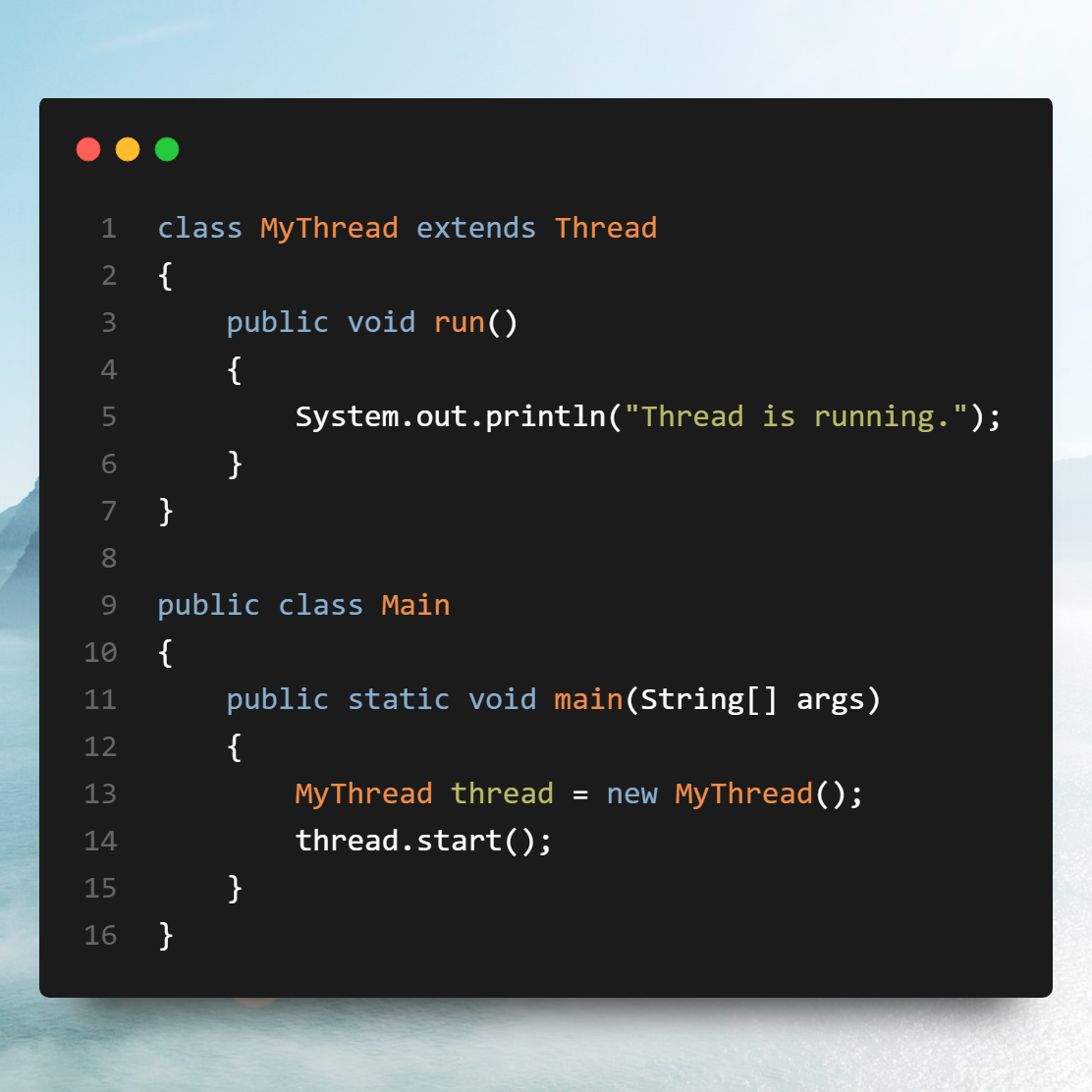


Separation of logic: By implementing Runnable, you decouple the task from the actual thread control. This promotes clean design because the task (business logic) is independent of how it is executed.

Multiple inheritance: Java doesn't allow multiple inheritance with classes, but it does allow you to implement multiple interfaces. So, if your class already extends another class, you can't extend Thread directly, but you can still implement Runnable.

Reusability: If you implement Runnable, you can reuse the same object in multiple threads. The same instance of the Runnable can be passed to different Thread objects, making it more flexible.

Use case: Use Runnable when you want to separate the thread's behaviour (execution logic) from the actual thread, or when your class already extends another class.



Advantages

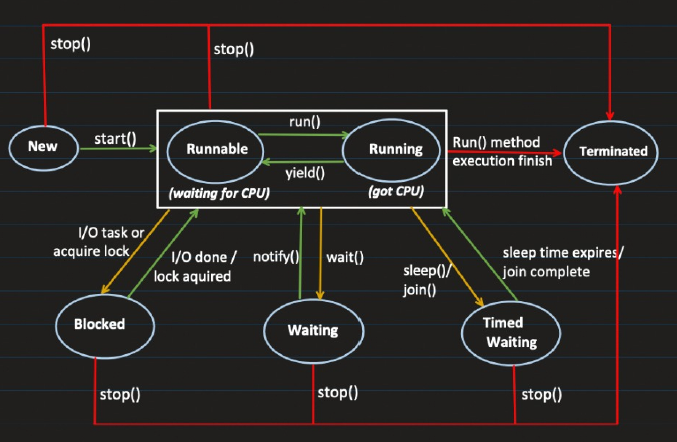
Simpler syntax: Since the class already extends Thread, you can directly call start() on the class itself, which might make the code look cleaner for simple scenarios.

Direct control: Extending Thread gives you access to more thread-specific methods, such as getId( ), getState( ), interrupt( ), and sleep( ), which could be useful in more advanced threading scenarios.

Disadvantages

Single inheritance: Since Java only supports single inheritance, extending Thread limits your class from extending any other class. This can be restrictive, especially if your class needs to inherit functionality from some other base class.

Thread Life Cycle



Thread Creation

* When a thread object is created using Thread thread = new Thread( ), the thread is in the New State.
* The thread is created but not yet started.

Start the Thread

* When thread.start( ) is called, the thread moves into the Runnable State.

Runnable vs. Running:

* Runnable: The thread is ready and waiting for CPU time.
* Running: The thread is actively executing on the CPU.
* The transition between runnable and running depends on CPU availability, handled by context switching.

Key Points

* Runnable State represents both runnable (waiting for CPU) and running (has CPU time) as a combined state.
* Running is not typically considered a separate state in thread lifecycle diagrams; it is part of the Runnable State.
* The thread keeps switching between runnable (waiting) and running (executing) based on CPU availability.

Completion of Task

* Once the thread completes its task, it finishes execution and enters the Terminated State.
* Once terminated, the thread cannot return to the runnable state.

Blocked State: The thread can enter a Blocked State while in the Runnable State due to:

* I/O operations (e.g., reading from a file or database).
* The thread is waiting for I/O to complete.
* Once the I/O task is done, the thread returns to the runnable state.
* If a thread tries to acquire a lock on a resource that is locked by another thread, it will go into the Blocked State.
* Once the thread acquires the lock, it returns to the runnable state.
* Once the thread is in the runnable state, the CPU will allocate time for it to start running.
* Monitor Locks: When a thread enters the blocked state, it releases all monitor locks.

Waiting State

* The thread can be explicitly put into the Waiting State by calling the wait( ) method.
* The thread will stay in the waiting state until the notify( ) or notifyAll( ) method is called.
* notify() signals the thread to return to the runnable state.
* If the notify() method is not called, the thread will remain indefinitely in the waiting state.
* During this state, the thread releases all monitor locks.

Blocked State: The Blocked State is typically caused by:

* I/O operations (e.g., waiting for a file or database).
* Resource locking (when a resource is locked by another thread).
* Once the operation (I/O or lock acquisition) is completed, the thread automatically returns to the runnable state.

Difference between Waiting and Blocked

* Blocked State occurs internally due to operations like I/O or acquiring a lock.
* Waiting State is a result of explicitly calling the wait( ) method.
* In the waiting state, the thread remains idle until notify( ) is called, while in the blocked state, the thread automatically resumes once the operation is complete.

Timed Waiting

* This state occurs when a thread is instructed to wait for a specific amount of time, like when calling the sleep() method.
* After the set time expires, the thread automatically goes back to the Runnable State.
* In this state, the thread does not release any monitor locks.

Monitor Locks

* Monitor Locks are mechanisms that control access to a resource by multiple threads.
* When a thread enters the Blocked or Waiting state, it releases all monitor locks.
* When a thread is in the Runnable or Running state, it holds any acquired monitor locks.
* In the Timed Waiting state, the thread does not release monitor locks.

Monitor Locks are of three types

* Instance method synchronization
* Static method synchronization
* Block synchronization

Monitor Lock

A monitor is a synchronization mechanism that allows threads to have mutual exclusive access to an object or a resource. When a thread enters a synchronized block or method, it acquires the monitor lock associated with the object, and it releases the lock when it exits the synchronized block or method.

* Each object in Java is associated with a monitor, which a thread can lock or unlock.
* Only one thread can hold the monitor lock at a time. Other threads trying to access the synchronized block or method will be blocked until the lock is released.

Key Points about Monitor Lock:

* Mutual exclusion: Only one thread can hold the lock on a given object or class at a time.
* Thread safety: It prevents race conditions by ensuring that only one thread can access the critical section of code at any point.
* Waiting and notifying: Threads can wait for a condition to become true by calling wait( ), and they can notify other threads waiting on the same monitor using notify( ) or notifyAll( ).

Instance Method Synchronization

When a thread enters an instance method synchronization, it acquires a lock on that specific object (instance), which leads to the following behaviours:

* Blocking on the Same Object:
* Other threads cannot enter any other instance synchronized method or block synchronized code on the same object.
* These threads must wait until the lock on that object is released.
* No Blocking on Static Synchronization:
* A thread holding the instance lock does not block other threads from entering static synchronized methods on the same object or class. This is because static methods are synchronized on the class-level lock, which is separate from the object instance lock.
* No Blocking on Different Objects:
* The instance lock is specific to the object. Therefore, other threads can still access instance synchronized methods or synchronized blocks on different instances of the same class (i.e., different objects), as each object has its own independent lock.



Static method synchronization

* If one thread enters a static synchronized method, it will block other threads from entering any static synchronized method in the same class. However, other threads can still access instance synchronized methods or synchronized blocks on any instance of the class, as these locks are independent of the static synchronization lock.



Block synchronization

When a thread enters a block synchronization on an object, it acquires a lock on that specific object (instance). This leads to the following behaviours:

Blocking on the Same Object

* If a thread enters a synchronized block on an object, it locks that object. As a result, other threads cannot enter any instance synchronized methods or other synchronized blocks on the same object.
* These threads must wait until the lock on the object is released.

No Blocking on Static Synchronization

* The instance-level lock acquired by entering the synchronized block does not block other threads from accessing static synchronized methods. This is because static methods are synchronized on the class-level lock, which is independent of the object instance lock.

No Blocking on Different Objects

* Since the lock is specific to the particular object, other threads can still access instance synchronized methods or synchronized blocks on different instances (i.e., different objects of the same class), as each object has its own separate lock.





