**Defining and Starting a Thread**

An application that creates an instance of *Thread* must provide the code that will run in that thread. Java provides two ways to create a thread programmatically.

1. Implementing the **Runnable** interface.
2. Extending the **Thread** class.

**1. Implementing the Runnable interface**

The **Runnable** interface defines a single method, run(), meant to contain the code executed in the thread. The **Runnable** object is passed to the Thread constructor.

Let's demonstrate how to use **Runnable** interface with an example.

First, create a *Task or WorkerThread* using *Runnable* interface.

**WorkerThread.java**

public class WorkerThread implements Runnable {

private String data;

public WorkerThread(final String anyData) {

this.data = anyData;

}

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println("[" + Thread.currentThread().getName() + "] [data=" +

this.data + "] Message " + i);

try {

Thread.sleep(200);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

}

Second, let's see how to use *WorkerThread*classin the main thread. In this example, the *main()* method is the main thread which will create another using Above *WorkerThread* class.

public class InstantiateUsingRunnable {

public static void main(final String[] args) {

System.out.println("Thread main started");

final Thread thread1 = new Thread(new WorkerThread("Process data through Runnable interface"));

thread1.start();

thread1.setName("Demo Thread");

System.out.println("Thread main finished");

}

}

Output:

Thread main started

Thread main finished

[Demo Thread] [data=Process data through Runnable interface] Message 0

[Demo Thread] [data=Process data through Runnable interface] Message 1

[Demo Thread] [data=Process data through Runnable interface] Message 2

[Demo Thread] [data=Process data through Runnable interface] Message 3

[Demo Thread] [data=Process data through Runnable interface] Message 4

**Runnable Interface Example using Anonymous Class**

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time.

public class RunnableExampleUsingAnonymousClass {

public static void main(final String[] args) {

System.out.println(" main thread started : " + Thread.currentThread().getName());

System.out.println("Creating Runnable...");

final Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("Inside : " + Thread.currentThread().getName());

}

};

System.out.println("Creating Thread...");

final Thread thread = new Thread(runnable);

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

thread.start();

System.out.println(" main thread ended : " + Thread.currentThread().getName());

}

}

Output:

main thread started : main

Creating Runnable...

Creating Thread...

Starting Thread...

main thread ended : main

Inside : Thread-0

**Runnable Interface Example Using Lambda Expression**

The above example can be made even shorter by using **Java 8 Lambda Expressions** -

public class RunnableExampleUsingLambda {

public static void main(final String[] args) {

System.out.println(" main thread started : " + Thread.currentThread().getName());

System.out.println("Creating Runnable...");

final Runnable runnable = () -> System.out.println("Inside : " + Thread.currentThread().getName());

System.out.println("Creating Thread...");

final Thread thread = new Thread(runnable);

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

thread.start();

System.out.println(" main thread ended : " + Thread.currentThread().getName());

}

}

Output:

Creating Runnable...

Creating Thread...

Starting Thread...

main thread ended : main

Inside : Thread-0

**2. Extending the Thread class**

You can create a new thread simply by extending your class from *java.lang.Thread* and overriding it’s *run()* method.

The *run()* method contains the code that is executed inside the new thread. Once a thread is created, you can start it by calling the *start()* method.

Let's demonstrate how to create and start a Thread by extending *java.lang.Thread* class.

In this example the *main()* method is the main thread, we will create and start the *WorkerThread*in the main thread.

class WorkerThread extends Thread {

private String anyData;

public WorkerThread(final String anyData) {

this.anyData = anyData;

}

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println("[" + Thread.currentThread().getName() + "] "

+ "[data=" + this.anyData + "] Message " + i);

try {

Thread.sleep(200);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

}

/\*\*

\* Instantiate a thread by using Thread class.

\* @author Ramesh fadatare

\*\*/

public class ThreadExample {

public static void main(final String[] args) {

System.out.println("Thread main started");

final Thread thread = new WorkerThread("Process data using WorkerThread");

thread.start();

thread.setName("WorkerThread");

System.out.println("Thread main finished");

}

}

Output:

Thread main started

Thread main finished

[WorkerThread] [data=Process data using WorkerThread] Message 0

[WorkerThread] [data=Process data using WorkerThread] Message 1

[WorkerThread] [data=Process data using WorkerThread] Message 2

[WorkerThread] [data=Process data using WorkerThread] Message 3

[WorkerThread] [data=Process data using WorkerThread] Message 4

**Creating Multiple Threads**

In this example, we have created 2 threads, you can create any number of threads that required to satisfy your requirement.

public class ThreadSleepExample {

public static void main(final String[] args) {

System.out.println("Thread main started");

final Thread thread1 = new Thread(new WorkerThread());

thread1.setName("WorkerThread 1");

final Thread thread2 = new Thread(new WorkerThread());

thread1.setName("WorkerThread 2");

thread1.start();

thread2.start();

System.out.println("Thread main ended");

}

}

class WorkerThread implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

System.out.println("[" + Thread.currentThread().getName() + "] Message " + i);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

}

Output:

[WorkerThread 2] Message 0

[Thread-1] Message 0

[Thread-1] Message 1

[WorkerThread 2] Message 1

[WorkerThread 2] Message 2

[Thread-1] Message 2

[WorkerThread 2] Message 3

[Thread-1] Message 3

[Thread-1] Message 4

[WorkerThread 2] Message 4

**Runnable or Thread, Which one to use?**

We have learned how to create and start a thread via implementing Runnable interface and extending java.lang.Thread class. Now it's time to decide which one should we use?

In general, You should always use a *Runnable* object to create a thread. This method is more flexible. It allows your class to extend from any other class. Also, you can use anonymous class syntax and Java 8’s lambda expression with Runnable to make your code more concise.

The first second, where we created a thread by extending from *Thread* class is very limited because once you extend your class from *Thread*, you cannot extend from any other class since Java doesn’t allow multiple inheritance.

Also, If you follow good design practice, Inheritance is meant for extending the functionality of the parent class, but when you create a thread, you don’t extend the functionality of *Thread*class, you merely provide the implementation of*run()* method

**Thread.sleep() Method Overview**

*Thread.sleep* causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

Thread class provides two overloaded versions of sleep method:

1. *static void sleep(long millis)* - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
2. *static void sleep(long millis, int nanos)* - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.

**Thread.sleep() Method Example**

In this example, we have created and started two threads *thread1*and *thread2*. note that we have used both overloaded versions of *sleep()* methods in this example.

Thread.sleep(1000);

Thread.sleep(1000, 500);

/\*\*

\* thread sleep method examples

\* @author Ramesh fadatare

\*

\*/

public class ThreadSleepExample {

public static void main(final String[] args) {

System.out.println("Thread main started");

final Thread thread1 = new Thread(new WorkerThread());

thread1.setName("WorkerThread 1");

final Thread thread2 = new Thread(new WorkerThread());

thread1.setName("WorkerThread 2");

thread1.start();

thread2.start();

System.out.println("Thread main ended");

}

}

class WorkerThread implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(1000);

Thread.sleep(1000, 500);

System.out.println("[" + Thread.currentThread().getName() + "] Message " + i);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

}

Output:

Thread main started

Thread main ended

[WorkerThread 2] Message 0

[Thread-1] Message 0

[WorkerThread 2] Message 1

[Thread-1] Message 1

[WorkerThread 2] Message 2

[Thread-1] Message 2

[WorkerThread 2] Message 3

[Thread-1] Message 3

[WorkerThread 2] Message 4

[Thread-1] Message 4

Note that *sleep()* method throws *InterruptedException*exception, when another thread interrupts the current thread while sleep is active.

## Thread join() Method Example

We will first create a *Task*which will calculate the sum of 1-5 numbers. In the main thread lets create 4 tasks:

final Task task1 = new Task(500l);

final Task task2 = new Task(1000l);

final Task task3 = new Task(2000l);

final Task task4 = new Task(50l);

Now, let's create 4 threads to run above 4 tasks:

final Thread thread1 = new Thread(task1);

final Thread thread2 = new Thread(task2);

final Thread thread3 = new Thread(task3);

final Thread thread4 = new Thread(task4);

Assign name to each thread and start all the 4 threads:

thread1.setName("thread-1");

thread2.setName("thread-2");

thread3.setName("thread-3");

thread4.setName("thread-4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

In this example, when the target thread finishes the sum, the caller thread (main) wakes up and calls the task.getSum() method which will certainly contain the total sum as the target thread has already finished its job.

The *task4*has a small sleep time and therefore it finishes the sum before the others. Hence, the main thread calls the *thread4.join()* but immediately returns to its execution as the *thread4*is finished.

/\*\*

\* This class demonstrate the how join method works with an example.

\* @author Ramesh Fadatare

\*

\*/

public class ThreadJoinExample {

public static void main(final String[] args) throws InterruptedException {

System.out.println("Thread main started");

final Task task1 = new Task(500l);

final Task task2 = new Task(1000l);

final Task task3 = new Task(2000l);

final Task task4 = new Task(50l);

final Thread thread1 = new Thread(task1);

final Thread thread2 = new Thread(task2);

final Thread thread3 = new Thread(task3);

final Thread thread4 = new Thread(task4);

thread1.setName("thread-1");

thread2.setName("thread-2");

thread3.setName("thread-3");

thread4.setName("thread-4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread1.getName());

thread1.join();

System.out.println(thread1.getName() + " finished! Result: " + task1.getSum());

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread2.getName());

thread2.join();

System.out.println(thread2.getName() + " finished! Result: " + task2.getSum());

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread3.getName());

thread3.join();

System.out.println(thread3.getName() + " finished! Result: " + task3.getSum());

// As thread-4 already finished (smaller sleep time), the join call only immediately

// returns the control to the caller thread

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread4.getName());

thread4.join();

System.out.println(thread4.getName() + " finished! Result: " + task4.getSum());

System.out.println("Thread main finished");

}

}

class Task implements Runnable {

private long sleep;

private int sum;

public Task(final long sleep) {

this.sleep = sleep;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("[" + Thread.currentThread().getName() + "] Adding " + i);

sum += i;

try {

Thread.sleep(sleep);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

public int getSum() {

return this.sum;

}

}

Output:

Thread main started

[thread-1] Adding 1

[thread-2] Adding 1

[thread-3] Adding 1

[main] waiting for thread-1

[thread-4] Adding 1

[thread-4] Adding 2

[thread-4] Adding 3

[thread-4] Adding 4

[thread-4] Adding 5

[thread-1] Adding 2

[thread-1] Adding 3

[thread-2] Adding 2

[thread-1] Adding 4

[thread-1] Adding 5

[thread-3] Adding 2

[thread-2] Adding 3

thread-1 finished! Result: 15

[main] waiting for thread-2

[thread-2] Adding 4

[thread-2] Adding 5

[thread-3] Adding 3

thread-2 finished! Result: 15

[main] waiting for thread-3

[thread-3] Adding 4

[thread-3] Adding 5

thread-3 finished! Result: 15

[main] waiting for thread-4

thread-4 finished! Result: 15

Thread main finished

Note that the output, *main()* thread finished it's executions last. Try to understand the example via looking into an output.

Note that *join()* method throws an *InterruptedException*, if any thread has interrupted the current thread.

## Thread join() Method Example

We will first create a *Task*which will calculate the sum of 1-5 numbers. In the main thread lets create 4 tasks:

final Task task1 = new Task(500l);

final Task task2 = new Task(1000l);

final Task task3 = new Task(2000l);

final Task task4 = new Task(50l);

Now, let's create 4 threads to run above 4 tasks:

final Thread thread1 = new Thread(task1);

final Thread thread2 = new Thread(task2);

final Thread thread3 = new Thread(task3);

final Thread thread4 = new Thread(task4);

Assign name to each thread and start all the 4 threads:

thread1.setName("thread-1");

thread2.setName("thread-2");

thread3.setName("thread-3");

thread4.setName("thread-4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

In this example, when the target thread finishes the sum, the caller thread (main) wakes up and calls the task.getSum() method which will certainly contain the total sum as the target thread has already finished its job.

The *task4*has a small sleep time and therefore it finishes the sum before the others. Hence, the main thread calls the *thread4.join()* but immediately returns to its execution as the *thread4*is finished.

/\*\*

\* This class demonstrate the how join method works with an example.

\* @author Ramesh Fadatare

\*

\*/

public class ThreadJoinExample {

public static void main(final String[] args) throws InterruptedException {

System.out.println("Thread main started");

final Task task1 = new Task(500l);

final Task task2 = new Task(1000l);

final Task task3 = new Task(2000l);

final Task task4 = new Task(50l);

final Thread thread1 = new Thread(task1);

final Thread thread2 = new Thread(task2);

final Thread thread3 = new Thread(task3);

final Thread thread4 = new Thread(task4);

thread1.setName("thread-1");

thread2.setName("thread-2");

thread3.setName("thread-3");

thread4.setName("thread-4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread1.getName());

thread1.join();

System.out.println(thread1.getName() + " finished! Result: " + task1.getSum());

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread2.getName());

thread2.join();

System.out.println(thread2.getName() + " finished! Result: " + task2.getSum());

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread3.getName());

thread3.join();

System.out.println(thread3.getName() + " finished! Result: " + task3.getSum());

// As thread-4 already finished (smaller sleep time), the join call only immediately

// returns the control to the caller thread

System.out.println("[" + Thread.currentThread().getName() + "] waiting for " + thread4.getName());

thread4.join();

System.out.println(thread4.getName() + " finished! Result: " + task4.getSum());

System.out.println("Thread main finished");

}

}

class Task implements Runnable {

private long sleep;

private int sum;

public Task(final long sleep) {

this.sleep = sleep;

}

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("[" + Thread.currentThread().getName() + "] Adding " + i);

sum += i;

try {

Thread.sleep(sleep);

} catch (final InterruptedException e) {

e.printStackTrace();

}

}

}

public int getSum() {

return this.sum;

}

}

Output:

Thread main started

[thread-1] Adding 1

[thread-2] Adding 1

[thread-3] Adding 1

[main] waiting for thread-1

[thread-4] Adding 1

[thread-4] Adding 2

[thread-4] Adding 3

[thread-4] Adding 4

[thread-4] Adding 5

[thread-1] Adding 2

[thread-1] Adding 3

[thread-2] Adding 2

[thread-1] Adding 4

[thread-1] Adding 5

[thread-3] Adding 2

[thread-2] Adding 3

thread-1 finished! Result: 15

[main] waiting for thread-2

[thread-2] Adding 4

[thread-2] Adding 5

[thread-3] Adding 3

thread-2 finished! Result: 15

[main] waiting for thread-3

[thread-3] Adding 4

[thread-3] Adding 5

thread-3 finished! Result: 15

[main] waiting for thread-4

thread-4 finished! Result: 15

Thread main finished

Note that the output, *main()* thread finished it's executions last. Try to understand the example via looking into an output.

Note that *join()* method throws an *InterruptedException*, if any thread has interrupted the current thread.