Java provides multithreading to parallelize execution of tasks (code) and you need threads to run multiple things in parallel.

You can create Thread in Java programming language by either

1. extending Thread class ,
2. implementing Runnable or Callable
3. by using Executor framework in Java. (not used to create thread)

There are two kinds of Thread in Java daemon or non daemon (also called user threads).

Java programs runs until there is at least one non-daemon thread exists. First non-daemon thread started by JVM is main thread which is created by JVM and responsible for executing code inside [main method in Java](http://java67.blogspot.sg/2012/08/what-is-main-method-in-java-why-main-is.html). This is called "VM thread" in HotSpot Virtual Machine. Any thread created using java.lang.Thread start() methods from main thread is by default non-daemon but you can make it daemon by calling setDaemon(true) method. Newly created thread in Java inherits daemon property from the thread which creates it. Since main thread is non-daemon, any thread created by it by default remains non-daemon.

**major difference between Process and Thread** is that, each process has its own separate memory space but Threads from same process same memory space

**Extends Thread vs implements Runnable**

1. a class can only extend one class in Java. So if you extend the Thread class then your class lose that option and it cannot extend another class
2. you implement Runnable, both Task and Executor ( a thread which execute the task) are loosely coupled but if you extend Thread then they are tightly coupled.
3. encapsulated in Runnable interface because you only need to make the change in one place but if that code is scattered around multiple Thread class, you need to make the change at multiple places.

you need to first define the task which will be executed by those threads. In order to create those task, you can either use [Runnable](http://java67.blogspot.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html) or [Callable](http://java67.blogspot.com/2013/01/difference-between-callable-and-runnable-java.html) interface.

## Callable vs Runnable interface in Java

1. to leverage additional features offered by Callable like it can throw an exception and it can also return value.
2. Runnable interface has run() method to define task while Callable interface uses call() method for task definition
3. run() method does not return any value, it's return type is void while call method returns value. The Callable interface is a [generic parameterized interface](http://javarevisited.blogspot.sg/2012/08/how-to-write-parametrized-class-method-Generic-example.html) and Type of value is provided when an instance of Callable implementation is created.
4. Another difference on run and call method is that run method can not [throw](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) checked exception while call method can throw checked exception in Java.
5. Use “execute” to put task in queue in Runnable and use “submit” to put task in queue in callable

you shouldn't create too many Thread instances in Java because both JVM and Operating system has a limit on how many threads you can create in Java.

You might already know that just creating an instance of java.lang.Thread class doesn't start a new thread, you need to start each thread manually by calling the [start()](http://javarevisited.blogspot.sg/2014/09/common-java-multi-threading-mistakes-1-run-vs-start.html) method of Thread class. This method first creates a thread and then call the run() method of Runnable task you have passed to this new thread.

If you directly call run() method then the code will be executed in the same thread on which you called run() method, multi-threading or concurrency will not be achieved

public class MultipleThreadDemo {

private static class ParallelTask implements *Runnable* {

@Override

public void run() {

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

+ " is executing this code");

}

}

public static void main(String[] args) {

// created three threads but none will start until you call

// start() method

Thread t1 = new Thread(new ParallelTask(), "Thread - T1");

Thread t2 = new Thread(new ParallelTask(), "Thread - T2");

Thread t3 = new Thread(new ParallelTask(), "Thread - T3");

// now, let's start all three threads

t1.start();

t2.start();

t3.start();

}

}

Output

Thread - T1 is executing this code

Thread - T3 is executing this code

Thread - T2 is executing this code

As you might already know that you [can not reuse this thread again](http://java67.blogspot.com/2014/01/10-points-about-thread-and-javalangthread-in-java.html). Calling the start() method on such thread will result in IllegalThreadStateException.

**Thread Pool:**

Thread pool is a pool of already created worker thread ready to do the job. The thread pool is one of essential facility any multi-threaded server side Java application requires

The core of this thread pool framework is Executor interface which defines an abstraction of task execution with method execute(Runnable task) and ExecutorService which extends Executor to add various life-cycle and thread pool management facilities like shutting down thread pool.

Executor framework also provides a [static utility class](http://javarevisited.blogspot.com/2013/03/difference-between-singleton-pattern-vs-static-class-java.html) called Executors ( similar to Collections) which provides several static factory method to create various type of Thread Pool implementation in Java e.g. fixed size thread pool, cached thread pool and scheduled thread pool. Runnable and Callable interface are used to represent task executed by worker thread managed in these Thread pools.

Thread pool will take care of how to execute that task, it can be executed by any free worker thread and if you are interested in result you can query Future object returned by submit() method. Executor framework also provides different kind of Thread Pool e.g. **SingleThreadExecutor** which creates just one worker thread or **CachedThreadPool** which creates worker threads as and when necessary

public class *ThreadPoolExample* {

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

ExecutorService service **=** Executors**.**newFixedThreadPool(*10*);

**for** (int i **=***0*; i**<***100*; i**++**){

service**.**submit(**new** Task(i));

}

}

}

final class *Task* implements *Runnable*{

private int taskId;

public **Task**(int *id*){

*this***.**taskId **=** id;

}

@Override

public void **run**() {

System**.**out**.**println("Task ID : " **+** *this***.**taskId **+**" performed by "   
                           **+** Thread**.**currentThread()**.**getName());

}

}

**Output:**

Task *ID* **:** *0* performed by pool**-***1***-**thread**-***1*

Task *ID* **:** *3* performed by pool**-***1***-**thread**-***4*

Task *ID* **:** *2* performed by pool**-***1***-**thread**-***3*

Task *ID* **:** *1* performed by pool**-***1***-**thread**-***2*

Task *ID* **:** *5* performed by pool**-***1***-**thread**-***6*

Task *ID* **:** *4* performed by pool**-***1***-**thread**-***5*

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

// Runnable, return void, nothing, submit and run the task async

executor.submit(() -> System.out.println("I'm Runnable task."));

// Callable, return a future, submit and run the task async

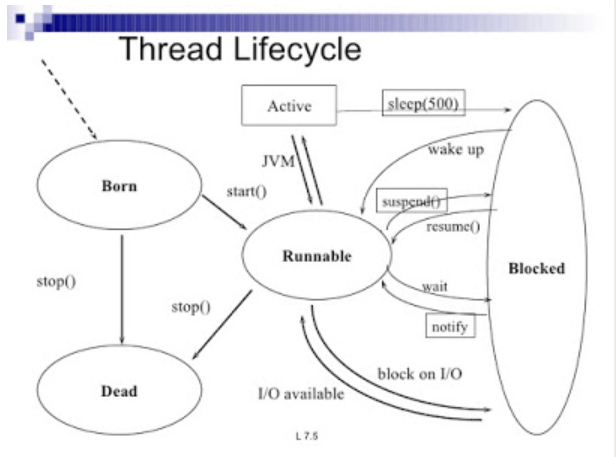
Future<Integer> futureTask1 = executor.submit(() -> {

System.out.println("I'm Callable task.");

return 1 + 1;

});

1. Use of Thread Pool reduces response time by avoiding thread creation during request or task processing.
2. Use of Thread Pool allows you to change your execution policy as you need.
3. Thread Pool in Java application increases the stability of the system by creating a configured number of threads decided based on system load and available resource.
4. Thread Pool frees application developer from thread management stuff and allows to focus on business logic.



<https://www.java67.com/2014/01/10-points-about-thread-and-javalangthread-in-java.html>

During its life time thread remains on various Thread states like NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING which describe what thread is doing. NEW means thread is just created but not yet stated, RUNNABLE means thread is started but waiting for CPU to be assigned by thread scheduler. BLOCKED, WAITING and TIMED\_WAITING means thread is not doing anything instead its been blocked and waiting for IO to finished, class or object lock, or any other thread etc.

**Wait() vs Thread.sleep() :**

1. wait method must be called from synchronized context i.e. from synchronized method or block in Java. If you call wait method without synchronization, it will throw IllegalMonitorStateException in Java. On the other hand there is no requirement of synchronization for calling sleep method , you can call it normally.
2. wait operates on Object and defined in Object class while sleep operates on current Thread and defined in java.lang.Thread class.
3. wait() method releases the lock of object on which it has called, it does release other locks if it holds any , while sleep method of Thread class does not release any lock at all.
4. Thread will be resumed only after call of notify()or notifyAll() method by other thread . sleeping thread will be resumed after time exceeded.
5. Use wait() and notify() method for inter thread communication while use sleep() method for introducing small pause during thread execution.
6. **synchronized**(monitor)  
   **while**(condition == **true**){ monitor.wait())  *//releases monitor lock*  
     
   **Thread**.sleep(100); *//puts current thread on Sleep*

wait() and notify() methods are used to communicate between two threads i.e. for inter thread communication in Java.

Java provides interrupt() method to interrupt a thread in Java. You can interrupt a running thread, waiting thread or sleep thread. This is the control Java provides to prevent a blocked or hanged thread. Once you interrupt a thread, it will also throw InterruptedException, which is a [checked exception](http://java67.blogspot.sg/2012/12/difference-between-runtimeexception-and-checked-exception.html) to ensure that your code should take handle interrupts.

## Difference between wait and yield in Java

1. wait vs yield method is that, wait() is Overloaded method declared in java.lang.Object class while Yield is static method declared on java.lang.Thread class.
2. Another *difference on wait and yield* is that When a Thread call waits it releases the monitor.

Java provides two ways to achieve mutual exclusion in your code,

1. either by using synchronized keyword or
2. by using java.util.concurrent.lock implementations.

You can use synchronized keyword to either make an entire method mutual exclusive or only critical section by declaring a [synchronized block](http://java67.blogspot.sg/2013/01/difference-between-synchronized-block-vs-method-java-example.html).

**Synchronization:**

Synchronization in Java is an important concept since Java is a multi-threaded language where multiple threads run in parallel to complete program execution.

**Java synchronized method and blocks.**

Concurrent access of shared objects in Java introduces to kind of errors: thread interference and memory consistency errors and to avoid these errors you need to properly synchronize your Java object to allow mutual exclusive access of critical section to two threads. to avoid any corruption of state or any kind of unexpected behavior.

***Java synchronized code will only be executed by one thread at a time***

synchronized variable in Java, you can have java volatile variable, which will instruct JVM threads to read the value of the volatile variable from main memory and don’t cache it locally.

**Synchronized keyword** along with method is easy just apply synchronized keyword in front of the method. What we need to take care is that static synchronized method locked on class object lock and nonstatic synchronized method locks on current object (this). So it’s possible that both static and nonstatic java synchronized method running in parallel.

**public** **class** **Counter**{

**private** **static** **int** count = **0**;

**public** **static** **synchronized** **int** **getCount**(){

**return** count;

}

**public** synchoronized **setCount**(**int** count){

**this**.count = count;

}

}

**synchronized block in java** is also similar to using **synchronized keyword in methods**. Only important thing to note here is that if object used to lock synchronized block of code, Singleton.class in below example is null then Java synchronized block will throw a [NullPointerException](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html). This is a classic example of [double checked locking in Singleton](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html).

**public** **class** **Singleton**{

**private** **static** **volatile** Singleton \_instance;

**public** **static** Singleton **getInstance**(){

**if**(\_instance == **null**){

**synchronized**(Singleton.class){

**if**(\_instance == **null**)

\_instance = **new** Singleton();

}

}

**return** \_instance;

}

1. Whenever a thread enters into java synchronized method or blocks it **acquires a lock** and whenever it leaves java synchronized method or block it releases the lock.
2. You can use java synchronized keyword only on synchronized method or synchronized block.
3. Java Thread acquires an **object level lock** when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.
4. Java Synchronization will throw NullPointerException if object used in java synchronized block is nulle.g. synchronized (myInstance) will throw java.lang.NullPointerException if myInstance is null
5. **Java synchronized block is better than java synchronized method** in Java because by using synchronized block you can only lock critical section of code and avoid locking the whole method which can possibly degrade performance.
6. **Java synchronized code could result in deadlock or starvation** while accessing by multiple threads if synchronization is not implemented correctly
7. **you can not use Java synchronized keyword with constructor** it’s illegal and result in compilation error.
8. **Do not synchronize on the non-final field on synchronized block in Java**

ReentrantLock

ReentrantLock is a concrete implementation of Lock [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) provided in Java concurrency package from

with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) until a call to unlock() method

1. synchronized and Reentrant lock is **tryLock()** method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread

1) Ability to lock interruptibly.

2) Ability to timeout while waiting for lock.

3) Power to create fair lock.

4) API to get list of waiting thread for lock.

5) Flexibility to try for lock without blocking

public class ReentrantLockHowto {

private final ReentrantLock lock = new ReentrantLock();

private int count = 0;

//Locking using Lock and ReentrantLock

public int getCount() {

lock.lock();

try {

System.out.println(Thread.currentThread().getName() + " gets Count: " + count);

return count++;

} finally {

lock.unlock();

}

}

//Implicit locking using synchronized keyword

public synchronized int getCountTwo() {

return count++;

}

public static void main(String args[]) {

final ThreadTest counter = new ThreadTest();

Thread t1 = new Thread() {

@Override

public void run() {

while (counter.getCount() &lt; 6) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace(); }

}

}

};

Thread t2 = new Thread() {

@Override

public void run() {

while (counter.getCount() &lt; 6) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

}

}

};

t1.start();

t2.start();

}

}

"**What is a deadlock?**"  
The answer is simple when two or more threads are waiting for each other to release the resource they need (lock) and get stuck for infinite time, the situation is called deadlock

## detect deadlock in Java?

first I would look at the code if I see a nested synchronized block or calling one synchronized method from other, or trying to get a lock on a different object then there is a good chance of deadlock if a developer is not very careful.

try to take a thread dump, in Linux you can do this by command "kill -3", this will print status of all threads in an application log file, and you can see which thread is locked on which object.

Another way is to use the **jConsole/VisualVM**, it will show you exactly which threads are getting locked and on which object.

public class DeadLockDemo {

/\*

\* This method request two locks, first String and then Integer

\*/

public void method1() {

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

}

}

}

/\*

\* This method also requests same two lock but in exactly

\* Opposite order i.e. first Integer and then String.

\* This creates potential deadlock, if one thread holds String lock

\* and other holds Integer lock and they wait for each other, forever.

\*/

public void method2() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

}

# Java CountDownLatch

CountDowaLatch is a high-level synchronization utility which is used to prevent a particular thread to start processing until all threads are ready. This is achieved by a countdown. The thread, which needs to wait for starts with a counter, each thread them make the count down by 1 when they become ready, once the last thread call countDown() method, then the latch is broken and the thread waiting with counter starts running.

**CountDownLatch starts with a fixed number of counts**

CyclicBarrier, which can also be used in this situation, where one thread needs to wait for other threads before they start processing. Only difference between CyclicBarrier and CountDownLatch is that you can reuse the barrier even after its broker but you cannot reuse the count down latch, once count reaches to zero.

public class CountDownLatchDemo {

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

CountDownLatch latch = new CountDownLatch(4);

Worker first = new Worker(1000, latch, "WORKER-1");

Worker second = new Worker(2000, latch, "WORKER-2");

Worker third = new Worker(3000, latch, "WORKER-3");

Worker fourth = new Worker(4000, latch, "WORKER-4");

first.start();

second.start();

third.start();

fourth.start();

// Main thread will wait until all thread finished

latch.await();

System.out.println(Thread.currentThread().getName() + " has finished");

}

}

class Worker extends *Thread* {

private int delay;

private CountDownLatch latch;

public Worker(int delay, CountDownLatch latch, String name) {

super(name);

this.delay = delay;

this.latch = latch;

}

@Override

public void run() {

try {

Thread.sleep(delay);

latch.countDown();

System.out.println(Thread.currentThread().getName() + " has finished");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Output

WORKER-1 has finished

WORKER-2 has finished

WORKER-3 has finished

WORKER-4 has finished

main has finished

# CyclicBarrier in Java

CyclicBarrier, which can also be used in this situation, where one thread needs to wait for other threads before they start processing.when last thread calls await() which signals that it has reached the barrier, all thread started processing again, also known as a barrier is broken

public class HelloHP {

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

CyclicBarrier barrier = new CyclicBarrier(4);

Party first = new Party(1000, barrier, "PARTY-1");

Party second = new Party(2000, barrier, "PARTY-2");

Party third = new Party(3000, barrier, "PARTY-3");

Party fourth = new Party(4000, barrier, "PARTY-4");

first.start();

second.start();

third.start();

fourth.start();

System.out.println(Thread.currentThread().getName() + " has finished");

}

}

class Party extends *Thread* {

private int duration;

private CyclicBarrier barrier;

public Party(int duration, CyclicBarrier barrier, String name) {

super(name);

this.duration = duration;

this.barrier = barrier;

}

@Override

public void run() {

try {

Thread.sleep(duration);

System.out.println(Thread.currentThread().getName() + " is calling await()");

barrier.await();

System.out.println(Thread.currentThread().getName() + " has started running again");

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

Output

main has finished

PARTY-1 is calling await()

PARTY-2 is calling await()

PARTY-3 is calling await()

PARTY-4 is calling await()

PARTY-4 has started running again

PARTY-1 has started running again

PARTY-2 has started running again

PARTY-3 has started running again

# Semaphore in Java

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource.

1. Semaphores – Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.
2. Mutex – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

// 5 tickets

Semaphore semaphore = new Semaphore(5);

// take 1 ticket

semaphore.acquire();

// 4

semaphore.availablePermits();

// return back ticket

semaphore.release();

// 5

semaphore.availablePermits();

Producer and Consumer:

**public** **class** **InterThreadCommunicationExample** {

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

**final** Queue sharedQ = **new** LinkedList();

        Thread producer = **new** Producer(sharedQ);

        Thread consumer = **new** Consumer(sharedQ);

        producer.start();

        consumer.start();

    }

}

**public** **class** **Producer** **extends** Thread {

**private** **static** **final** Logger logger = Logger.getLogger(Producer.class);

**private** **final** Queue sharedQ;

**public** **Producer**(Queue sharedQ) {

**super**("Producer");

**this**.sharedQ = sharedQ;

    }

    @Override

**public** **void** **run**() {

**for** (**int** i = **0**; i < **4**; i++) {

**synchronized** (sharedQ) {

                //waiting condition - wait until Queue is not empty

**while** (sharedQ.size() >= **1**) {

**try** {

                        logger.debug("Queue is full, waiting");

                        sharedQ.wait();

                    } **catch** (InterruptedException ex) {

                        ex.printStackTrace();

                    }

                }

                logger.debug("producing : " + i);

                sharedQ.add(i);

                sharedQ.notify();

            }

        }

    }

}

**public** **class** **Consumer** **extends** Thread {

**private** **static** **final** Logger logger = Logger.getLogger(Consumer.class);

**private** **final** Queue sharedQ;

**public** **Consumer**(Queue sharedQ) {

**super**("Consumer");

**this**.sharedQ = sharedQ;

    }

    @Override

**public** **void** **run**() {

**while**(**true**) {

**synchronized** (sharedQ) {

                //waiting condition - wait until Queue is not empty

**while** (sharedQ.size() == **0**) {

**try** {

                        logger.debug("Queue is empty, waiting");

                        sharedQ.wait();

                    } **catch** (InterruptedException ex) {

                        ex.printStackTrace();

                    }

                }

**int** number = sharedQ.poll();

                logger.debug("consuming : " + number );

                sharedQ.notify();

                //termination condition

**if**(number == **3**){**break**; }

            }

        }

    }

}

Producer and Consumer : BlockingQueue

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

/\*\*

\* Producer Consumer Problem solution using BlockingQueue in Java.

\* BlockingQueue not only provide a data structure to store data

\* but also gives you flow control, require for inter thread communication.

\*

\* @author Javin Paul

\*/

public class ProducerConsumerSolution {

public static void main(String[] args) {

BlockingQueue<Integer> sharedQ = new LinkedBlockingQueue<Integer>();

Producer p = new Producer(sharedQ);

Consumer c = new Consumer(sharedQ);

p.start();

c.start();

}

}

class Producer extends *Thread* {

private BlockingQueue<Integer> sharedQueue;

public Producer(BlockingQueue<Integer> aQueue) {

super("PRODUCER");

this.sharedQueue = aQueue;

}

public void run() {

// no synchronization needed

for (int i = 0; i < 10; i++) {

try {

System.out.println(getName() + " produced " + i);

sharedQueue.put(i);

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer extends *Thread* {

private BlockingQueue<Integer> sharedQueue;

public Consumer(BlockingQueue<Integer> aQueue) {

super("CONSUMER");

this.sharedQueue = aQueue;

}

public void run() {

try {

while (true) {

Integer item = sharedQueue.take();

System.out.println(getName() + " consumed " + item);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

20

# Object level locking:

Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe. This can be done as below :

public class DemoClass

{

public synchronized void demoMethod(){}

}

or

public class DemoClass

{

public void demoMethod(){

synchronized (this)

{

//other thread safe code

}

}

}

or

public class DemoClass

{

private final Object lock = new Object();

public void demoMethod(){

synchronized (lock)

{

//other thread safe code

}

}

# Class level locking:

Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

public class DemoClass

{

public synchronized static void demoMethod(){}

}

or

public class DemoClass

{

public void demoMethod(){

synchronized (DemoClass.class)

{

//other thread safe code

}

}

}

or

public class DemoClass

{

private final static Object lock = new Object();

public void demoMethod(){

synchronized (lock)

{

//other thread safe code

}

}

}