MultiThreading

MultiThreading

Using multithreading in programs gives you the opportunity to solve a few of problems:

- ❖ To accelerate the user interface's response to the user's actions, by placing long-term tasks (large calculations, port waiting, etc.) in separate threads;
- ❖ Full using of the power of multi-core machines.

However, we recommend that you do not abuse a large number of threads, as the Operating System spends time tracking the number of threads and their maintenance.

MultiTasking

Multitasking is a process of executing multiple tasks simultaneously.

Multitasking can be achieved by two ways:

- ❖ **Process-based Multitasking (Multiprocessing)**
- ❖ **Thread-based Multitasking (Multithreading)**

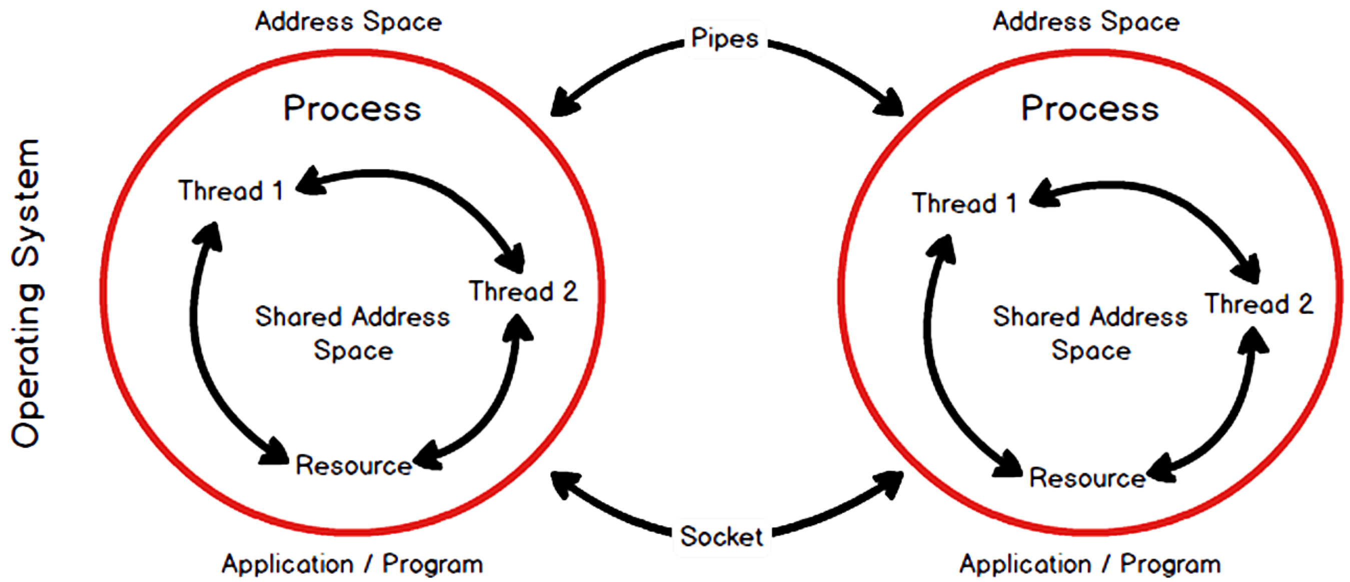
MultiTasking

- ❖ **In multitasking**, the system allows executing multiple programs and tasks at the same time, whereas, **in multithreading**, the system executes multiple threads of the same or different processes at the same time.

- ❖ **In multitasking** CPU has to switch between multiple programs so that it appears that multiple programs are running simultaneously. On other hands, **in multithreading** CPU has to switch between multiple threads to make it appear that all threads are running simultaneously.

- ❖ **Multitasking** allocates separate memory and resources for each process/program whereas, **in multithreading** threads belonging to the same process shares the same memory and resources as that of the process.

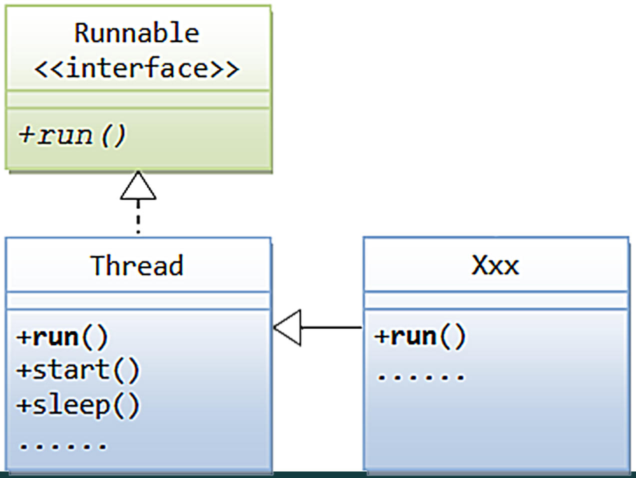
A Process and a Thread

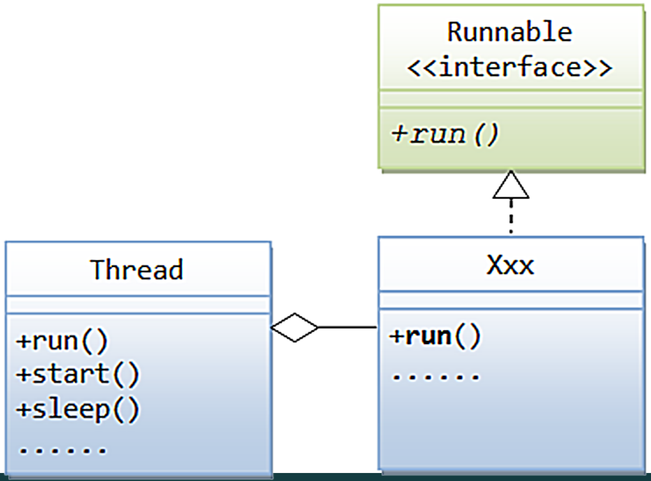


How to make Thread

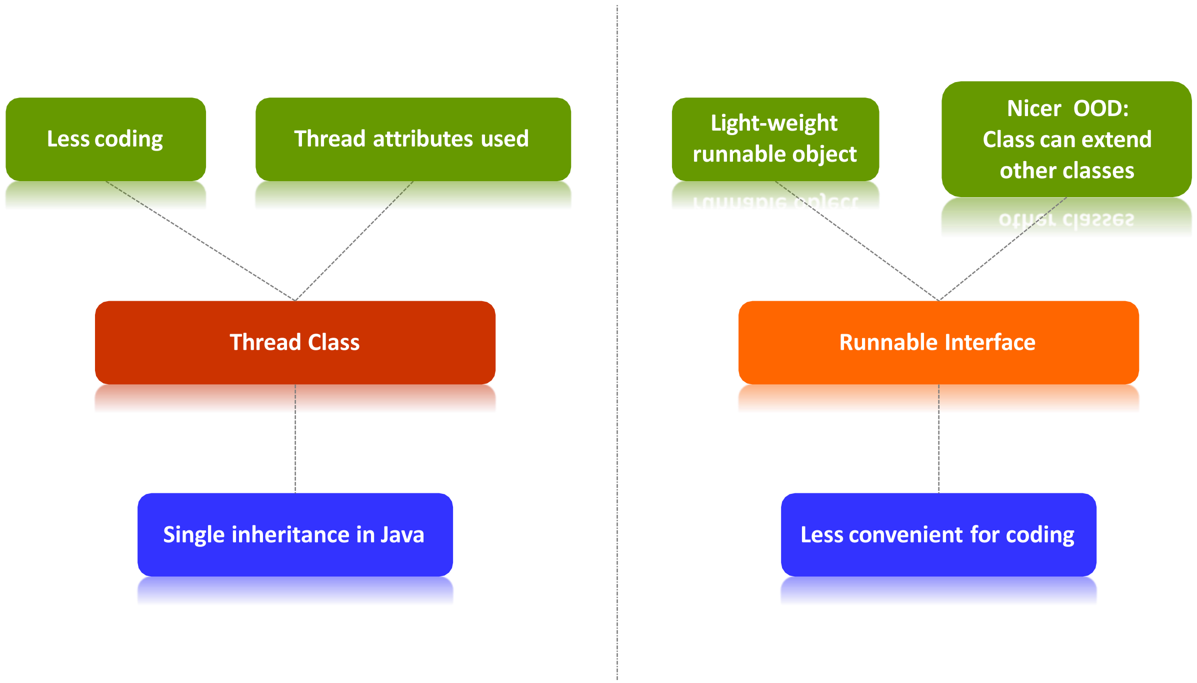
There are two ways of implementing threading in Java (since JDK 1.0):

- ❖ By extending **java.lang.Thread** class
- ❖ By implementing **java.lang.Runnable** interface.





Thread vs. Runnable


Less coding
Thread attributes used
Thread Class
Single inheritance in Java
Light-weight runnable object
Nicer OOD: Class can extend other classes
Runnable Interface
Less convenient for coding

How to make Thread

```java
public class ExampleThreads {
  public class MyThread extends Thread {
    public void run() {
      System.out.println("Hello, my name is " + Thread.currentThread().getName());
  } }

  public class MyRunnable implements Runnable {
    public void run() {
      System.out.println("Hello, my name is " + Thread.currentThread().getName());
  } }


  public void show() {
    MyThread myThread = new MyThread();
    myThread.setName("T1");
    Thread myRunnable = new Thread(new MyRunnable(), "T2");
    myThread.start();
    myRunnable.start();
  }
}
```

```
Hello, my name is T2
Hello, my name is T1
```

How to make Thread

```
public void show() {

  Thread myThread = new Thread() {
    // Create an anonymous inner class extends Thread
    @Override
    public void run() {
      System.out.println("Hello, my name is " +  Thread.currentThread().getName());
    } };
  myThread.setName("T3");
  myThread.start();

  Thread myRunnable = new Thread(new Runnable() {
    // Create an anonymous inner class that implements Runnable
    public void run() {
      System.out.println("Hello, my name is " + Thread.currentThread().getName());
    } }, "T4");
  myRunnable.start();

}
```

```
Hello, my name is T3
Hello, my name is T4
```

How to make Thread with Lambda expression

```
public void show() {

  Thread myRunnable = new Thread(
     () -> System.out.println("Hello, my name is " + Thread.currentThread().getName()), "T4");
  myRunnable.start();

}
```

Thread Class Constructors

Thread()

Thread(String name)

Thread (Runnable target)

Thread (Runnable target, String name)

Thread(ThreadGroup group, Runnable target)

Thread (ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, String name)

Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Methods to manage the Threads

Start() *// When program calls start() method a new Thread is created*
// and code inside run() method is executed in new Thread

Run() *// This method is the Entry point for the thread*

SetName(String Name) *// You can use to set the name of the thread*

String **GetName**()

static **Sleep**(Long Milliseconds) *// Suspends a thread for the specified period*

Boolean **IsAlive**() *// It determines if a thread is still running*

Join(Long Milliseconds) *// This method gets called when a thread wants*
// to wait for another thread to terminate

Methods to manage the Threads

boolean **isDaemon**()	*// Checks if the thread is a daemon thread*
setDaemon(boolean b)	*// Marks the thread as daemon or user thread*
Thread **CurrentThread**()	*// It returns the instance reference* *// of the currently executing thread*
Thread.State **GetState**()	*// It returns the state of the thread*
Yield()	*// This method causes the currently executing thread object* *// to pause temporarily and allow other threads to run*
Final Int **GetPriority**()	*// It returns the priority of the thread*
Final Void **SetPriority**(Int Priority)	*// This function is used to change* *// the priority of a thread*

Methods to manage the Threads

Int **GetId**() *// It returns the id of the thread*

Interrupt() *// It interrupts the thread*

Boolean **IsInterrupted**() *// Tests if the thread has been interrupted*

Boolean **Interrupted**() *// The static method tests*
// if the thread has been interrupted

~~**Suspend**~~() *// You can use it to suspend the thread*

~~**Resume**~~() *// You can use it to resume the suspended thread*

~~**Stop**~~() *// You can use it to halt the thread*

A sample code for Sleep() and IsAlive() methods

```
public void show() {
 Thread myThread = new Thread(
    () -> {
     System.out.println("My thread is in running state");
     try {
       Thread.sleep(4000);
     } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
    });

 myThread.start();
 int i = 1;
 while (myThread.isAlive()) {
   try {
     Thread.sleep(1000);
     System.out.println("sec=" + i++);
   } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
 }
 System.out.println("Thread-Sleep Demo Complete");
}
```

```
My thread is in running state
sec=1
sec=2
sec=3
sec=4
Thread-Sleep Demo Complete
```

A sample code for Join() method

```
public void show() {
  Thread myThread = new Thread(
     () -> {
       System.out.println("My thread is in running state");
       try {
         Thread.sleep(5000);
       } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
     });
  myThread.start();

  System.out.println(LocalDateTime.now());
  try {
    myThread.join();
  } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
  System.out.println(LocalDateTime.now());
  System.out.println("Thread-Sleep Demo Complete");
}
```

```
My thread is in running state
2018-05-02   12:05:48.559
2018-05-02   12:05:53.519
Thread-Sleep Demo Complete
```

A sample code for IsDaemon() and SetDaemon() methods

```
public static void main(String[] args) {
  Thread myThread = new Thread(
     () -> {
       for (int i = 1; i <= 5; i++) {
        try {   System.out.println("Daemon is " + Thread.currentThread().isDaemon () + ", sec=" + i);
                Thread.sleep(1000);
            } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
            }
            System.out.println("Good-bye!");
            });

  myThread.setDaemon(false);
  myThread.start();
  try {
  Thread.sleep(500);
  } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
  System.out.println("Finish");
}
```

```
Daemon is false, sec=1
Finish
Daemon is false, sec=2
Daemon is false, sec=3
Daemon is false, sec=4
Daemon is false, sec=5
Good-bye!
```

A sample code for IsDaemon() and SetDaemon() methods

```
Daemon is true, sec=1
Finish
```

```
public static void main(String[] args) {
  Thread myThread = new Thread(
     () -> {
       for (int i = 1; i <= 5; i++) {
        try {  System.out.println("Daemon is " + Thread.currentThread().isDaemon () + ", sec=" + i);
               Thread.sleep(1000);
          } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
          }
          System.out.println("Good-bye!");
          });

  myThread.setDaemon(true);
  myThread.start();
  try {
  Thread.sleep(500);
  } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
  System.out.println("Finish");
}
```

A sample code for interrupted() and interrupt() methods

```java
public static void main(String[] args) {
  Thread myThread = new Thread(
     () -> {
       System.out.println(LocalDateTime.now());
       while (true){
         if (Thread.interrupted()) {
           System.out.println(LocalDateTime.now());
           break;
         } else { // active work  }
       }
     });
  myThread.start();

  try {
    Thread.sleep(2000);
  } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
  myThread.interrupt();
  System.out.println(" Finish");
}
```

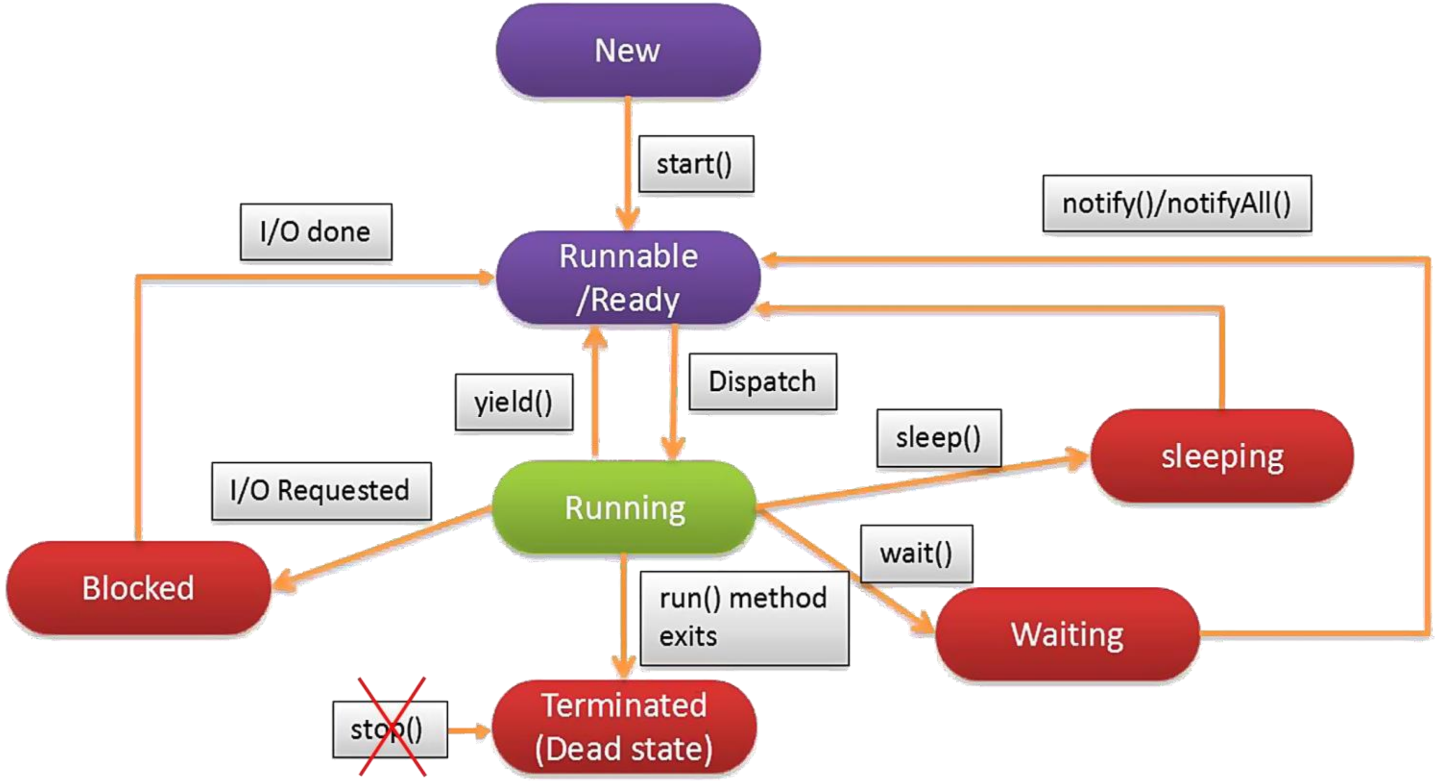
```
2018-05-02 12:48:42.605
 Finish
2018-05-02 12:48:44.566
```

A sample code for interrupted() and interrupt() methods

```
public static void main(String[] args) {
  Thread myThread = new Thread(
     () -> {
       for (int i = 1; i <= 10; i++) {
         try {
           System.out.println("sec=" + i);
           Thread.sleep(1000);
         } catch (InterruptedException  e)  {
                System.out.println("Sleeping thread interrupted"); break;
         }
       }  });

myThread.start();
  try {
    Thread.sleep(2000);
  } catch (InterruptedException e) { System.out.println("Sleeping thread interrupted"); }
  myThread.interrupt();
  System.out.println(" Finish");
 }
```

```
sec=1
sec=2
Finish
Sleeping thread interrupted
```

Life cycle of a Thread (Thread States)



Life cycle of a Thread

New – A thread is in New state when you create the instance of thread class but the start() method is yet to get called.

Runnable – The thread is in runnable after executing the start() method. In this stage, it waits for the thread scheduler to run it.

Running – A thread picked by the thread scheduler for execution remains in running state.

Non-Runnable (Blocked) – In this state, the thread remains alive but it is not eligible to run. It may be due to some sleep operation, waiting for any File I/O to finish or in the locked state, etc.

Waiting – As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed.

Terminated – A thread is said to be in the terminated state when its run() method exits.

Thread Priority

Priorities are represented by a number between 1 and 10. But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

<u>3 constants defined in Thread class:</u>

- ❖ public static int MIN_PRIORITY (1)
- ❖ public static int NORM_PRIORITY (5)
- ❖ public static int MAX_PRIORITY (10)

Thread Priority

```
public class TestMultiPriority extends Thread {
 public void run(){
  System.out.println("running thread name is: "+Thread.currentThread().getName());
  System.out.println("running thread priority is: "+Thread.currentThread().getPriority());
 }

 public static void main(String[] args) {
  TestMultiPriority m1=new TestMultiPriority();
  TestMultiPriority m2=new TestMultiPriority();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
  System.out.println(" Finish");
 }
}
```

```
 Finish
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10
```

ThreadGroup

- ❖ Every thread in JVM belongs to some thread group even if you, as the programmer, never specified any when the thread was created.
- ❖ By default, any newly created thread will belong to **main thread group** unless you specify otherwise at the time of creation.

Note

The assigning of a thread to a particular thread group can only be done at creation time otherwise it will be assigned to the main thread group implicitly. Once a thread has been assigned a thread group, this thread group will remain for the entire duration of the thread's lifetime.

Benefits of using Thread Groups

- ❖ Creates a logical grouping of related threads.
- ❖ Provides collective thread management.
- ❖ Get Aggregate CPU usage for the group of threads.
- ❖ Stack trace reveals thread group names providing more insight into offending thread.

ThreadGroup Methods

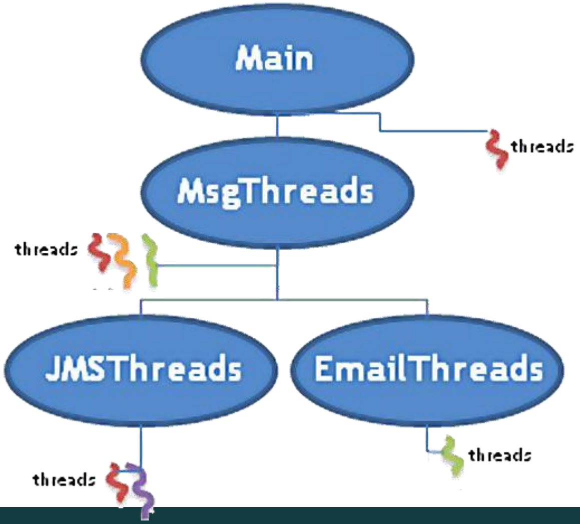
Method	Description
ThreadGroup(String name)	Create a thread group with name
ThreadGroup(String parent, String name)	Create a thread group under parent with name
int **activeCount**()	Returns the estimated of the number of active threads
int **activeGroupCount**()	Returns the estimated of the number of active groups
int **getMaxPriority**()	Returns the maximum priority of the thread group
String **getName**()	Returns the thread group name
int **getParent**()	Returns the parent of the thread group
void **interrupt**()	Interrupts all threads
boolean **isDaemon**()	Tests if the thread group is a daemon thread group
boolean **isDestroyed**()	Tests if the thread group has been destroyed
void **list**()	Prints information about this thread group to the standard output
void **setDaemon**(boolean daemon)	Changes the daemon status of the thread group
void **setMaxPriority**(int pri)	Sets the maximum priority of the thread group
String **toString**()	Returns a string representation

Thread Group Construction and Usage

```
ThreadGroup MsgThreads = new ThreadGroup("MsgThreads");

ThreadGroup JmsThreads = new ThreadGroup(MsgThreads, "JMSThreads");

Thread mt1 = new MyThread(MsgThreads, "msgThread_1");
```

Thread Group Example

```
public class MyThread extends Thread {
 public MyThread(ThreadGroup group, String name) {
    super(group, name);
 }
 @Override
 public void run() {
  for (int i=0; i<20000000; i++);
 } }
```



Thread Group Example

```
public static void main(String[] args) {
  ThreadGroup MsgThreads = new ThreadGroup("MsgThreads");
  ThreadGroup JmsThreads = new ThreadGroup(MsgThreads, "JMSThreads");
  ThreadGroup EmailThreads = new ThreadGroup(MsgThreads, "EmailThreads");

  new MyThread(MsgThreads, "msgThread_1").start();
  new MyThread(JmsThreads, "jmsThread_1").start();
  new MyThread(JmsThreads, "jmsThread_2").start();
  new MyThread(EmailThreads, "emailThread_1").start();
  new MyThread(EmailThreads, "emailThread_2").start();

  System.out.println("\nThreadGroup Name....: " + MsgThreads.getName());
  System.out.println("ThreadGroup Parent..: " + MsgThreads.getParent());
  System.out.println("Active Count........: " + MsgThreads.activeCount());
  System.out.println("Active Group Count..: " + MsgThreads.activeGroupCount());
  System.out.println("Max Priority........: " + MsgThreads.getMaxPriority());
  MsgThreads.setMaxPriority(6);
  System.out.println("Setting Group Priority to 6");
  System.out.println("Max Priority........: " + MsgThreads.getMaxPriority());
```

Thread Group Example

```
    System.out.println("\nThreadGroup Name....: " + EmailThreads.getName());
    System.out.println("ThreadGroup Parent..: " + EmailThreads.getParent());
    System.out.println("Active Count........: " + EmailThreads.activeCount());
    System.out.println("Active Group Count..: " + EmailThreads.activeGroupCount());
    System.out.println("Attempting to set Group Priority to 8 -- should not work!!!");
    EmailThreads.setMaxPriority(8);
    System.out.println("Max Priority........: " + EmailThreads.getMaxPriority());
    System.out.println("Attempting to set Group Priority to 4");
    EmailThreads.setMaxPriority(4);
    System.out.println("Max Priority........: " + EmailThreads.getMaxPriority());

    System.out.println("\nThreadGroup Name....: " + JmsThreads.getName());
    System.out.println("ThreadGroup Parent..: " + JmsThreads.getParent());
    System.out.println("Active Count........: " + JmsThreads.activeCount());
    System.out.println("Active Group Count..: " + JmsThreads.activeGroupCount());
    System.out.println("Max Priority........: " + JmsThreads.getMaxPriority());
  }
```

Thread Group Example

```
ThreadGroup Name....: MsgThreads
ThreadGroup Parent..: java.lang.ThreadGroup[name=main,maxpri=10]
Active Count........: 5
Active Group Count..: 2
Max Priority........: 10
Setting Group Priority to 6
Max Priority........: 6

ThreadGroup Name....: EmailThreads
ThreadGroup Parent..: java.lang.ThreadGroup[name=MsgThreads,maxpri=6]
Active Count........: 2
Active Group Count..: 0
Attempting to set Group Priority to 8 -- should not work!!!
Max Priority........: 6
Attempting to set Group Priority to 4
Max Priority........: 4

ThreadGroup Name....: JMSThreads
ThreadGroup Parent..: java.lang.ThreadGroup[name=MsgThreads,maxpri=6]
Active Count........: 2
Active Group Count..: 0
Max Priority........: 6
```

Competition for Resources

```java
public class Competition {
 private static long A = 0;
 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 1000000000; i++) { A++; }
   System.out.println("finish " + Thread.currentThread().getName());
  } }

 public void show() {
  MyThread t1 = new MyThread();
  MyThread t2 = new MyThread();
  MyThread t3 = new MyThread();
  t1.start();     t2.start();     t3.start();
  try {
  t1.join();      t2.join();     t3.join();
  } catch (InterruptedException e) { e.printStackTrace(); }
  System.out.println("A=" + A);
 }  }
```

Expected result:
3 000 000 000

Obtained result with 4-core PC

```
finish  Thread-1
finish  Thread-0
finish  Thread-2
A=1001468446
```

```
finish  Thread-0
finish  Thread-1
finish  Thread-2
A=1000039041
```

```
00:11:47.84

finish  Thread-2
finish  Thread-0
finish  Thread-1
A=1092937296

00:11:48.96
```

Synchronization in Java

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Thread Synchronization

There are two base types of thread synchronization:

- Mutual Exclusive
  - Synchronized method
  - Synchronized block
  - Static synchronization
- Cooperation (Inter-thread communication in java)
  - wait()
  - notify()
  - notifyAll()

The **synchronized** Keyword

The synchronized keyword can be used on different levels:

- Instance methods
- Static methods
- Code blocks

When we use a synchronized block, internally Java uses a monitor also known as monitor lock or intrinsic lock, to provide synchronization.

These monitors are bound to an object, thus all synchronized blocks of the same object can have only one thread executing them at the same time.

Synchronized Blocks within Methods

```
public class SyncClass {
 private int count = 0;

 private static Object obj = new Object();

 public void myMethod() {
  // code ...
  synchronized (obj) { // code ... }
  // code ...
 }

 public void otherMethod() {
  // code ...
  synchronized (obj) { // code ... }
  // code ...
 }
}
```

***Note**. The monitor cannot be null.*

Synchronized Instance Methods

```
public class SyncClass {

    private int count=0;

    public synchronized void add(int value){
        count += value;
    }

    public void add2(int value){
        synchronized(this){
            count += value;
        }
    }
}
```

Instance methods are synchronized over the instance of the class owning the method.

Which means only one thread per instance of the class can execute this method.

Synchronized Static Methods

```
public class SyncClass {

    private static int count=0;

    public static synchronized void add(int value){
        count += value;
    }

    public static void add2(int value){
        synchronized(SyncClass.class) {
            count += value;
        }
    }
}
```

These methods are synchronized on the Class object associated with the class and since only one Class object exists per JVM per class, only one thread can execute inside a static synchronized method per class, irrespective of the number of instances it has.

Synchronization Monitor

Internally Java uses a so called **monitor** also known as monitor lock or intrinsic lock in order to manage synchronization. This monitor is bound to an object, e.g. when using synchronized methods each method share the same monitor of the corresponding object.

Synchronized code	**Monitor**
Instance Method	**this** object
Static Method	**Class** object (for class in which the method is defined)
Code Block	object reference (not null) – parameter for synchronized

Synchronized Competition for Resources

```java
public class Competition {
 private static long A = 0;                        // Shared Resource
 private static Object sync = new Object();    // Monitor
 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 1000000000; i++) {  synchronized (sync){ A++; }   }
   System.out.println("finish " + Thread.currentThread().getName());
  }  }

 public void show() {
  MyThread t1 = new MyThread();
  MyThread t2 = new MyThread();
  MyThread t3 = new MyThread();
  t1.start();      t2.start();      t3.start();
  try {
  t1.join();       t2.join();      t3.join();
  } catch (InterruptedException e) { e.printStackTrace(); }
  System.out.println("A=" + A);
 }  }
```

Expected result:
3 000 000 000

Obtained result with 4-core PC

```
23:42:34.84

finish Thread-0
finish Thread-1
finish Thread-2
A=3 000 000 000

23:46:02.77
```

Lock Reentrance

Synchronized blocks in Java are reentrant.

This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object.

```java
public class Reentrant {

  public static synchronized void outer() {
    inner();
  }

  public static synchronized void inner() {
    //do something
  }
}
```

Lock Reentrance – Example

```
public class Reentrant {

  public static synchronized void outer() {
    inner();
  }

  public static synchronized void inner() {
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    System.out.println(Thread.currentThread().getName() + LocalDateTime.now());
  }

  public static void main(String[] args) {
    System.out.println("start " + LocalDateTime.now());
    Thread thread1 = new Thread(() -> outer(), "T1");
    Thread thread2 = new Thread(() -> outer(), "T2");
    thread1.start();
    thread2.start();
  }
}
```

```
start 15:14:08.738
T1    15:14:09.810
T2    15:14:10.811
```

Inter-thread communication

- A thread using the **wait()** method must own a lock on the object.
- Once **wait()** is called, the thread releases the lock, and waits for another thread to either call notify() or notifyAll() method.
- Once thread awakens after **notify()** or **notifyAll()** have been called, it will re-acquire the lock on the object and resume its normal execution.
- **wait()**, **notify()** and **notifyAll()** must be called inside of a synchronized block otherwise java.lang.IllegalMonitorStateException will be thrown.
- **wait()**, **notify()** and **notifyAll()** are found in the Object class.

Wait Methods

Method	Description
wait()	Waits indefinitely until another thread calls either the notify or notifyAll methods or the thread is interrupted by another thread.
wait(long millis)	Waits at most milliseconds until either notify or notifyAll gets called from another thread, is interrupted by another thread or the duration in milliseconds expires, called a *spurious wakeup.*
wait(long millis, int nanos)	Waits at most milliseconds plus nanos until either notify or notifyAll gets called from another thread, is interrupted by another thread or the duration in milliseconds plus nanoseconds expires, called a *spurious wakeup.*

notify() and notifyAll() Methods

- The java.long.Object.**notify()** method wakes up a single thread waiting on the object’s monitor. If there are any threads waiting on the object’s monitor, one of them will arbitrarily chosen by the scheduler and woken up. Once awakened, the chosen thread will need to wait until the current thread relinquishes control of the object’s lock before proceeding.

- The java.long.Object.**notifyAll()** method wakes all the threads waiting on the object’s monitor. Once awakened, these threads will need to wait until the current thread relinquishes control of the object’s lock before proceeding. All the threads will get a chance to execute once each one has the released control on the object’s lock.

Inter-thread communication (example)

```
public class Competition {
  private volatile static long A = 0;                  // Shared Resource
  private static Object sync = new Object();         // Monitor
  public void show() {
       Thread t1 = new Thread(() -> { synchronized (sync) {
           for (int i = 1; i <= 10000000; i++) {
             try {  sync.wait();  } catch (InterruptedException e) { }
             A++;
             sync.notify();
           }
           System.out.println("finish "+Thread.currentThread().getName());
         } });

       Thread t2 = new Thread(() -> { synchronized (sync) {
           for (int i = 1; i <= 10000000; i++) {
             sync.notify();
             try { sync.wait();  } catch (InterruptedException e) { }
             A++;
           }
           System.out.println("finish "+Thread.currentThread().getName());
         } });
```

Inter-thread communication (example)

```
        System.out.println(LocalDateTime.now());
        t1.start();
        t2.start();
        try { t1.join();  t2.join();
        } catch (InterruptedException e) { }
        System.out.println(LocalDateTime.now());
        System.out.println("A=" + A);
    }
}
```

Expected result:
20 000 000

Obtained result
with 4-core PC

```
14:35:40
finish Thread-0
finish Thread-1
14:37:09
A=20 000 000
```

Volatile Keyword

volatile is used to indicate that a **variable's value will be modified by different threads**.

Declaring a **volatile** Java variable means:

- ❖ The value of this variable will **never be cached thread-locally**: all reads and writes will go straight to "main memory";
- ❖ Access to the variable **acts as though it is enclosed in a synchronized block**, synchronized on itself.

We say "acts as though" in the second point, because to the programmer at least (and probably in most JVM implementations) there is no actual lock object involved.

Characteristic	**Synchronized**	**Volatile**
Type of variable	Object	Object or primitive
Null allowed?	No	Yes
Can block?	Yes	No
All cached variables synchronized on access?	Yes	From Java 5
When synchronization happens	When you explicitly enter/exit a synchronized block	Whenever a volatile variable is accessed.

Volatile Keyword

In other words, the main differences between synchronized and volatile are:

- a primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized);
- an access to a volatile variable **never has the potential to block**: we're only ever doing a simple read or write, so unlike a synchronized block we will never hold on to any lock;
- because accessing a volatile variable **never holds a lock**, it is **not suitable** for cases where we want to ***read-update-write*** as an atomic operation (unless we're prepared to "miss an update");
- a volatile variable that is an object reference may be null (because you're effectively synchronizing on the *reference*, not the actual object).

Attempting to synchronize on a null object will throw a NullPointerException.

Volatile Keyword

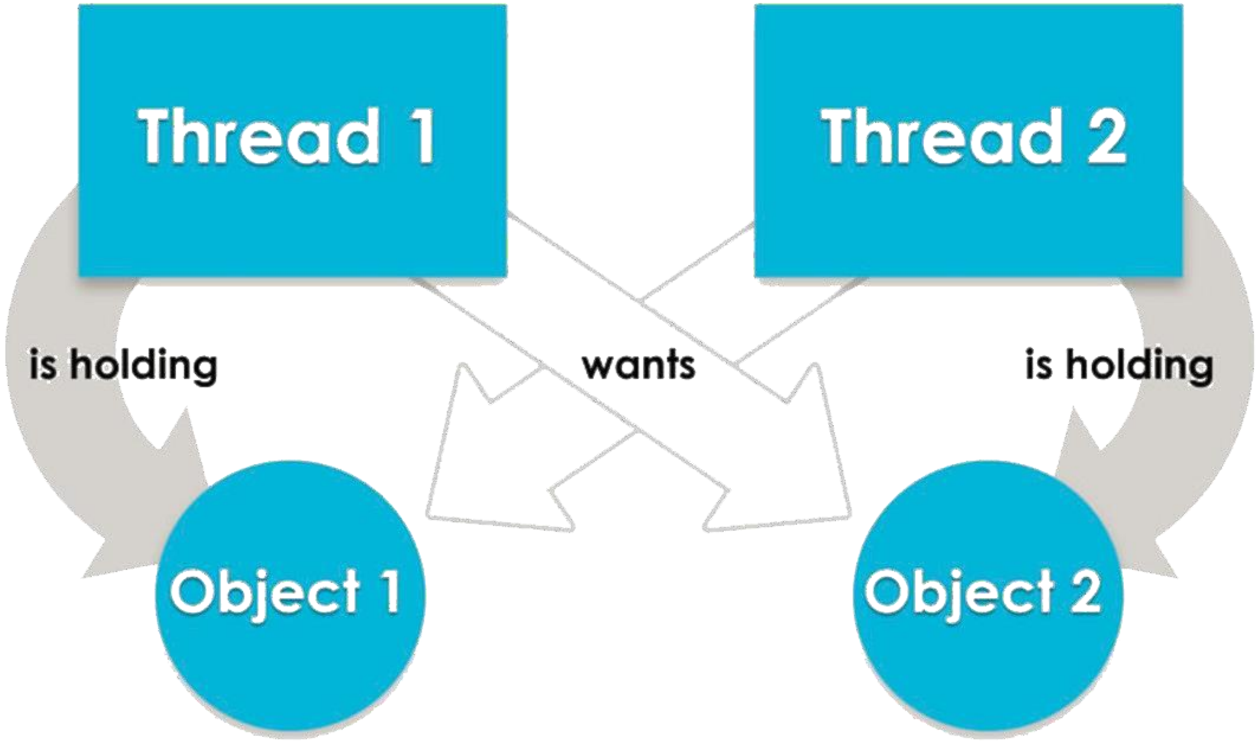
```
public class MyTask {
 private volatile static long A = 0;        // Shared Resource

 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 1000000000; i++) { A++; }
   System.out.println("finish " + Thread.currentThread().getName());
  }  }

 public void show() {
  MyThread t1 = new MyThread(); MyThread t2 = new MyThread();
  MyThread t3 = new MyThread();
  System.out.println(LocalDateTime.now());
  t1.start(); t2.start(); t3.start();
  try { t1.join(); t2.join(); t3.join(); } catch (InterruptedException e) { }
  System.out.println(LocalDateTime.now());
  System.out.println("A=" + A);
 }
}
```

Expected result:
3 000 000 000

Obtained result with 4-core PC

```
15:12:55
finish  Thread-1
finish  Thread-0
finish Thread-2
15:13:43

A=1 268 491 307
```

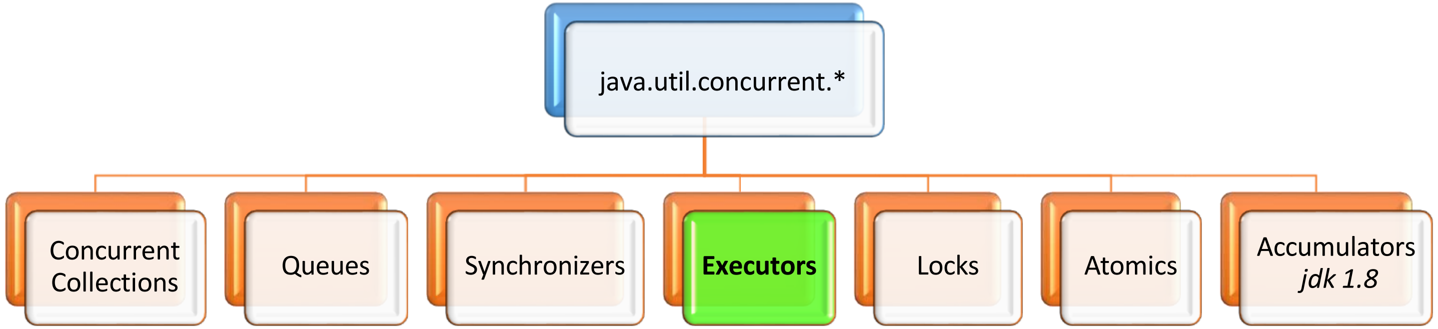
Thread Deadlock



Java Concurrency package

since JDK 1.5

Enhancements have followed in Java 6, Java 7, and Java 8



Concurrent Collections work efficiently in a multithreaded environment.

Queues are non-blocking and blocking queues that support multithreading.

Synchronizers are utility tools for synchronizing threads.

Executors contains frames for creating Thread Pools, work schedule of asynchronous tasks with obtaining results.

Locks introduces alternative and more flexible thread synchronization mechanisms.

Atomics are classes with support for atomic operations on primitives and references.

Accumulators allow for primitive operations (like sum or finding the maximum value) with the numerical elements in a multithreaded environment without CAS (Compare And Swap) using.

Executors Framework

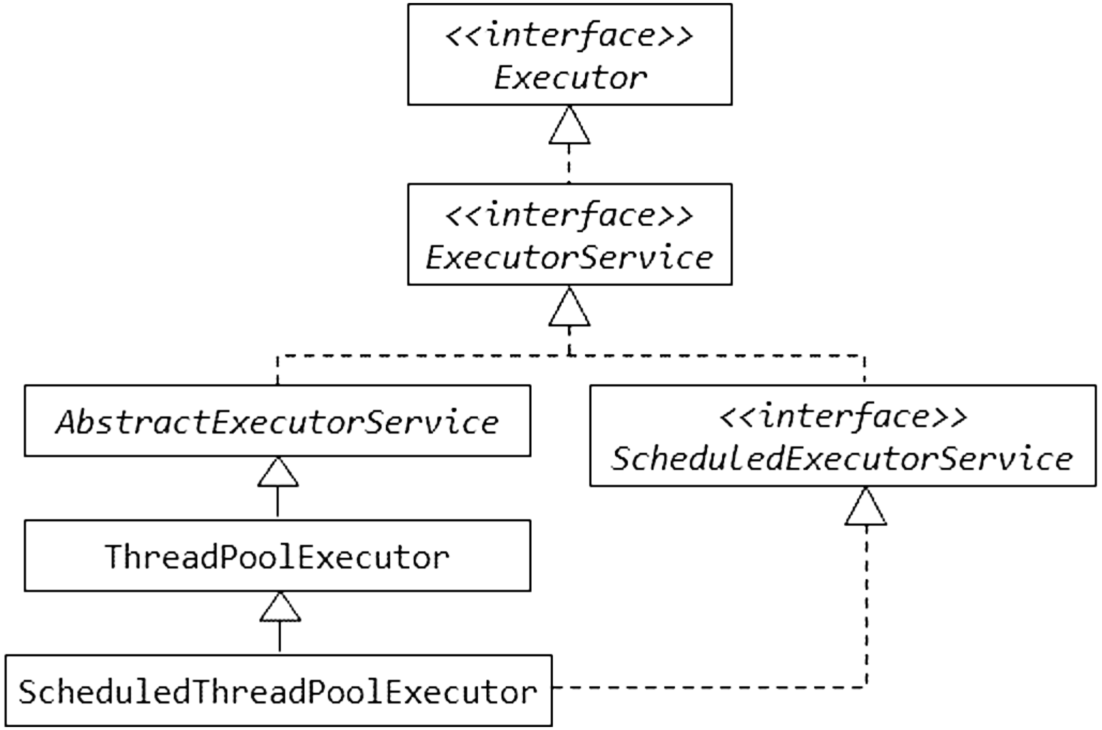
Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads:

- ❖ **Executor** – A simple interface that contains a method called execute() to launch a task specified by a Runnable object.

- ❖ **ExecutorService** – A sub-interface of Executor that adds functionality to manage the lifecycle of the tasks. It also provides a submit() method whose overloaded versions can accept a Runnable as well as a Callable object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value.

- ❖ **ScheduledExecutorService** – A sub-interface of ExecutorService. It adds functionality to schedule the execution of the tasks.

Executors Framework



Executor Interface

```
public interface Executor {
   void execute(Runnable command);
}
```

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

  Runnable myRunnable = () -> System.out.println("Hello, my name is " +
                                                   Thread.currentThread().getName());

  Executor executor = Executors.newSingleThreadExecutor();
  executor.execute(myRunnable);

}
```

```
Hello, my name is pool-1-thread-1
```

Callable Interface

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

A Callable is similar to a Runnable. However, Callable provides a way to return a result or Exception to the thread that spin this Callable. Callable declares an abstract method call() (instead of run() in the Runnable).

Future Interface

```
public interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning);          // for the cancellation of task execution
   boolean isCancelled();
   boolean isDone();                                        // return true if this task completed

   V get() throws InterruptedException, ExecutionException;     // wait if necessary, retrieve result
   V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,
                                                                 TimeoutException;
}
```

Simply put, the Future class represents a future result of an asynchronous computation – a result that will eventually appear in the Future after the processing is complete.

Some examples of operations that would leverage the async nature of Future are:

❖ computational intensive processes (mathematical and scientific calculations);

❖ manipulating large data structures (big data);

❖ remote method calls (downloading files, HTML scrapping, web services).

ExecutorService Interface

- **execute(Runnable)**
- **submit(Runnable)** – submit or schedule the *runnable task* for execution, which returns a *Future* object.
- **submit(Callable)** – submit or schedule the *callable task* for execution, which returns a *Future* object.
- **invokeAny(...)** – this method blocks until the first *callable* terminates and returns the result of that *callable*.
- **invokeAll(...) –** accepts a collection of *callables* and returns a list of *futures*.
- **shutdown()** – when method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
- **shutdownNow()** – this method interrupts the running task and shuts down the executor immediately.

ExecutorService Interface

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

  ExecutorService executorService = Executors.newSingleThreadExecutor();
  executorService.submit(() -> {
    for (int i=1; i<=3; i++){
      System.out.println(i + " Hello1 " + Thread.currentThread().getName());
    }
  });

  executorService.submit(() -> {
    for (int i=1; i<=3; i++){
      System.out.println(i + " Hello2 " + Thread.currentThread().getName());
    }
  });

  executorService.shutdown();
}
```

```
1 Hello1 pool-1-thread-1
2 Hello1 pool-1-thread-1
3 Hello1 pool-1-thread-1
4 Hello2 pool-1-thread-1
5 Hello2 pool-1-thread-1
6 Hello2 pool-1-thread-1
```

ExecutorService Interface

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

  ExecutorService executorService = Executors.newSingleThreadExecutor();
  executorService.submit(() -> {
    for (int i=1; i<=3000000; i++){
      System.out.println(i + " Hello1 " + Thread.currentThread().getName());
    }
  });

  executorService.submit(() -> {
    for (int i=1; i<=3; i++){
      System.out.println(i + " Hello2 " + Thread.currentThread().getName());
    }
  });

  executorService.shutdownNow();
}
```

```
1 Hello1 pool-1-thread-1
2 Hello1 pool-1-thread-1
…
2999999 Hello1 pool-1-thread-1
3000000 Hello1 pool-1-thread-1
```

ExecutorService Interface

```
public static void main(String[] args) {
  ExecutorService executorService = Executors.newSingleThreadExecutor();
  Future<Integer> futureA = executorService.submit(() -> {
    for (long i=1; i<=3000000000L; i++){}
    System.out.println(" Hello1 " + Thread.currentThread().getName());
    return 125;
  });
  Future<Integer> futureB = executorService.submit(() -> {
    for (long i=1; i<=3000000000L; i++){}
    System.out.println(" Hello2 " + Thread.currentThread().getName());
    return 250;
  });

  try {
    System.out.println("futureA = " + futureA.get());
    System.out.println("futureB = " + futureB.get());
  } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
  executorService.shutdown();
}
```

```
 Hello1 pool-1-thread-1
futureA = 125
 Hello2 pool-1-thread-1
futureB = 250
```

ExecutorService Interface

```
public static void main(String[] args) throws InterruptedException {
  ExecutorService executor = Executors.newWorkStealingPool();
  List<Callable<String>> callables = Arrays.asList(() -> "task1", () -> "task2", () -> "task3");
  executor.invokeAll(callables)
    .stream().map(future -> {
       try {
         return future.get();
       } catch (Exception e) { throw new IllegalStateException(e); }
    })
    .forEach(System.out::println);
}
```

```
task1
task2
task3
```

Executors support batch submitting of multiple callables at once via invokeAll(). This method accepts a collection of callables and returns a list of futures.

ExecutorService Interface

```
static Callable<String> callable(String result, long sleepSeconds) {
 return () -> {
  TimeUnit.SECONDS.sleep(sleepSeconds);
  return result;
 } };

public static void main(String[] args) throws InterruptedException, ExecutionException {
 ExecutorService executor = Executors.newWorkStealingPool();
 List<Callable<String>> callables =
    Arrays.asList( callable("task1", 2),   callable("task2", 1),   callable("task3", 3) );
 String result = executor.invokeAny(callables);
 System.out.println(result);
}
```

```
task2
```

Instead of returning future objects this method blocks until the first callable terminates and returns the result of that callable.

ScheduledFuture Interface

```
public interface ScheduledFuture<V> extends Delayed, Future<V> {
}

public interface Delayed extends Comparable<Delayed> {
   long getDelay(TimeUnit unit);
}
```

Scheduling a task produces a specialized future of type *ScheduledFuture* which – in addition to *Future* – provides the method getDelay() to retrieve the remaining delay. After this delay has elapsed the task will be executed concurrently.

ScheduledExecutorService Interface

```
public interface ScheduledExecutorService extends ExecutorService {

    // Creates and executes a one-shot action that becomes enabled after the given delay
    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

    // Creates and executes a ScheduledFuture that becomes enabled after the given delay
    public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

   // Creates and executes a periodic action that becomes enabled first after the given initial
   // delay, and subsequently with the given period
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                long initialDelay, long period, TimeUnit unit);

    // Creates and executes a periodic action that becomes enabled first after
   // the given initial delay, and subsequently with the given delay between the executions
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                  long initialDelay, long delay, TimeUnit unit);

}
```

ScheduledExecutorService Interface

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

  ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

  Runnable task = () -> System.out.println(" Scheduling: " + System.nanoTime());
  ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

  TimeUnit.MILLISECONDS.sleep(1337);

  long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
  System.out.printf("Remaining Delay: %sms", remainingDelay);
}
```

```
Remaining Delay: 1668ms
Scheduling: 542008455362334
```

A ScheduledExecutorService is capable of scheduling tasks to run either periodically or once after a certain amount of time has elapsed.

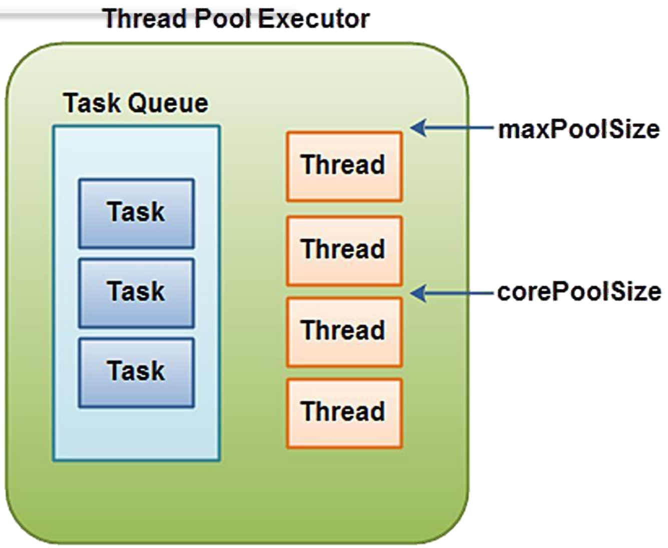
This code sample schedules a task to run after an initial delay of three seconds has passed:

ScheduledExecutorService Interface

```
public class MyTask {
 static Callable<String> callable(String result, long sleepSeconds) {
  return () -> {
   TimeUnit.SECONDS.sleep(sleepSeconds);
   return result;
  } };

 public static void main(String[] args) throws InterruptedException, ExecutionException {
  ScheduledExecutorService executorService =
                                            Executors.newSingleThreadScheduledExecutor();
  Runnable callableTask = () -> System.out.println(LocalDateTime.now());
  executorService.scheduleAtFixedRate(callableTask, 1,1, TimeUnit.SECONDS);
 }
}
```

```
2018-05-04T 15:39:49.83
2018-05-04T 15:39:50.79
2018-05-04T 15:39:51.79
2018-05-04T 15:39:52.78
2018-05-04T 15:39:53.79
2018-05-04T 15:39:54.79
...
```

ThreadPoolExecutor



```
int  corePoolSize  =    5;
int  maxPoolSize    =   10;
long keepAliveTime = 5000;
ExecutorService threadPoolExecutor =
   new ThreadPoolExecutor( corePoolSize, maxPoolSize, keepAliveTime,
      TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>() );
```

How to Create an Executor

To create an Executor it is possible to use the factory **Executors** class.

Most common methods are used to create:

- an **ExecutorService** with a single thread to execute commands with method **newSingleThreadExecutor**.
- a **ScheduledExecutorService** with a single thread to execute commands with the method newSingleThreadScheduledExecutor.
- an **ExecutorService** that use a fixed length pool of threads to execute commands with the method **newFixedThreadPool**.
- an **ExecutorService** with a pool of threads that creates a new thread if no thread is available and reuse an existing thread if they are available with **newCachedThreadPool**.
- a **ScheduledExecutorService** with a fixed length pool of threads to execute scheduled commands with the method **newScheduledThreadPool**.

How to Create an Executor

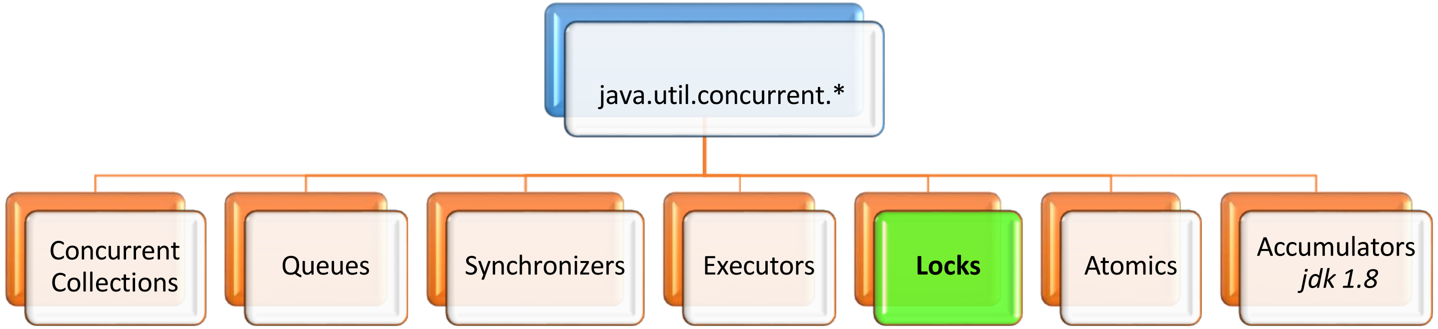
Here are examples to creates ExecutorService and ScheduledExecutorService instances:

```
// Creates a single thread ExecutorService
ExecutorService singleExecutorService = Executors.newSingleThreadExecutor();

// Creates a single thread ScheduledExecutorService
ScheduledExecutorService singleScheduledExecutorService =
                                        Executors.newSingleThreadScheduledExecutor();

// Creates an ExecutorService that use a pool of 10 threads
ExecutorService fixedExecutorService = Executors.newFixedThreadPool(10);

// Creates an ExecutorService that use a pool that creates threads on demand
// And that kill them after 60 seconds if they are not used
ExecutorService onDemandExecutorService = Executors.newCachedThreadPool();

// Creates a ScheduledExecutorService that use a pool of 5 threads
ScheduledExecutorService fixedScheduledExecutorService =
                                        Executors.newScheduledThreadPool(5);
```

Java Concurrency package

since JDK 1.5

Enhancements have followed in Java 6, Java 7, and Java 8



Concurrent Collections work efficiently in a multithreaded environment.

Queues are non-blocking and blocking queues that support multithreading.

Synchronizers are utility tools for synchronizing threads.

Executors contains frames for creating Thread Pools, work schedule of asynchronous tasks with obtaining results.

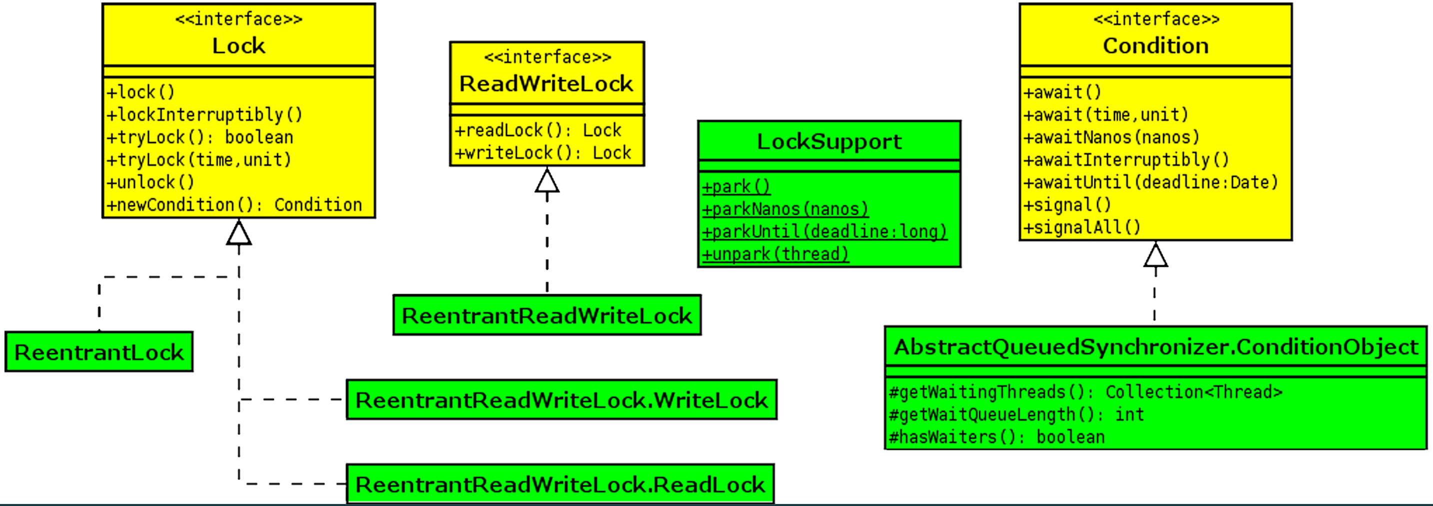
Locks introduces alternative and more flexible thread synchronization mechanisms.

Atomics are classes with support for atomic operations on primitives and references.

Accumulators allow for primitive operations (like sum or finding the maximum value) with the numerical elements in a multithreaded environment without CAS (Compare And Swap) using.

java.util.concurrent.locks;

- **Lock** – It provides all the features of synchronized keyword with additional ways to create different Conditions for locking, providing timeout for thread to wait for lock.
- **ReadWriteLock** – It contains a pair of associated locks, one for read-only operations and another one for writing.
- **Condition** – Condition objects are similar to Object wait-notify model with additional feature to create different sets of wait.



Lock interface API

- void **lock**() – acquire the lock if it's available; if the lock is not available a thread gets blocked until the lock is released

- void **lockInterruptibly**() – this is similar to the lock(), but it allows the blocked thread to be interrupted and resume the execution through a thrown java.lang.InterruptedException

- boolean **tryLock**() – this is a non-blocking version of lock() method; it attempts to acquire the lock immediately, return true if locking succeeds

- boolean **tryLock**(long timeout, TimeUnit timeUnit) – this is similar to tryLock(), except it waits up the given timeout before giving up trying to acquire the Lock

- void **unlock**() – unlocks the Lock instance

ReentrantLock – Example1

```
public class MyLocks {
 private final static ReentrantLock lock = new ReentrantLock();
 private static long A = 0;                    // Shared Resource
 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 1000000000; i++) { lock.lock();
    try { A++; }  finally { lock.unlock(); }
   }
   System.out.println("finish " + Thread.currentThread().getName());
  } }

 public void show() {
  MyThread t1 = new MyThread(); MyThread t2 = new MyThread();
  MyThread t3 = new MyThread();
  t1.start(); t2.start();  t3.start();
  try { t1.join();  t2.join();  t3.join(); } catch (InterruptedException e) {   }
  System.out.println("A=" + A);
 }
}
```

Expected result:
3 000 000 000

Obtained result with 4-core PC

```
13:49:34.27

finish Thread-2
finish Thread-1
finish Thread-0
A=3 000 000 000

13:50:55.50
```

ReentrantLock – Example2

```
public class MyLocks2 {
 private final static ReentrantLock lock = new ReentrantLock();
 private static long A = 0;                 // Shared Resource
 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 100000000; i++) {
    try {
     if (lock.tryLock(5, TimeUnit.SECONDS)) {
      try { A++;  lock.lock(); }  finally { lock.unlock();   /*   ???   */ }
     } else {
      System.out.println("Sorry " + Thread.currentThread().getName()); break;
     }
    } catch (InterruptedException e) { }
   }
   System.out.println(LocalDateTime.now() + "  finish " + Thread.currentThread().getName());
  }
 }
 public void show() {  /* as in Example 1  */   }       }
```

Expected result:
300 000 000

Obtained result
with 4-core PC

```
14:20:46.335
14:20:49.894  finish Thread-1
Sorry Thread-2
Sorry Thread-0
14:20:51.411  finish Thread-0
14:20:51.411  finish Thread-2
A=100 000 000
```

ReentrantLock – Example3

```
public class MyLocks3 {
  private final static ReentrantLock lock = new ReentrantLock();
  private static long A = 0;                // Shared Resource
  class MyThread extends Thread {
    @Override
    public void run() {
        work();
        finish();
    }
    private void work() {
      lock.lock();
      for (int i = 1; i <= 1000000000; i++) {  A++;  }
    }
    private void finish() {
      lock.unlock();
      System.out.println("finish " + Thread.currentThread().getName());
    } }

  public void show() {  /* as in Example 1  */    }      }
```

Expected result:
3 000 000 000

Obtained result with 4-core PC

```
15:03:58.019

finish Thread-0
finish Thread-1
finish Thread-2
A=3 000 000 000

15:03:58.344
```

Lock Reentrance – Example 4

```
public class Reentrant {
  private final static Lock lock = new ReentrantLock();

  public static void outer() {
    lock.lock();
    inner();
    lock.unlock();
  }
  public static void inner() {
    lock.lock();
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    System.out.println(Thread.currentThread().getName() + LocalDateTime.now());
    lock.unlock();
  }

  public static void main(String[] args) {
    System.out.println("start " + LocalDateTime.now());
    Thread thread1 = new Thread(() -> outer(), "T1 ");
    Thread thread2 = new Thread(() -> outer(), "T2 ");
    thread1.start();      thread2.start();
  }   }
```

```
start 15:50:21.031
T1    15:50:22.100
T2    15:50:23.100
```

ReadWriteLock interface API

- Lock **readLock**() – returns the lock that’s used for reading
- Lock **writeLock**() – returns the lock that’s used for writing

Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping.

But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock.

ReentrantReadWriteLock – Example 1

```java
public class MyReadWriteLock {
  private static ReadWriteLock  rwLock = new ReentrantReadWriteLock();
  private static Lock readLock = rwLock.readLock();
  private static Lock writeLock = rwLock.writeLock();
  private static long  A = 0;                // Shared Resource

  class MyThread implements Runnable {
    @Override
    public void run() {
      readLock.lock();
      try {
        System.out.println(LocalDateTime.now() + Thread.currentThread().getName() + " A=" + A);
        Thread.sleep(2000);
      } catch (InterruptedException e) {
      } finally { readLock.unlock(); }
    }
  }
```

continue ↓

ReentrantReadWriteLock – Example 1

previous ↑

```
public void show() {
  ExecutorService executor = Executors.newFixedThreadPool(4);
  Runnable rT1 = new MyThread();   Runnable rT2 = new MyThread();
  Runnable rT3 = new MyThread();
  executor.submit(rT1);   executor.submit(rT2);   executor.submit(rT3);
  executor.submit( () -> {  writeLock.lock();
  try {
     A = 25;
     System.out.println(LocalDateTime.now() + Thread.currentThread().getName() + " A=" + A);
     Thread.sleep(5000);
   } catch (InterruptedException e) { } finally {  writeLock.unlock(); }   });
  executor.submit(rT1);   executor.submit(rT2);   executor.submit(rT3);
 }
}
```

```
13:54:40   start
13:54:40   pool-1-thread-2    A=0
13:54:40   pool-1-thread-3    A=0
13:54:40   pool-1-thread-1  A=0
13:54:42   pool-1-thread-4    A=25
13:54:47   pool-1-thread-1    A=25
13:54:47   pool-1-thread-3    A=25
13:54:47   pool-1-thread-2    A=25
```

Lock Reentrance – Example 2

```
public class MyReadWriteLockReentrant {
  private static ReadWriteLock rwLock = new ReentrantReadWriteLock();
  private static Lock readLock = rwLock.readLock();
  private static Lock writeLock = rwLock.writeLock();
  public static void outer() {
    readLock.lock();
    System.out.println(Thread.currentThread().getName() + "outer: " + LocalDateTime.now());
    inner();
    readLock.unlock();
    }
    public static void inner() {
    readLock.lock();
    System.out.println(Thread.currentThread().getName() + " inner: " + LocalDateTime.now());
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    readLock.unlock();
  }
  public static void main(String[] args) {
    Thread thread1 = new Thread(() -> outer(), "T1");
    Thread thread2 = new Thread(() -> outer(), "T2");
    thread1.start();  thread2.start();    }   }
```

```
start        16:30:56
T1 outer: 16:30:56
T1 inner: 16:30:56
T2 outer: 16:30:56
T2 inner: 16:30:56
```

Lock Reentrance – Example 3

```
public class MyReadWriteLockReentrant {
 private static ReadWriteLock rwLock = new ReentrantReadWriteLock();
 private static Lock readLock = rwLock.readLock();
 private static Lock writeLock = rwLock.writeLock();
 public static void outer() {
   writeLock.lock();
   System.out.println(Thread.currentThread().getName() + "outer: " + LocalDateTime.now());
   inner();
   writeLock.unlock();
   }
   public static void inner() {
   writeLock.lock();
   System.out.println(Thread.currentThread().getName() + " inner: " + LocalDateTime.now());
   try { Thread.sleep(1000); } catch (InterruptedException e) { }
   writeLock.unlock();
 }
 public static void main(String[] args) {
   Thread thread1 = new Thread(() -> outer(), "T1");
   Thread thread2 = new Thread(() -> outer(), "T2");
   thread1.start();  thread2.start();    }  }
```

```
start          16:33:53
T1 outer: 16:33:53
T1 inner: 16:33:53
T2 outer: 16:33:54
T2 inner: 16:33:54
```

Lock Reentrance – Example 4

```
public class MyReadWriteLockReentrant {
  private static ReadWriteLock rwLock = new ReentrantReadWriteLock();
  private static Lock readLock = rwLock.readLock();
  private static Lock writeLock = rwLock.writeLock();
  public static void outer() {
    readLock.lock();
    System.out.println(Thread.currentThread().getName() + "outer: " + LocalDateTime.now());
    inner();
    readLock.unlock();
    }
    public static void inner() {
    writeLock.lock();
    System.out.println(Thread.currentThread().getName() + " inner: " + LocalDateTime.now());
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    writeLock.unlock();
  }
  public static void main(String[] args) {
    Thread thread1 = new Thread(() -> outer(), "T1");
    Thread thread2 = new Thread(() -> outer(), "T2");
    thread1.start();  thread2.start();    }  }
```

```
start        16:36:20
T1   outer16:36:20
T2 outer: 16:36:20
```

Lock Reentrance – Example 5

```
public class MyReadWriteLockReentrant {
 private static ReadWriteLock rwLock = new ReentrantReadWriteLock();
 private static Lock readLock = rwLock.readLock();
 private static Lock writeLock = rwLock.writeLock();
 public static void outer() {
  writeLock.lock();
  System.out.println(Thread.currentThread().getName() + "outer: " + LocalDateTime.now());
  inner();
  writeLock.unlock();
  }
  public static void inner() {
  readLock.lock();
  System.out.println(Thread.currentThread().getName() + " inner: " + LocalDateTime.now());
  try { Thread.sleep(1000); } catch (InterruptedException e) { }
  readLock.unlock();
 }
 public static void main(String[] args) {
  Thread thread1 = new Thread(() -> outer(), "T1");
  Thread thread2 = new Thread(() -> outer(), "T2");
  thread1.start();  thread2.start();    }  }
```

```
start        16:49:02
T1 outer: 16:49:02
T1 inner: 16:49:02
T2 outer: 16:49:03
T2 inner: 16:49:03
```

Condition interface

Condition objects give wait and notify functionality to explicit locks the same way wait() and notify() work with implicit locks. This gives us a situation where the add threads and the remove threads are in separate wait sets!

- ❖ Condition objects give threads a mechanism to suspend until another thread modifies state so it can resume
- ❖ Condition is just an interface
- ❖ Classes that implement conditions are free to do lots of cool things like specifying a guaranteed order for notifications or make notifications without owning a lock (!)

Condition interface – Example

```
public class ReentrantLockWithCondition {
 static Stack<String> stack = new Stack<>();
 final static int CAPACITY = 2;
 static ReentrantLock lock = new ReentrantLock();
 static Condition stackEmptyCondition = lock.newCondition();
 static Condition stackFullCondition = lock.newCondition();

 public static void pushToStack(String item) {
  try {
   lock.lock();
   while (stack.size() == CAPACITY) {
    System.out.println("Stack is full " + LocalDateTime.now());
    stackFullCondition.await();
   }
   stack.push(item);
   System.out.println("Stack <- " + item + " " + LocalDateTime.now());
   stackEmptyCondition.signalAll();
  } catch (InterruptedException e) { } finally {
   lock.unlock();
  }    }
```

continue ↓

Condition interface – Example

previous ↑

```
public static String popFromStack() {
 String result = "";
 try {
  lock.lock();
  while (stack.size() == 0) {
   System.out.println("Stack is empty " + LocalDateTime.now());
   stackEmptyCondition.await();
  }
  result = stack.pop();
 } catch (InterruptedException e) { } finally {
  stackFullCondition.signalAll();
  lock.unlock();
 }
 System.out.println("Stack -> " + result + " " + LocalDateTime.now());
 return result;
}
```

continue ↓

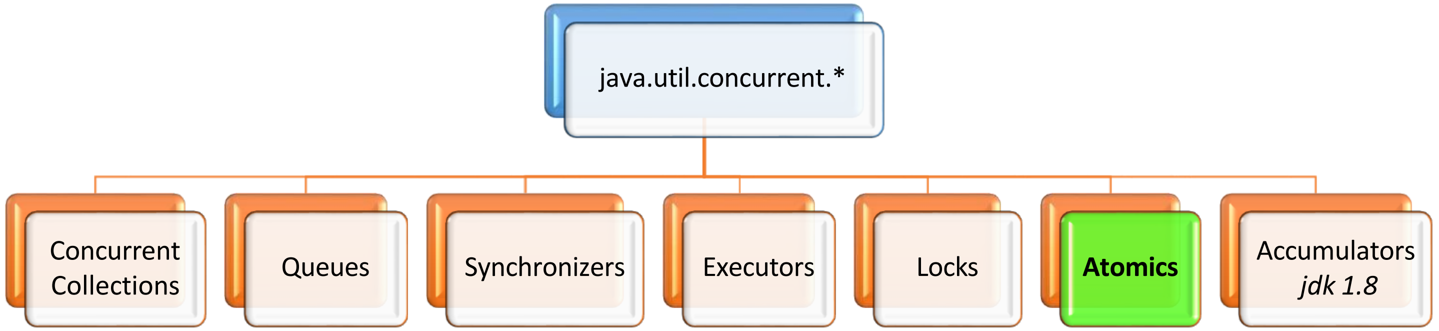
Condition interface – Example

previous ↑

```
public static void main(String[] args) {
  ExecutorService executor = Executors.newFixedThreadPool(2);

  executor.submit(() -> {
    pushToStack("value=1");
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    pushToStack("value=2");
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    pushToStack("value=3");
    try { Thread.sleep(2000); } catch (InterruptedException e) { }
    pushToStack("value=4");    });

  executor.submit(() -> {
    try { Thread.sleep(3000); } catch (InterruptedException e) { }
    popFromStack();
    popFromStack();
    popFromStack();
    popFromStack();     });
}  }
```

```
Stack <- value=1   09:55:23
Stack <- value=2   09:55:24
Stack is full      09:55:25
Stack -> value=2   09:55:26
Stack <- value=3   09:55:26
Stack -> value=3   09:55:26
Stack -> value=1   09:55:26
Stack is empty     09:55:26
Stack <- value=4   09:55:28
Stack -> value=4   09:55:28
```

Java Concurrency package



Atomic Classes

The package java.util.concurrent.atomic has a bunch of classes to make accessing single variables thread-safe without using locks. (Native methods are used instead.)

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference<V>
- AtomicMarkableReference<V>
- AtomicStampedReference<V>
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray<E>
- AtomicIntegerFieldUpdater<T>
- AtomicLongFieldUpdater<T>
- AtomicReferenceFieldUpdater<T,V>

Atomic Classes

- ❖ The basic idea is that an atomic object is like a volatile field plus an atomic compareAndSet operation.
- ❖ The numeric classes also have atomic **incrementAndGet**, **decrementAndGet**, **addAndGet**, **getAndIncrement**, **getAndDecrement**, **getAndAdd**.
- ❖ These classes are provided as building blocks for a non-blocking data structures and other higher level entities. You would rarely use them on their own.

  One exception is a sequence generator:

```
public class Sequencer {
  private AtomicLong sequenceNumber = new AtomicLong(0);
  public long next() { return sequenceNumber.getAndIncrement(); }
}
```

AtomicLong Class

long **addAndGet**(long delta) – Atomically adds the given value to the current value

boolean **compareAndSet**(long expect, long update) – Atomically sets the value to the given updated value if the current value is same as the expected value

long **decrementAndGet**() – Atomically decrements by one the current value

long **get**() – Gets the current value

long **getAndAdd**(long delta) – Atomiclly adds the given value to the current value

long **getAndDecrement**() – Atomically decrements by one the current value

long **getAndIncrement**() – Atomically increments by one the current value

long **getAndSet**(long newValue) – Atomically sets to the given value and returns the old value

long **incrementAndGet**() – Atomically increments by one the current value

void **lazySet**(long newValue) – Eventually sets to the given value

void **set**(long newValue) – Sets to the given value

boolean **weakCompareAndSet**(long expect, long update) – Atomically sets the value to the given updated value if the current value is same as the expected value

AtomicLong Class – Example

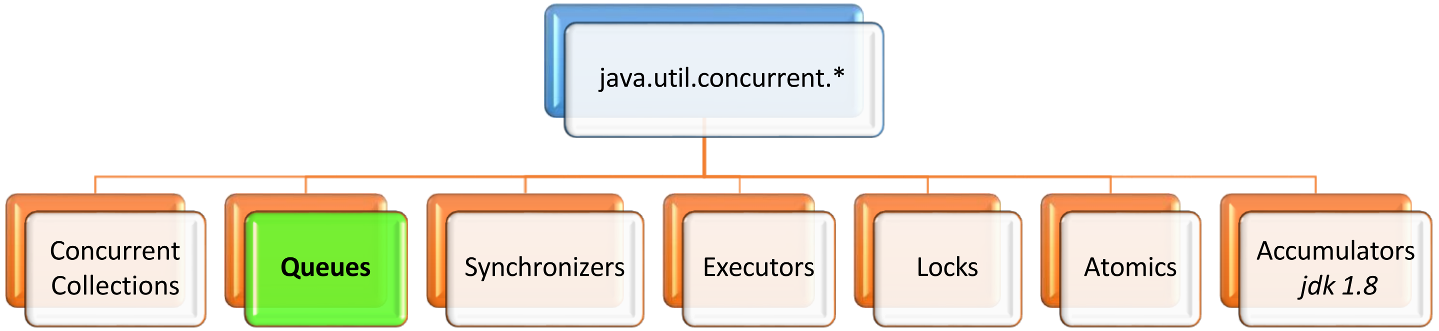
```
public class MyAtomicLong {
 AtomicLong atomicLong= new AtomicLong(0); // Shared Resource

 class MyThread extends Thread {
  @Override
  public void run() {
   for (int i = 1; i <= 1000000000; i++) {
    atomicLong.incrementAndGet();
   }
   System.out.println("finish " + Thread.currentThread().getName());
  }   }

 public void show() {
  MyThread t1 = new MyThread(); MyThread t2 = new MyThread();
  MyThread t3 = new MyThread();
  t1.start();   t2.start();   t3.start();
  try {    t1.join(); t2.join(); t3.join(); } catch (InterruptedException e) { }
  System.out.println("value=" + atomicLong.get());
 }   }
```

Expected result:
3 000 000 000

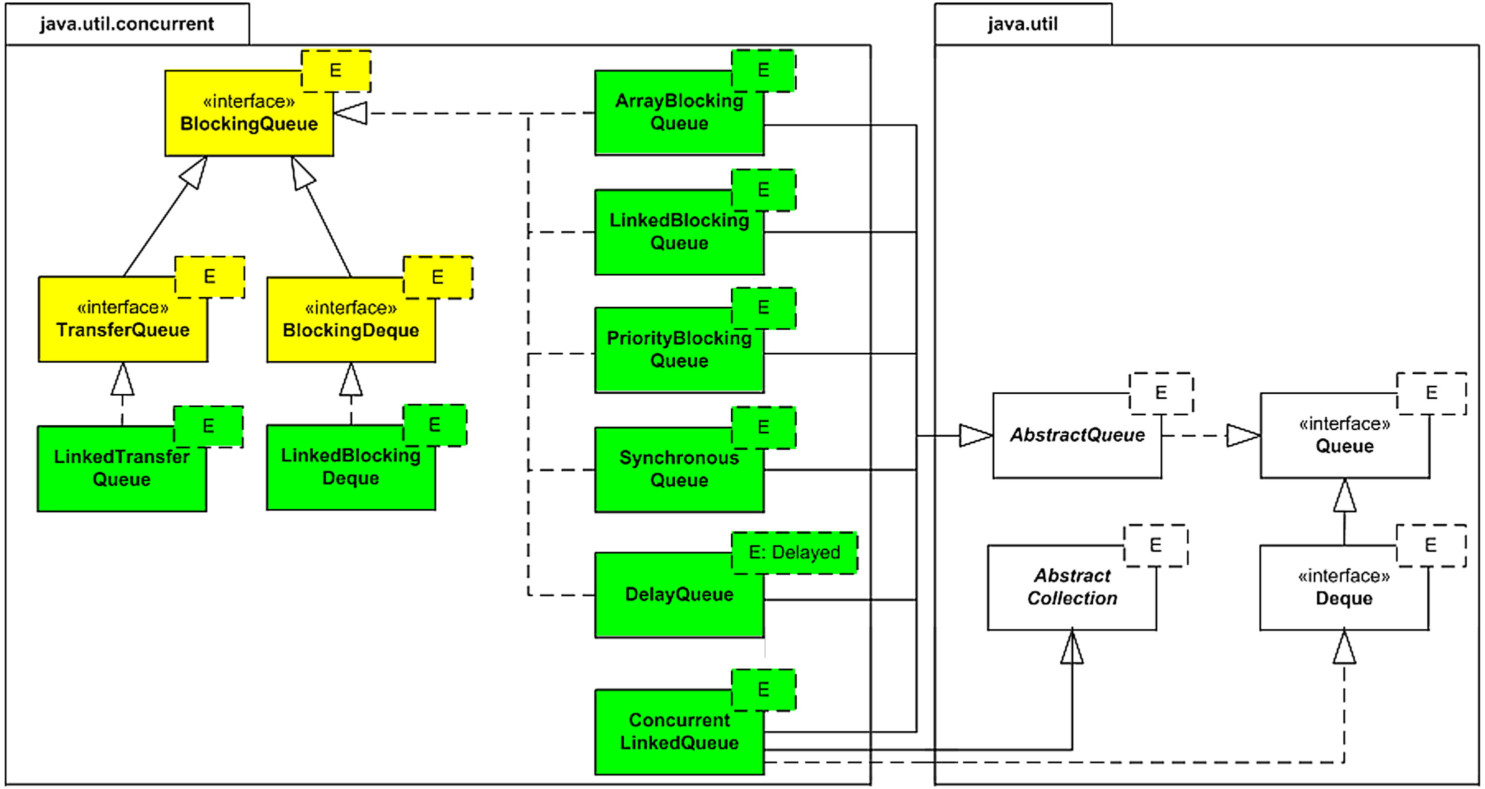
Obtained result with 4-core PC

```
10:56:15

finish  Thread-1
finish  Thread-2
finish Thread-0
value=3 000 000 000

10:56:50
```

Java Concurrency package



java.util.concurrent.BlockingQueue



BlockingQueue<E> Interface

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()		

Adding Elements

add() – returns true if insertion was successful, otherwise throws an IllegalStateException
put() – inserts the specified element into a queue, waiting for a free slot if necessary
offer() – returns true if insertion was successful, otherwise false
offer(E e, long timeout, TimeUnit unit) – tries to insert element into a queue and waits for an available slot within a specified timeout

Retrieving Elements

take() – waits for a head element of a queue and removes it. If the queue is empty, it blocks and waits for an element to become available
poll(long timeout, TimeUnit unit) – retrieves and removes the head of the queue, waiting up to the specified wait time if necessary for an element to become available. Returns null after a timeout

BlockingDeque<E> Interface

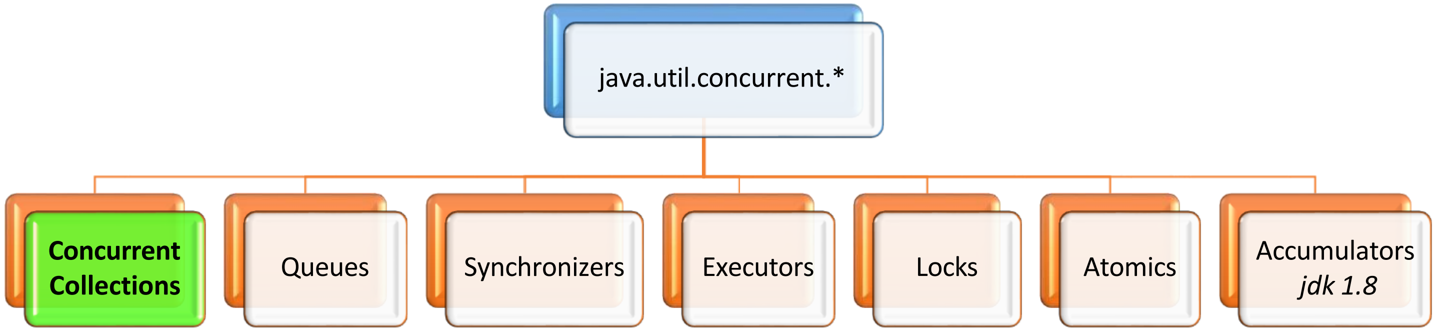
First Element (Head)

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, time, unit)
Remove	removeFirst()	pollFirst()	takeFirst()	pollFirst(time, unit)
Examine	getFirst()	peekFirst()		

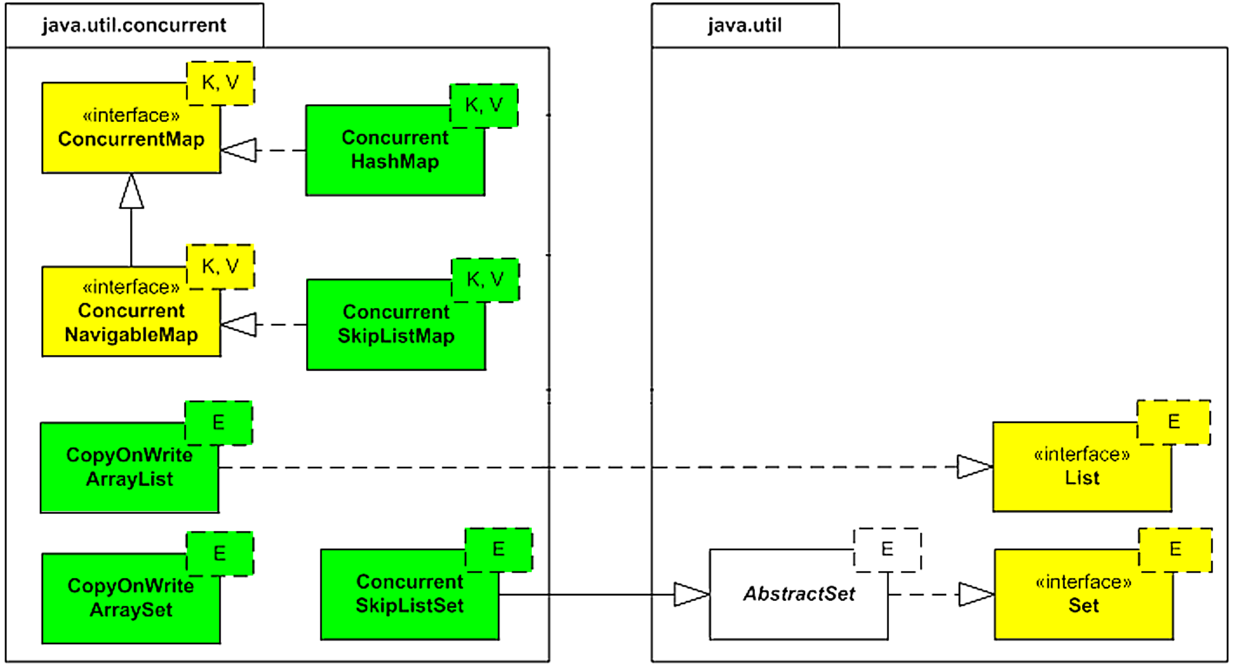
Last Element (Tail)

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, time, unit)
Remove	removeLast()	pollLast()	takeLast()	pollLast(time, unit)
Examine	getLast()	peekLast()		

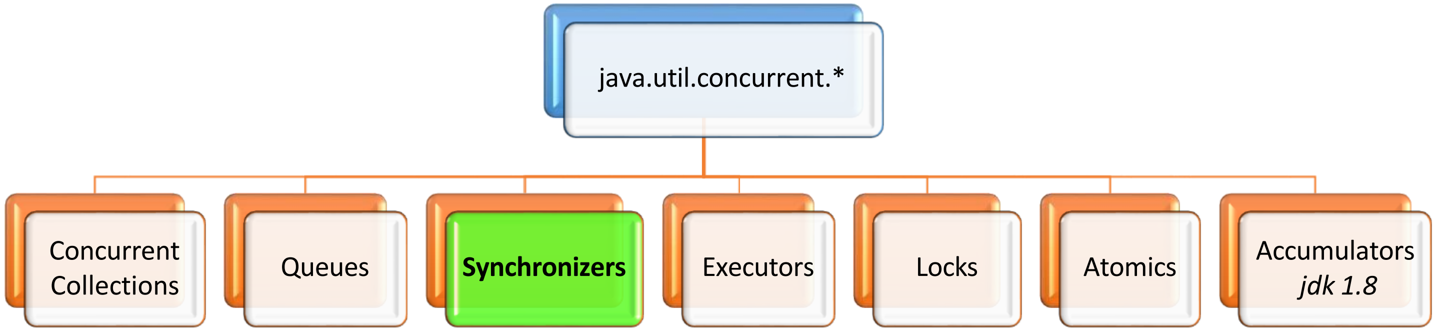
Java Concurrency package



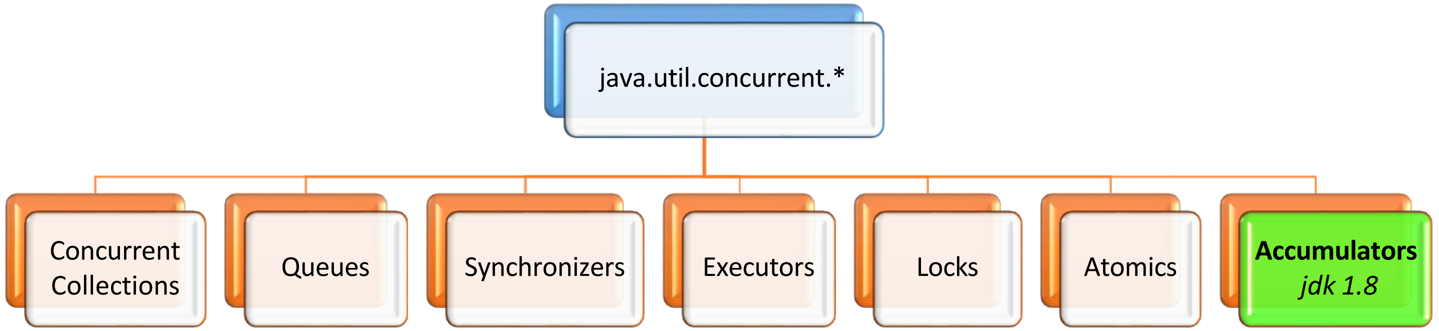
Concurrent Collections



Java Concurrency package



Java Concurrency package



Task

1. Write simple “ping-pong” program using wait() and notify().
2. Create a task that produces a sequence of n Fibonacci numbers, where n is provided to the constructor of the task. Create a number of these tasks and drive them using threads.
3. Repeat previous exercise using the different types of executors.
4. Modify Exercise 2 so that the task is a Callable that sums the values of all the Fibonacci numbers. Create several tasks and display the results.
5. Create a task that sleeps for a random amount of time between 1 and 10 seconds, then displays its sleep time and exits. Create and run a quantity (given on the command line) of these tasks. Do it by using ScheduledThreadPool.
6. Create a class with three methods containing critical sections that all synchronize on the same object. Create multiple tasks to demonstrate that only one of these methods can run at a time. Now modify the methods so that each one synchronizes on a different object and show that all three methods can be running at once.
7. Write program in which two tasks use a pipe to communicate.