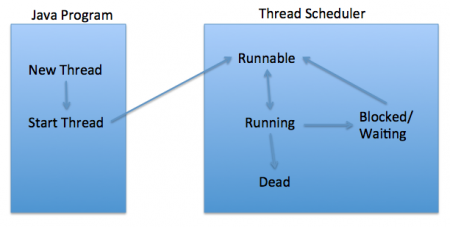
### What are different states in lifecycle of Thread?

**Ans.**

**Thread Life Cycle in Java**

Below diagram shows different states of thread life cycle in java. We can create a thread in java and start it but how the thread states change from Runnable to Running to Blocked depends on the OS implementation of thread scheduler and java doesn’t have full control on that.

[](https://cdn.journaldev.com/wp-content/uploads/2012/12/Thread-Lifecycle-States.png)

**New**

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.

**Runnable**

When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

**Running**

When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

**Blocked/Waiting**

A thread can be waiting for other thread to finish using [thread join](https://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available. For example [producer consumer problem](https://www.journaldev.com/1034/java-blockingqueue-example) or [waiter notifier implementation](https://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example) or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.

**Dead**

Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

Above are the different **states of thread**. It’s good to know them and how thread changes it’s state. That’s all for thread life cycle in java.

### How can we create a Thread in Java?

### Ans.

### Java Thread Example – implementing Runnable interface

To make a class runnable, we can implement java.lang.Runnable interface and provide implementation in public void run() method. To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call start() method to execute the run() method in a separate thread.

package com.journaldev.threads;

public class HeavyWorkRunnable implements Runnable {

@Override

public void run() {

System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

### Java Thread Example – extending Thread class

We can extend **java.lang.Thread** class to create our own java thread class and override run() method. Then we can create it’s object and call start() method to execute our custom java thread class run method.

Here is a simple java thread example showing how to extend Thread class.

package com.journaldev.threads;

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

System.out.println("MyThread - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("MyThread - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

Here is a test program showing how to create a java thread and execute it.

package com.journaldev.threads;

public class ThreadRunExample {

public static void main(String[] args){

Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");

Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");

System.out.println("Starting Runnable threads");

t1.start();

t2.start();

System.out.println("Runnable Threads has been started");

Thread t3 = new MyThread("t3");

Thread t4 = new MyThread("t4");

System.out.println("Starting MyThreads");

t3.start();

t4.start();

System.out.println("MyThreads has been started");

}

}

Output of the above java thread example program is:

Starting Runnable threads

Runnable Threads has been started

Doing heavy processing - START t1

Doing heavy processing - START t2

Starting MyThreads

MyThread - START Thread-0

MyThreads has been started

MyThread - START Thread-1

Doing heavy processing - END t2

MyThread - END Thread-1

MyThread - END Thread-0

Doing heavy processing - END t1

Once we start any thread, it’s execution depends on the OS implementation of time slicing and we can’t control their execution. However we can set threads priority but even then it doesn’t guarantee that higher priority thread will be executed first.

Run the above program multiple times and you will see that there is no pattern of threads start and end.

### Runnable vs Thread

If your class provides more functionality rather than just running as Thread, you should implement Runnable interface to provide a way to run it as Thread. If your class only goal is to run as Thread, you can extend Thread class.

Implementing Runnable is preferred because java supports implementing multiple interfaces. If you extend Thread class, you can’t extend any other classes.

**Tip**: As you have noticed that thread doesn’t return any value but what if we want our thread to do some processing and then return the result to our client program, check our [**Java Callable Future**](https://www.journaldev.com/1090/java-callable-future-example).

**Update**: From Java 8 onwards, Runnable is a functional interface and we can use lambda expressions to provide it’s implementation rather than using anonymous class. For more details, check out [Java 8 Functional Interfaces](https://www.journaldev.com/2763/java-8-functional-interfaces).