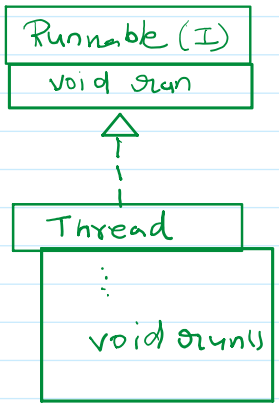
The Java Threading API

19 January 2026 18:31



neutral behavior

public void run() { }

run

```
public void run()
```

This method is run by the thread when it executes. Subclasses of Thread may override this method.

This method is not intended to be invoked directly. If this thread is a platform thread created with a Runnable task then invoking this method will invoke the task's run method. If this thread is a virtual thread then invoking this method directly does nothing.

Specified by:

run in interface Runnable

Implementation Requirements:

The default implementation executes the Runnable task that the Thread was created with. If the thread was created without a task then this method does nothing.

Object created, when Thread class is not extended.

```
jshell> chapter1.OurClass oc = new chapter1.OurClass();
oc ==> chapter1.OurClass@421faab1
```

Normal object

```
package chapter1;
public class OurClass{
    @Override
    public void run(){
        int i;
        for(i = 0; i < 100; i++){
            System.out.println("Hello");
        }
    }
}
```

javr.lang.Thread.

To start the thread we need to call start() method.

This method will be executed by newly created thread.
Developers should override this method. with the code they want new thread

```
package chapter1;
public class OurClass extends Thread{
    @Override
    public void run(){
        int i;
        for(i = 0; i < 100; i++){
            System.out.println("Hello");
        }
```

```
        for(i = 0; i < 100; i++){
            System.out.println("Hello");
        }
    }
}
```

Developers should override this method with the code they want new thread to run.

```
jshell> chapter1.OurClass oc = new chapter1.OurClass();
oc ==> Thread[#39,Thread-0,5,main]
```

Thread ID | Thread name | Priority | Which thread main.

Threading using the Thread class

We consider threads as separate task that execute in parallel. These tasks are simply code executed by the thread, and this code is actually part of our program. The code may download an image from the server or may play audio file on the speakers or any other task. Because it is a code. It can be executed by our original thread. To introduce the parallelism we desire, we must create a new thread and arrange for the new thread to. execute the appropriate code.

```
package chapter1;
public class TestFirstThread {
    public static void main(String[] args) {
        OurClass oc = new OurClass();
        oc.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("Main");
        }
    }
}
```

execute.

Main

born & ready

oc

oc.start()

↖ it will call the run() method, when processor allots the time

calls the run() method defined in this thread class.

run() vs main()

In essence, the run() method may be thought of as main() method of the newly formed thread. A new thread begins execution with a run() method in the same way a program begins execution with the main() method.
While the main() method receive its argument from argv parameter (which is typically set from the command line) The newly created thread must receive its arguments programmatically from the originating threads. Hence, parameters can be passed in via the constructor, the static instance variable, or any other technique designed by the developer.

sleep() methods:

public static void sleep(long milliseconds) throws InterruptedException: It puts the current executing thread to sleep for the specified number of milliseconds. This method is static and may be accessed through the thread class name.

public static void sleep(long milliseconds, int nanoseconds) throws InterruptedException: It puts the currently executing thread to sleep for a specified number of milliseconds and nanoseconds.

```java
package chapter1;
public class FirstThread  extends Thread{
    @Override
    public void run() {
        final int max = 10;
        for (int i = 0; i < max; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
package chapter1;
public class FirstThreadTest {
    public static void main(String[] args) throws InterruptedException {
        FirstThread ft = new FirstThread();
        ft.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("Main: " + i);
            Thread.sleep(1000);
        }
    }
}
```

Output:

```
0
Main: 0
Main: 1
1
Main: 2
2
Main: 3
Main: 4
3
Main: 5
4
Main: 6
Main: 7
5
Main: 8
6
Main: 9
7
8
9
```

Java supports single inheritance through classes.

public class classname extends singleclass, implements multiple interfaces, ...
{
}

Thread using the Runnable interface

As simple as it to create another thread of control. There is one problem

with the technique we have outlined so far. It's caused by the fact that Java classes can inherit their behavior only from a single class, which means that inheritance itself can be considered a scarce resource and therefore expensive to the developer.
In our example, we are threading a simple loop so that it is not much of a concern, however. If we have a complete class structure that already has a detailed inheritance tree and want it to run its own thread, we cannot simply make this class structure inherit from the Thread class as we did before. One solution would be to create a new class that inherits from the thread and contains references to the instance of the class we need. This level of indirection is an annoyance.
The Java language deals with this lack of multiple inheritance by using the mechanism known as interfaces. This mechanism is supported by the Thread class and simply means that instead of inheriting from the Thread class, we can implement the Runnable interface(java.lang.Runnable) which is defined as follows:

An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method.

@FunctionalInterface
public interface Runnable
Represents an operation that does not return a result.
This is a functional interface whose functional method is run().

abstract method

Since:
1.0

The Runnable interface contains only one method, the run () method. The Thread class actually implements the Runnable interface. Hence when you inherit from Thread class, your sub class also implements the Runnable interface. However, in this case we want to implement the Runnable interface without actually inheriting from the Thread class. This is achieved by simply substituting the phrase "**implements Runnable**" for the phrase "**extends Threads**" No other changes are necessary in our thread creation process.

```
package chapter1;
public class FirstThread  implements Runnable{
    @Override
    public void run() {
        final int max = 10;
        for (int i = 0; i < max; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
package chapter1;
public class FirstThreadTest {
    public static void main(String[] args) throws InterruptedException {
        FirstThread ft = new FirstThread();
        Thread t1 = new Thread(ft);
        t1.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("Main: " + i);
            Thread.sleep(1000);
        }
    }
}
```

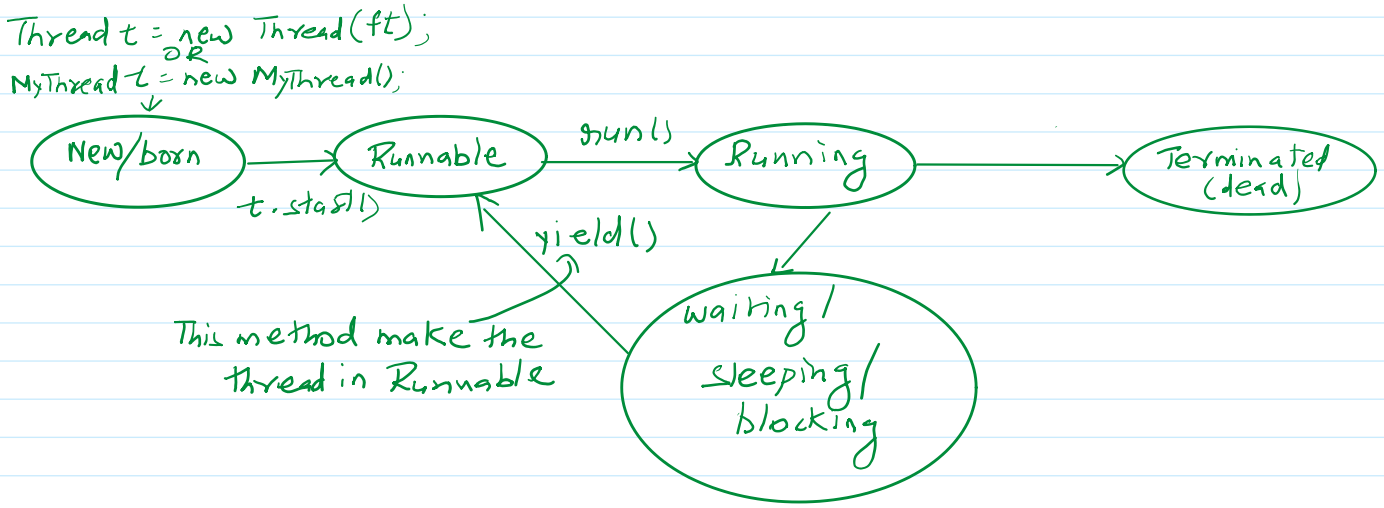
Object of user defined thread class.

Object of Thread class is created, and we have passed the object of user-defined thread class. To get access of start() method of Thread class, which calls the run() method of user-defined class.

Output:

```
0
Main: 0
Main: 1
1
Main: 2
2
Main: 3
Main: 4
3
Main: 5
4
Main: 6
Main: 7
5
Main: 8
6
Main: 9
7
8
9
```

Life Cycle of a Thread in Java



A thread in Java goes through several states from creation to completion. These states are defined in the Thread.State enum.

public static enum **Thread.State**
extends Enum<Thread.State>

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

Since:

1.5

1. NEW state: A thread is in the new state when it is created but not started.
   Example:
      Thread t = new Thread(); //new state
   At this point:
    - Memory allocated
    - Thread object exists
    - But the thread is not scheduled to run.

2. RUNNABLE state: When we call start() method on a thread, it enters the runnable state.
   t.start(); //runnable state
   This means:
    - The thread is ready to run.
    - It is placed in the thread scheduler queue.
    - It may or may not execute immediately.

3. RUNNING state: A thread enters the running state when the thread scheduler picks it to run. Inside the run() method.
   The CPU decides when a thread actually runs your thread(code).

4. WAITING/BLOCKED/TIMED_WATITING state: These states occur, when a thread is temporarily inactive but not finished.
   a. WAITING state: Thread waits indefinitely for another thread to perform an action. Example:
      i. Obj.wait(); // Waiting thread can be awake by (runnable) notify() or notifyAll() methods. This method belongs to Object class, when your thread object is synchronized then use wait() method.

      ii. Obj.join();// waiting, this method belongs to Thread class. Ensures sequential execution by causing the current threat to pause its execution

until the thread on which join() is called complete its execution. Can be called with or without a synchronized context. It does not involve acquiring or releasing object locks.

b. TIMED_WAITING: Thread waits for specific time.
   Examples:
   ```
   Thread.sleep(1000); //Timed waiting
   Obj.wait(5000);    //Timed waiting
   t.join(2000);      //Timed waiting
   ```

c. BLOCKED State: Thread is blocked because it wants a monitor lock that another thread holds occurs in synchronized code.
   Example:
   ```
   synchronized(obj){
      //Code that acquired lock
   }
   ```
   If another thread is using obj, current thread becomes, BLOCKED.

5. TERMINATED State: A thread enters the terminated (dead) state when:
   i. run() method finishes normally
   ii. Thread is stopped using stop () (unsafe, deprecated).
   iii. And uncought exception occurs inside the run()

```
package ThreadState;
public class MyNewThread extends Thread{
    @Override
    public void run() {
        System.out.println("Thread is running...");
        System.out.println("Thread goes to Timed Waiting(sleep)...");
        try {
            Thread.sleep(3000);
            System.out.println("Thread is running again...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread is terminated");
    }
}
```

```
package ThreadState;
public class ThreadLifeCycle {
    public static void main(String[] args) {
        MyNewThread t = new MyNewThread();
        System.out.println("Thread state after creation: " + t.getState());
        t.start();
        try {
            Thread.sleep(1000);
            System.out.println("Thread state while running: " + t.getState());
            t.join(); //Main thread waits
            System.out.println("Thread state after join(): " + t.getState());
//TERMINATED
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Output:

```
Thread state after creation: NEW
Thread is running...
Thread goes to Timed Waiting(sleep)...
Thread state while running: TIMED_WAITING
Thread is running again...
Thread is terminated
Thread state after join(): TERMINATED
```