12.6: Thread States in Java

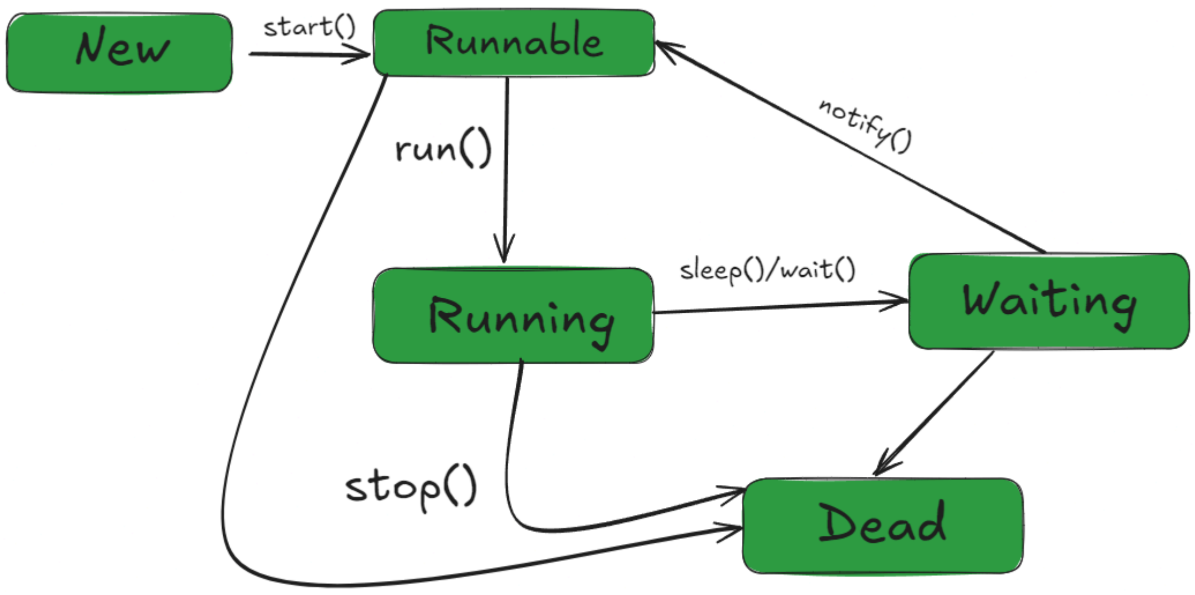
When working with threads in Java, understanding the various states a thread can go through is crucial. A thread, at any given point, can only exist in one of the predefined states. Let's look at the different stages of a thread's lifecycle and how it transitions between them.

Thread Lifecycle States

In Java, a thread exists in one of the following states during its lifecycle:

1. **New State**
2. **Runnable State**
3. **Running State**
4. **Waiting/Blocked State**
5. **Terminated/Dead State**

Each of these states plays an important role in the multithreading environment, allowing Java to manage thread execution and synchronization effectively.



---

1. New State

- **Definition**: When a thread is first created using the Thread class, it enters the **New** state.

- **Description**: In this state, the thread has been initialized but has **not yet started**. The thread is ready for execution but waits for a signal (invocation of the start() method) to begin its lifecycle.
- **Transition**: Once the start() method is called on the thread object, it moves from the **New** state to the **Runnable** state.

Example Code:

```
Thread thread = new Thread(); // Thread is in New State
```

---

2. Runnable State

- **Definition**: After a thread is started, it enters the **Runnable** state.
- **Description**: In this state, a thread is **ready to run** and waiting for CPU allocation. It could be actively running or just **waiting for CPU time** to be scheduled.
- **Thread Scheduler Role**: The thread scheduler decides which thread in the runnable pool will get CPU time.
- **Transition**: The thread remains in the **Runnable** state until it is allocated CPU time, at which point it moves to the **Running** state.

Example Code:

```
thread.start(); // Moves the thread to Runnable
State
```

---

3. Running State

- **Definition**: The **Running** state is where the thread is **actively executing** its code.
- **Description**: In this state, the CPU has assigned a time slot for the thread, and it's currently executing its task in the run() method.

- **Thread Scheduler**: The thread scheduler periodically allocates CPU time to threads. After a thread's time slice is over, it may go back to the **Runnable** state.

Transition:

  - Runnable → Running: The thread starts running when the CPU is allocated to it.
  - Running → Runnable: If the thread is **paused** (due to context switching), it returns to the **Runnable** state.

---

4. Waiting/Blocked State

- **Definition**: A thread enters the **Waiting** or **Blocked** state when it is **temporarily inactive** and waiting for some condition to be met.
- **Description**: This happens when a thread calls methods like wait(), sleep(), or is waiting for a resource (e.g., input/output operation) to become available.
  - **Blocked State**: The thread is waiting for a **resource**.
  - **Waiting State**: The thread is **waiting for another thread** to perform a specific action, such as a notification.
- **Transition Back to Runnable**: Once the required condition is met or the waiting period ends (e.g., using notify(), notifyAll(), or the specified sleep time ends), the thread returns to the **Runnable** state.

Example Methods:

  - sleep(): Pauses the thread for a specified duration.
  - wait(): Waits indefinitely until another thread calls notify() or notifyAll().

Example Code:

```
try {
    Thread.sleep(1000); // Pauses the thread for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();}
```

5. Terminated/Dead State

- **Definition**: A thread enters the **Terminated** or **Dead** state when it has **completed its execution**.
- **Description**: This happens when the run() method completes execution or when the thread is terminated using certain deprecated methods (e.g., stop()).
- **Deprecated Method**: Calling the stop() method can also cause a thread to terminate, but this is **not recommended** as it can cause the program to behave unexpectedly.

Example Code:

```
public void run() {
    System.out.println("Thread is running...");
    // After this method completes, the thread moves to the
Dead State
}
```

Conclusion

Understanding the different states of a thread and their transitions is crucial for effective multithreading in Java. Each state—**New**, **Runnable**, **Running**, **Waiting/Blocked**, and **Terminated**—serves a specific purpose in managing thread execution. A thread moves through these states based on method calls like start(), sleep(), and wait(), as well as through conditions such as CPU allocation and resource availability.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

By mastering these thread states and transitions, developers can better control thread behavior, optimize program performance, and avoid potential issues such as deadlock or thread starvation. A clear grasp of the thread lifecycle helps in writing robust and efficient multithreaded programs.