Thread:

We can define a thread in the following two ways:

By extending Thread class

By implementing Runnable interface

In the first approach, Our class always extends Thread class. There is no chance of extending any other class. Hence we are missing Inheritance benefits. In the second approach, while implementing Runnable interface we can extends any other class. Hence we are able to use the benefits of Inheritance.

Because of the above reasons, implementing Runnable interface approach is recommended than extending Thread class.

**The significant differences between extending Thread class and implementing Runnable interface:**

* When we extend Thread class, we can’t extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.
* When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.
* Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

**An example of Runnable Interface Implementing:**

**class ImplementsRunnable implements Runnable**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ImplementsRunnable : Counter : " + counter);**

**}**

**}**

**class ExtendsThread extends Thread**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ExtendsThread : Counter : " + counter);**

**}**

**}**

**public class ThreadVsRunnable**

**{**

**public static void main(String args[]) throws Exception**

**{**

**//Multiple threads share the same object.**

**ImplementsRunnable rc = new ImplementsRunnable();**

**Thread t1 = new Thread(rc);**

**t1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t2 = new Thread(rc);**

**t2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t3 = new Thread(rc);**

**t3.start();**

**//Creating new instance for every thread access.**

**ExtendsThread tc1 = new ExtendsThread();**

**tc1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**ExtendsThread tc2 = new ExtendsThread();**

**tc2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**ExtendsThread tc3 = new ExtendsThread();**

**tc3.start();**

**}**

**}**

**The output is something like this:**

ImplementsRunnable : Counter : 1

ImplementsRunnable : Counter : 2

ImplementsRunnable : Counter : 3

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

It proves differences given above. I make a slight modification in code given below:

**class ImplementsRunnable implements Runnable**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ImplementsRunnable : Counter : " + counter);**

**}**

**}**

**public class Runnable**

**{**

**public static void main(String args[]) throws Exception**

**{**

**//Multiple threads share the same object.**

**ImplementsRunnable rc = new ImplementsRunnable();**

**Thread t1 = new Thread(rc);**

**t1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t2 = new Thread(rc);**

**t2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t3 = new Thread(rc);**

**t3.start();**

**}**

**}**

Now, this is a sample of Runnable interface using. Now, ImplementsRunnable class which implements Runnable, can be shared among various threads.

Now, how to pass that Runnable object in thread? Pass it in constructor during thread creation.

**More On Thread:**

An user thread is created as an instance of java.lang.Thread.

The task which is allocated to a thread always starts from a **run() method.**

**Some important point:**the thread class itself implements Runnable. After all, it has a run method that we are overriding. That means you could pass a thread to another thread’s constructor:

**This is a bit silly, but it’s legal. In this case, you really just need a Runnable(), and creating a whole other thread is overkill**

**Starting A Thread:**

You have created a thread object and it knows its target. Now, it’s time to get the whole new thread thing happening-to launch a new call stack. It’s so simple, it hardly deserves its own subheading:

t.start()

**What happen when start is called?**

* A new thread of execution starts (with a new call stack)
* The thread moves from the new state to the runnable state
* When the thread gets a chance to execute, it’s target run method will run.

**Be sure, you remember the following, start a thread, not a runnable. You call start() on a thread instance, not a runnable instance.**

When start method is called, it changes the thread state from new to runnable.

Now, the run function will be implicitly called. (because, it’s like a thread is submitted to a queue of runnable threads. Whenever, this threads turn comes, it will implicitly execute run method. **Note that, implicitly.**There’s nothing special about the run() method as far java is concerned. Like main(), it just happens to be the name and signature of the method that the new thread knows to invoke(). So if you see code that calls the run() method on a runnable (or even a thread instance) that’s perfectly legal. But that does not mean the run() method will run in a separate thread. Calling a run() method just means you are invoking a method from whatever thread you are currently executing, and run() method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:

**Thread t=new Thread();**

**t.run();**

It’s perfectly legal. But does not start a new thread.

For each thread, there are two special functions. (Now, we are not talking about setDaemon(true). We are talking about setName() and getName() it could be good to identify different threads.

(suppose, separate threads share a common object of a class which implements Runnable. Now, that class has common specific tasks for each thread. However, based on current thread name, it might also performs some specific task.

**Thread.currrentThread().getName() could help.**

**In additiuon to using setName and getName to identify threads, you might see getId(). the getId() method returns a positive, unique, long number and that number will be that thread’s only ID number for the entire thread life.**

**Thread Priorities And Yield:**

To understand yield(), you must understand the concept of thread priorities. Threads always run with some priority. The thread scheduler in most JVMs use preemptive, priority based scheduling. (starving and aging)

Now, you can set a thread’s priority, **t.setPriority()** function can do that. Where, t is a thread object.

**Don’t rely on thread priorities when designing your multithreaded application, because thread scheduling priority behaviour is not guaranteed, use thread priorities as a way to improve the efficiency of your program. But be sure, your program does not depend on that behaviour for correctness.**

**The yield method prevents starving in a way. Yield is supposed to do is make the the currently running thread head back to runnable to allow other threads of the same priority to get their turn.**

Note, a yield() wont ever cause a thread to go from running to waiting/ sleeping/ blocked. It makes a thread to go to runnable from running, atmost.

**Join:**

We all know how join method works.

**Thread States:**

New, Runnable, Running, Waiting/Blocked/sleeping, Dead

**Preventing Thread Execution:**sleeping

Waiting

Blocked

The sleep() method is a static method of class thread. You use it your class thread to slow a thread down by forcing it go into a sleep mode.

Waking up from sleep puts a thread from sleep mode to runnable mode. It does not necessarily starts its execution immediately.

**Now, When do you use multithreading and what are the things do you need to find out before thread designing:**

**First job** should be Identifying parallel tasks:

**Second Job** should be identifying how many threads can you run?

Now, a quad core CPU (with four processors per unit) might be able to to execute two threads per core for a total of eight concurrently executing threads. Now, you can start with 10000 threads, but not all of them will be running at the same time. The underlying operating system’s task scheduler rotates the threads so that they each get a slice of time on a processor. So, 10000 threads all competing for a turn on processor would not result a very responsive system. Threads would either have to wait for long for a turn or get such small turns that performance would suffer.

In addition, each thread consumes system resources. It takes processor cycles to perform a context switch (saving the current state of the thread and resuming other thread) and each thread consumes system memory for it’s stack space. Stack space is used for temporary storage and to keep keep track of where a thread returns to after completing a method call. Depending on a thread’s behaviour, it might be possible to lower the cost (in RAM) of creating thread by reducing a thread’s stack size.

**To reduce a thread’s stack size, the oracle JVM supports using the nonstandard Xss1024k option to the java command. Note that decreasing value too far can result in some threads throwing exceptions when performing certain tasks, such as making a large number of recursive method calls.**

**Third Job** will be identifying CPU intensive tasks vs I/O intensive tasks.

**If your tasks will be performing I/O operations, you should be concerned about how increased load (users) might affect scalability. If your tasks perform blocking I/O, then you might need to utilize a thread par task model. If you don’t then all all of your tasks may be tied up to I/O operations with no threads remaining to support additional users. Another option would be investigating whether you can use non blocking I/O instead of blocking I/o.**

**Decoupling Tasks From Threads:**

The best design would one that utilized as many system resources as possible without overdoing it. If 16 threads are all you need to fully utilize your CPU, why would you start more than that? **In traditional system, you start more threads than your system can concurrently run and hope that only a small number of them are in running state.** If we want to adjust the number of threads that are started, we need to decouple the tasks that are to be performed (our Runnable instances) from our thread creation and starting. This is where a java.util.Executor would help. The basic usage looks like:

Runnable r=new MyRunnableTask();

Executor ex=//details to follow

ex.execute( r);

A java.util.concurrent.Executor is used to execute the run method in a Runnable instance much like a thread, unlike a more traditional new Thread( r).start(), an executor can be designed to use any number of threading approach, including:

* Not starting any threads at all (task the run in the calling thread)
* Starting a new thread for each task
* Queueing tasks and processing them with only enough threads to keep the CPU utilized.

You can easily create your own implementations of an Executor with custom behaviors.

This next example wont start any new threads, instead it executes the Runnable using the thread that invoked the executor.

**import java.util.concurrent.Executor;**

**public class SameThreadExecutor implements Executor**

**{**

**@Override**

**public void execute(Runnable command)**

**{**

**command.run();**

**//caller waits**

**}**

**}**

Whereas, the following thread executor implementation would use a new thread for each task:

**import java.util.concurrent.Executor;**

**public class NewThreadExecutor implements Executor**

**{**

**@Override**

**public void execute(Runnable command)**

**{**

**Thread t=new Thread(command);**

**command.run();**

**//caller waits**

**}**

**}**

There is no “right number” of threads for task execution. The type of task(CPU intensive vs I/O intensive), number of tasks, I/O latency, and system resources all factor into determining the ideal number of threads to use. You should perform testing your applications to determine the ideal threading model. That is one reason why the ability too separate task submission from task execution is important.

**Difference Between Threads And Executors?**

1. First and foremost difference between Thread and Executor is that java.lang.Thread is a [class](http://www.java67.com/2016/08/difference-between-class-and-interface-in-java.html" \t "https://javarevisited.blogspot.com/2016/12/_blank) in Java while java.util.concurrent.Executor is an [interface](http://www.java67.com/2014/02/what-is-actual-use-of-interface-in-java.html" \t "https://javarevisited.blogspot.com/2016/12/_blank).
2. The Executor concept is actually an abstraction over parallel computation. It allows concurrent code to be run in managed way. On the other hand, Thread is a concrete way to run the code in parallel.
3. The third difference between an Executor and a Thread class is that former decouples a task (the code which needs to be executed in parallel) from execution, while in the case of a Thread, both task and execution are tightly coupled.
4. The executor concept can be implemented in many ways, (you can allocate a single thread to execute all tasks, you can have a new thread every time when you want to execute a task and many other ways) while Thread itself execute your task.
5. One more key difference between a Thread and an Executor is that a Thread can only execute one Runnable task but an Executor can execute any number of Runnable task. (by initializing executor with newFixedThreadPool or something)

MyCallable callable1 = new MyCallable(1000);

MyCallable callable2 = new MyCallable(2000);

FutureTask<String> futureTask1 = new FutureTask<String>(callable1);

FutureTask<String> futureTask2 = new FutureTask<String>(callable2);

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.execute(futureTask1);

executor.execute(futureTask2);

**More About Executor, Executors, ScheduledExecutorService:**

**Executor:** is the super type of all executors. It defines only one method execute(Runnable).

**ExecutorService:** is an Executor that allows tracking progress of value-returning tasks (Callable) via Future object, and manages the termination of threads. Its key methods include submit() and shutdown().

**ScheduledExecutorService:** is an ExecutorService that can schedule tasks to execute after a given delay, or to execute periodically. Its key methods are schedule(), scheduleAtFixedRate() and scheduleWithFixedDelay().

You can create an executor by using one of several factory methods provided by **the Executors utility class.**

Here’s to name a few:

**newCachedThreadPool():** creates an expandable thread pool executor. New threads are created as needed, and previously constructed threads are reused when they are available. Idle threads are kept in the pool for one minute. This executor is suitable

for applications that launch many short-lived concurrent tasks.

**newFixedThreadPool(int n):** creates an executor with a fixed number of threads in the pool. This executor ensures that there are no more than n concurrent threads at any time. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread becomes available. If any thread terminates due to failure during execution, it will be replaced by a new one. The threads in the pool will exist until it is explicitly shutdown. Use this executor if you and to limit the maximum number of concurrent threads.

**newSingleThreadExecutor():** creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another.

**newScheduledThreadPool(int corePoolSize):** creates an executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute concurrently.

**newSingleThreadScheduleExecutor():**

creates a single-threaded executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute sequentially.

**Callable Interface:**

Now, In Java 5, java.util.concurrent was introduced. The callable interface was introduced in concurrency package, which is similar to the Runnable interface, but it can return any object, and is able to throw an Exception.

An ExecutorService might can be passed a Callable instead of a Runnable.

The primary benefit of using a callable is the ability to return a result. Because, an ExecutorService may execute the Callable interface asynchronously, (just like a runnable), you need a way to check the completion status of a Callable and obtain the result later. Now, java.util.concurrent.Future is used to obtain the status and result of a callable. Without a Future object, we neither get the completion status of a Callable nor obtain the result.

**Future Object:**

Java Callable tasks return java.util.concurrent.Future objects. Java Future provides a cancel() method to cancel the associated Callable task. There are two get methods, one is an overloaded version of the other. The get() with no arguments wait for the computation to be completed and gets the result and get() method with arguments Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available. Where we can specify the time to wait for the result. It’s useful to avoid a current thread getting blocked for a longer time. Please note that the get method is a synchronous method. Until the callable finishes its task and returns a value, it will wait for a callable. There are also isDone() and isCancelled() methods to find out the current status of an associated Callable task.

**import java.util.concurrent.Callable;**

**public class MyCallable implements Callable<String> {**

**private long waitTime;**

**public MyCallable(int timeInMillis){**

**this.waitTime=timeInMillis;**

**}**

**@Override**

**public String call() throws Exception {**

**Thread.sleep(waitTime);**

**//return the thread name executing this callable task**

**return Thread.currentThread().getName();**

**}**

**}**

Here, we create a simple callable implementation.

It waits for some time and return the name of the calling thread.

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.FutureTask;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.TimeoutException;

public class FutureTaskExample {

public static void main(String[] args) {

MyCallable callable1 = new MyCallable(1000);

MyCallable callable2 = new MyCallable(2000);

FutureTask<String> futureTask1 = new FutureTask<String>(callable1);

FutureTask<String> futureTask2 = new FutureTask<String>(callable2);

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.execute(futureTask1);

executor.execute(futureTask2);

while (true) {

try {

if(futureTask1.isDone() && futureTask2.isDone()){

System.out.println("Done");

//shut down executor service

executor.shutdown();

return;

}

if(!futureTask1.isDone()){

//wait indefinitely for future task to complete

System.out.println("FutureTask1 output="+futureTask1.get());

}

System.out.println("Waiting for FutureTask2 to complete");

String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);

if(s !=null){

System.out.println("FutureTask2 output="+s);

}

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}catch(TimeoutException e){

//do nothing

}

}

}

}

This will generate the following output:

FutureTask1 output=pool-1-thread-1

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

FutureTask2 output=pool-1-thread-2

Done

**A Simple Example Of Executor Using Single Threaded Executor:**

import java.util.concurrent.\*;

/\*\*

 \* SimpleExecutorExample.java

 \* This program demonstrates how to create a single-threaded executor

 \* to execute a Runnable task.

 \* @author www.codejava.net

 \*/

public class SimpleExecutorExample {

    public static void main(String[] args) {

        ExecutorService pool = Executors.newSingleThreadExecutor();

        Runnable task = new Runnable() {

            public void run() {

                System.out.println(Thread.currentThread().getName());

            }

        };

        pool.execute(task);

**//Now, since, there is only one task, we need not to submit it**

**//we can execute it directly**

        pool.shutdown();

    }

}

Now, Single thread executor is used in the following condition:

creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another. And ExecutorService is an Executor that allows tracking progress of value-returning tasks (Callable) via Future object, and manages the termination of threads.

As you can see, a Runnable task is created using anonymous-class syntax. The task simply prints the thread name and terminates. Compile and run this program and you will see the output something like this:

pool-1-thread-1

Note that you should call shutdown() to destroy the executor after the thread completes execution. Otherwise, the program is still running afterward. You can observe this behavior by commenting the call to shutdown.

And the following program shows you how to submit a Callable task to an executor. A Callable task returns a value upon completion and we use the Future object to obtain the value.

**A Complex Example:  
  
public class SimpleExecutorServiceExample {**

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

**ExecutorService pool = Executors.newSingleThreadExecutor();**

**Callable<Integer> task = new Callable<Integer>() {**

**public Integer call() {**

**try {**

**// fake computation time**

**Thread.sleep(5000);**

**} catch (InterruptedException ex) {**

**ex.printStackTrace();**

**}**

**return 1000;**

**}**

**};**

**Future<Integer> result = pool.submit(task);**

**try {**

**Integer returnValue = result.get();**

**System.out.println("Return value = " + returnValue);**

**} catch (InterruptedException | ExecutionException ex) {**

**ex.printStackTrace();**

**}**

**pool.shutdown();**

**}**

**}**

Now, there are two ways to assign a Future object to get result after execution of callable interface.

FutureTask<String> futureTask1 = new FutureTask<String>(callable1);

Pool.execute(callable1);

Or,   Future<Integer> result = pool.submit(task);