Lecture 9
Java SE – Advanced Multithreading / HPC

presentation

Java Programming – Software App Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 9 – JSE Multithreading

java.util.concurrent - Executor & ExecutorService, Callable, Future

Advanced Multi-Threading

1. Summary of Multi-threading in C vs. Java

Multi-threading vs. Multi-process "quite good/imperfect" analogies & terms:

USE MULTI-THREADING for

- PARALLELISM (e.g. adding 2 matrixes in parallel) and/or;
- CONCURENCY (e.g. handling GUI on one thread & business processing on other thread / even on mono-processor PC, handling multiple HTTP requests from the network / cooperation on the same data structure between at least 2 threads – producer and consumer)

Mutexes are used to prevent data inconsistencies due to ***race conditions***.
A ***race condition*** often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.
Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.
One can apply a mutex to protect a segment of memory ("***critical region***") from other threads.
Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores in Linux IPC.

In Java Mutex is quite ⇔ synchronized (also please see java.util.concurrent.*)

1. Threads Issues

Advantages of Multithreading:

- Reactive systems – constantly monitoring
- More responsive to user input – GUI application can interrupt a time-consuming task
- Server can handle multiple clients simultaneously
- Can take advantage of parallel processing

When Multithreading?:

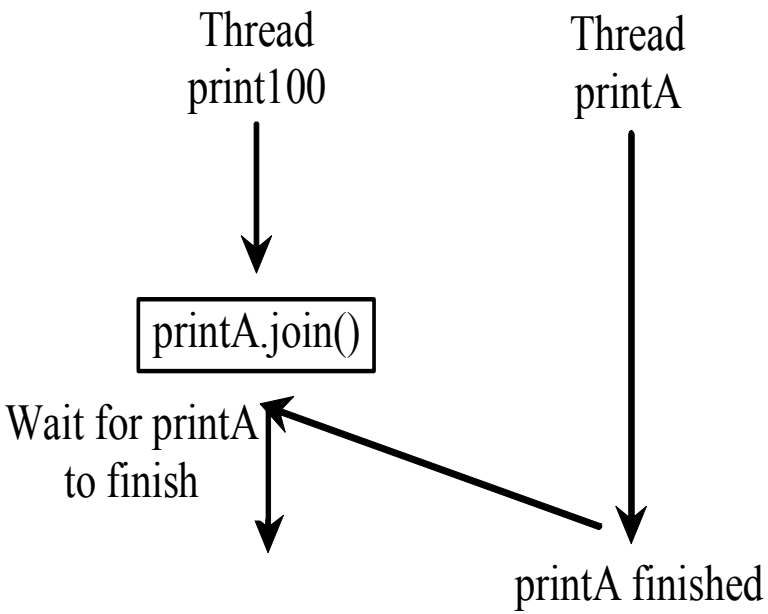
Concurrency (incl. Cooperation)

and/or Parallelism

The join() Method

You can use the join() method to force one thread to wait for another thread to finish.

```
public void run() { //print100 - method
  Thread thread4printA = new Thread(
    new PrintChar('A', 40));
  thread4printA.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4printA.join();
    }
  }
  catch (InterruptedException ex) {
  }
```



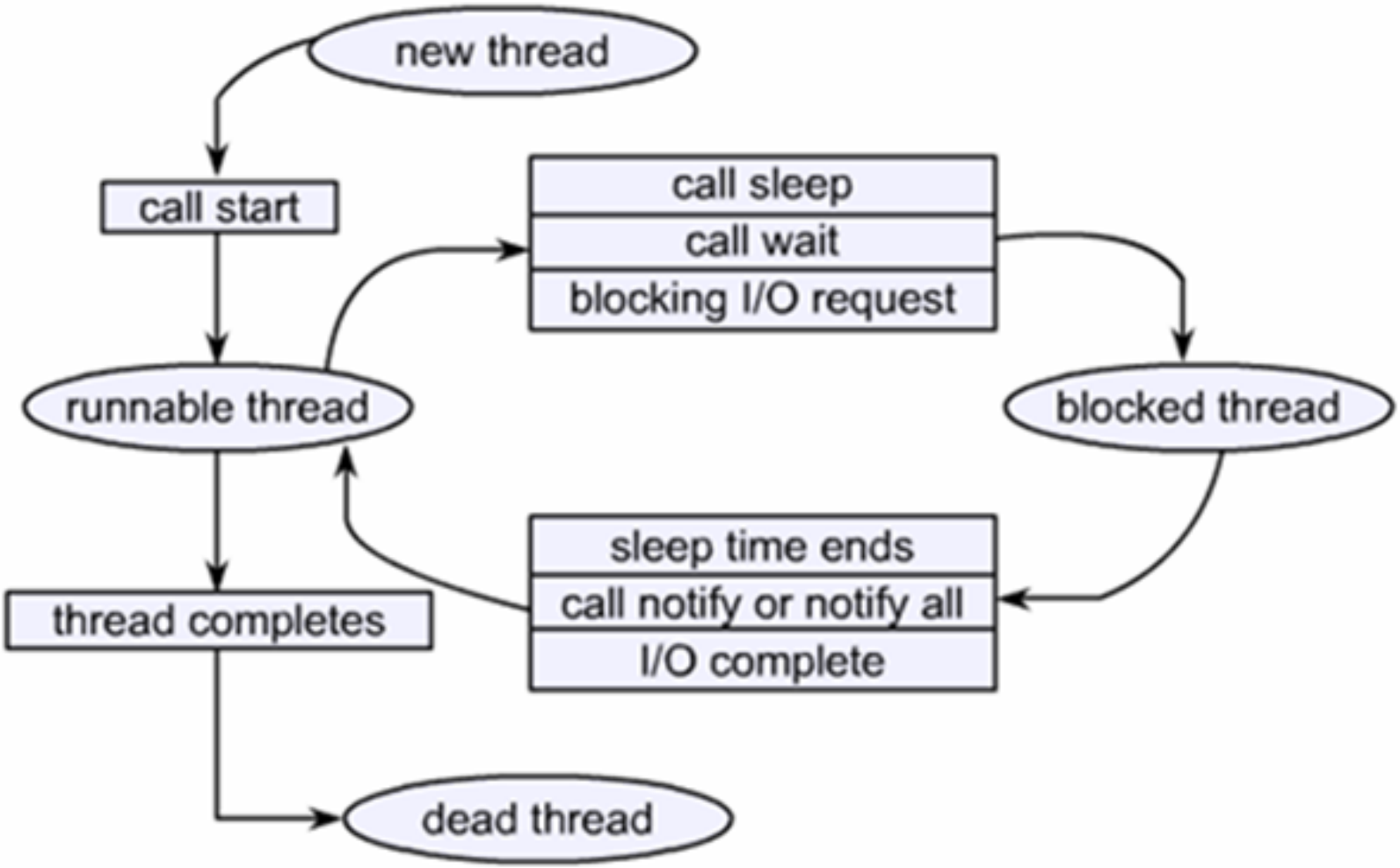
The numbers after 50 are printed after thread printA is finished.

1. Threads Concurrency – Synchronization

Java Multi-threading Cooperation – The *join()* method

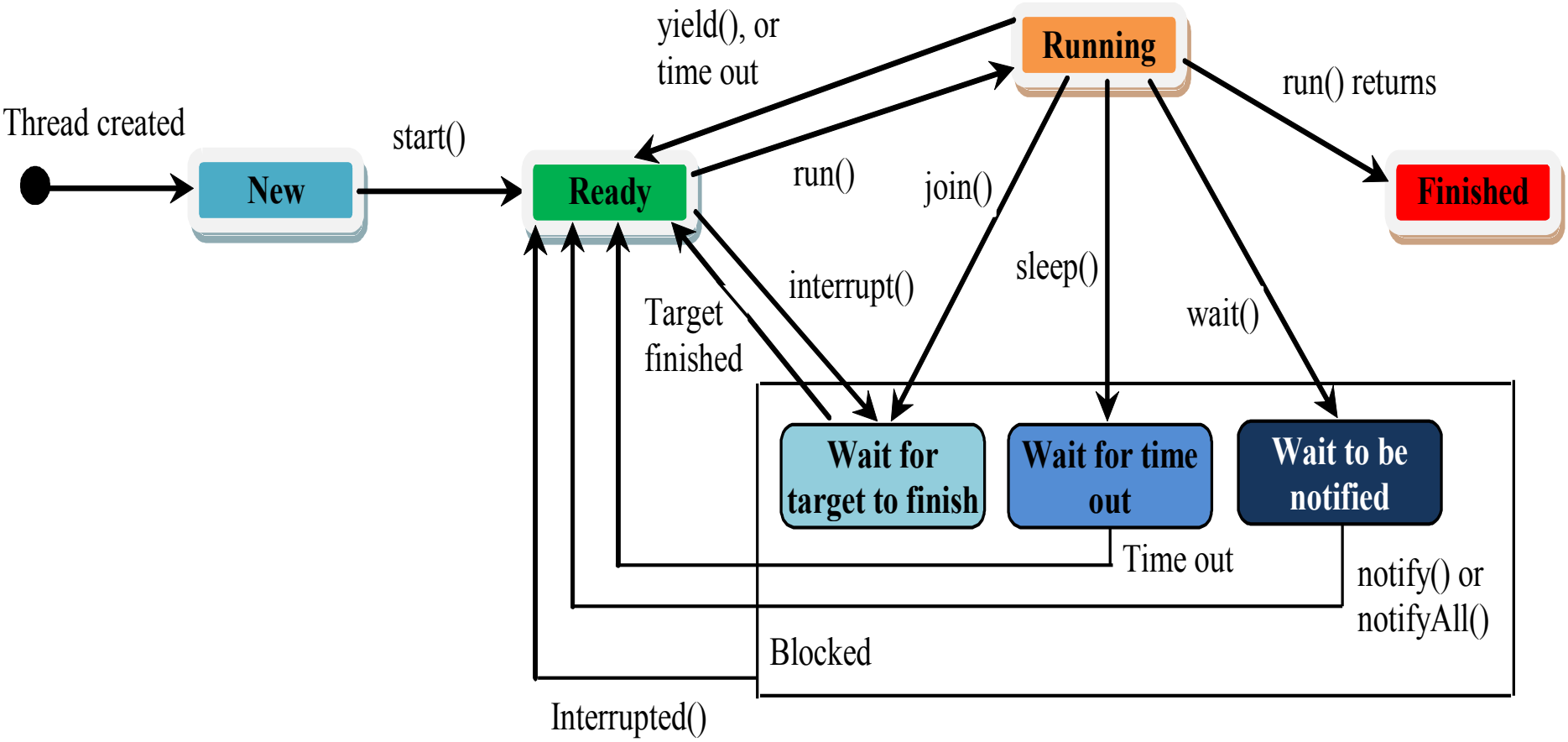
```
class SampleThread extends Thread {
  private String message;
  private double result;

  public SampleThread(String m) {this.message = m; this.result = 0.0;}
  public void run() {
     this.result = this.doAlgorithm();
  }
  private double doAlgorithm() {
     double val = 0;
     val++; //but could be 200 lines of code running for 10 sec
     try {
       this.sleep(3000);
     } catch(InterruptedException ie) { ie.printStackTrace(); }
     return val;
  }
  public double getSolution() {return this.result;}
}


public class ProgMainJoin {
  public static void main(String[] args) {
   SampleThread t = new SampleThread("T1");

   t.start();
   try {
        t.join(); //if this is comment, then the solution is NOT consistent
        System.out.println("Result = " + t.getSolution());
   } catch(InterruptedException ie) { ie.printStackTrace(); }
 }
}
```

1. Threads States - Recapitulation



1. Threads States – Update

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.



1. Threads Methods

isAlive()

- method used to find out the state of a thread.
- returns true: thread is in the Ready, Blocked, or Running state
- returns false: thread is new and has not started or if it is finished.

interrupt()

f a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

The isInterrupt() method tests whether the thread is interrupted.

1. Threads Methods

The deprecated stop(), suspend(), and resume() Methods

NOTE: The Thread class also contains the stop(), suspend(), and resume() methods. As of Java 2, these methods are ***deprecated*** (or *outdated*) because they are known to be inherently unsafe. You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

Thread Priority

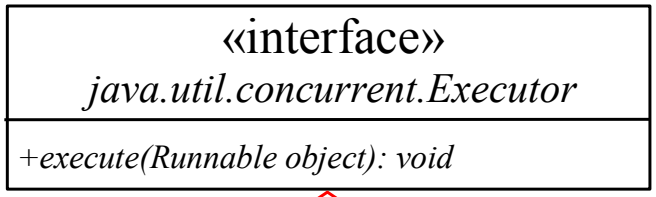
- Each thread is assigned a default priority of `Thread.NORM_PRIORITY` (constant of 5). You can reset the priority using `setPriority(int priority)`.
- Some constants for priorities include `Thread.MIN_PRIORITY` `Thread.MAX_PRIORITY` `Thread.NORM_PRIORITY`
- By default, a thread has the priority level of the thread that created it.
- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use *timeslicing* for threads of equal priority.
- ***Preemptive scheduling***: when a thread of higher priority enters the running state, it preempts the current thread.
- ***Starvation***: Higher-priority threads can postpone (possible forever) the execution of lower-priority threads.

Java Multithreading Cooperation - Thread Priority

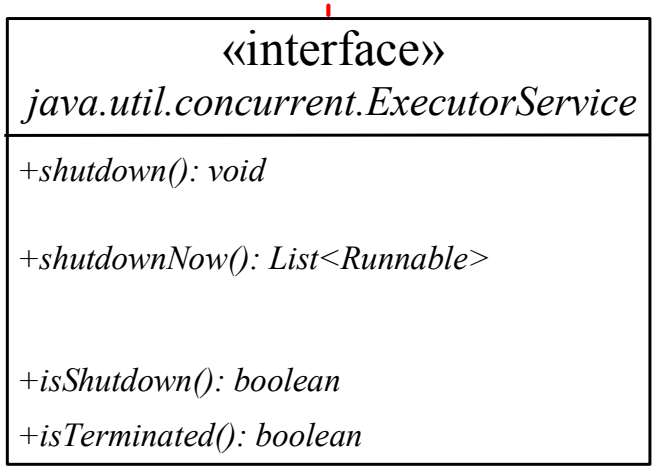
```
class SampleThread extends Thread {
  String message;
  public SampleThread(String m) {this.message = m;}
  public void run() {
     for (int i = 0; i < 100; i++) {
       System.out.println(this.message);
       this.yield();
     }
  }
}


public class ProgMainPriority {
  public static void main(String[] args) {
   SampleThread f1 = new SampleThread("f1");
   SampleThread f2 = new SampleThread("f2");

    Thread curent = Thread.currentThread();
    f1.setPriority(curent.getPriority() - 1);
    f2.setPriority(curent.getPriority() + 3);
    f1.start();
    f2.start();
 }
}
```

Thread Pool

- Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool is ideal to manage the number of tasks executing concurrently.
- **Executor** interface for executing Runnable objects in a thread pool
- **ExecutorService** is a sub-interface of Executor.



Executes the runnable task.



Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

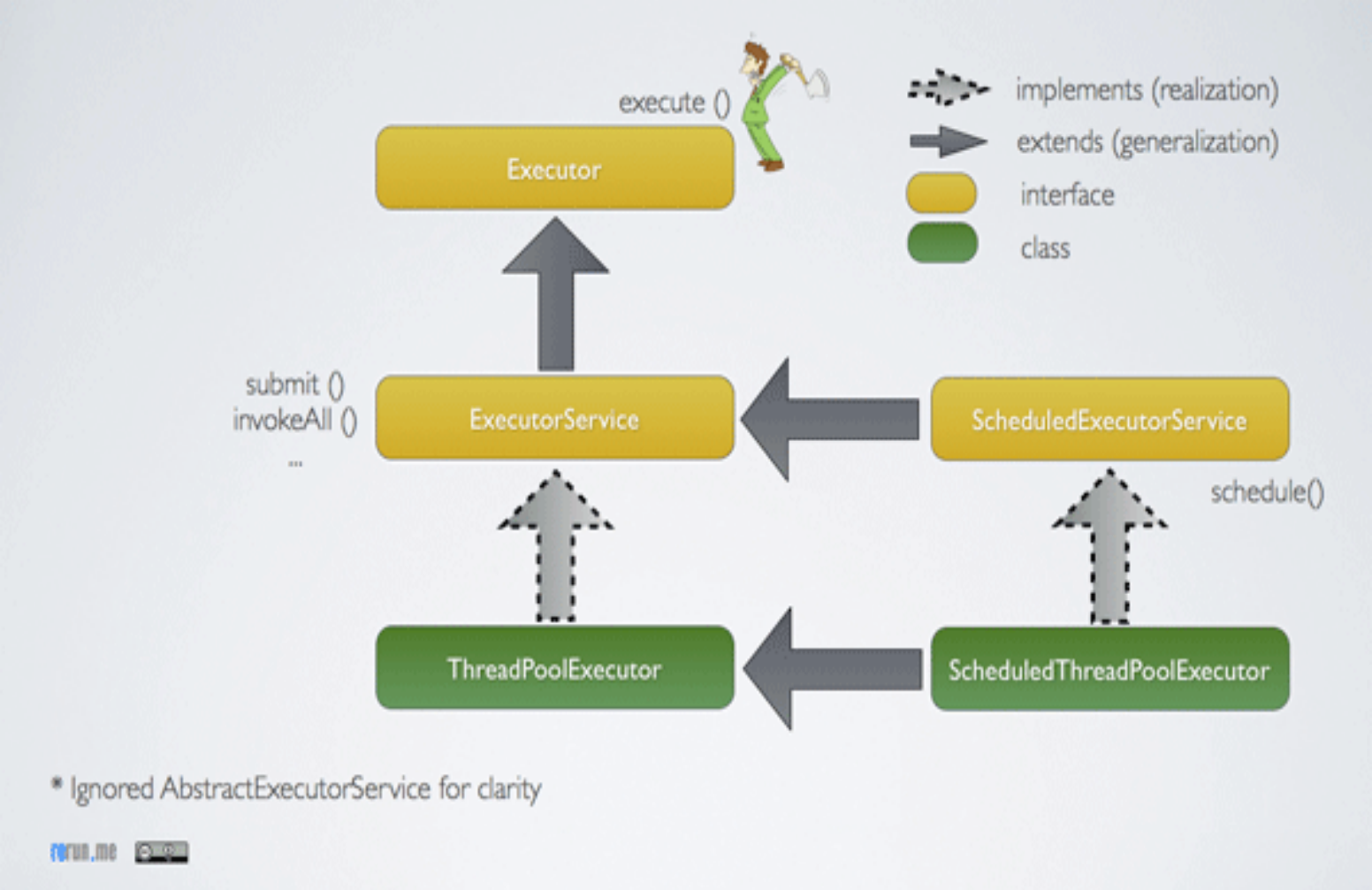
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shutdown.

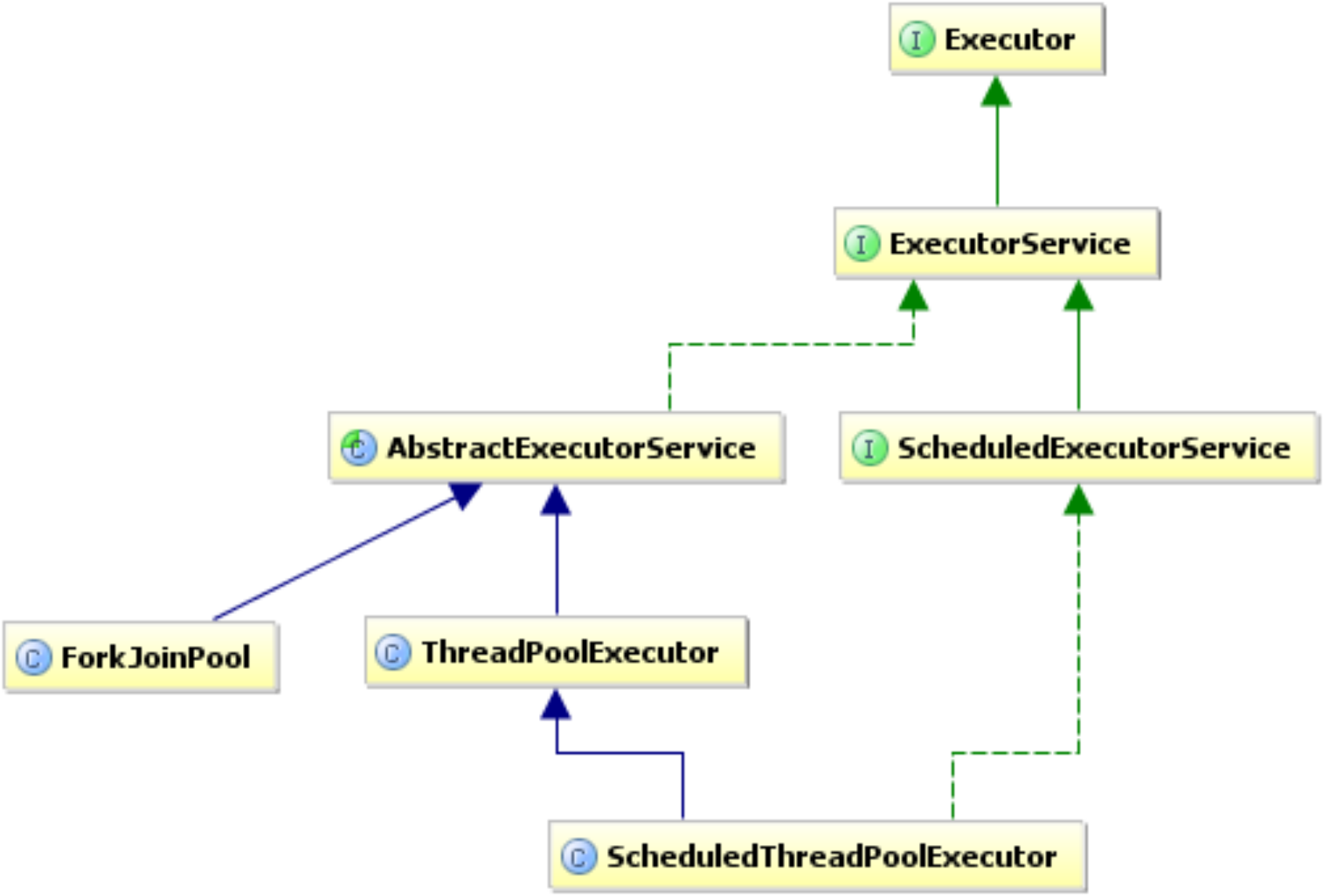
Returns true if all tasks in the pool are terminated.

Thread Pool

To create an Executor object, use the static methods in the Executors class.



1. Threads Concurrency & Parallelism



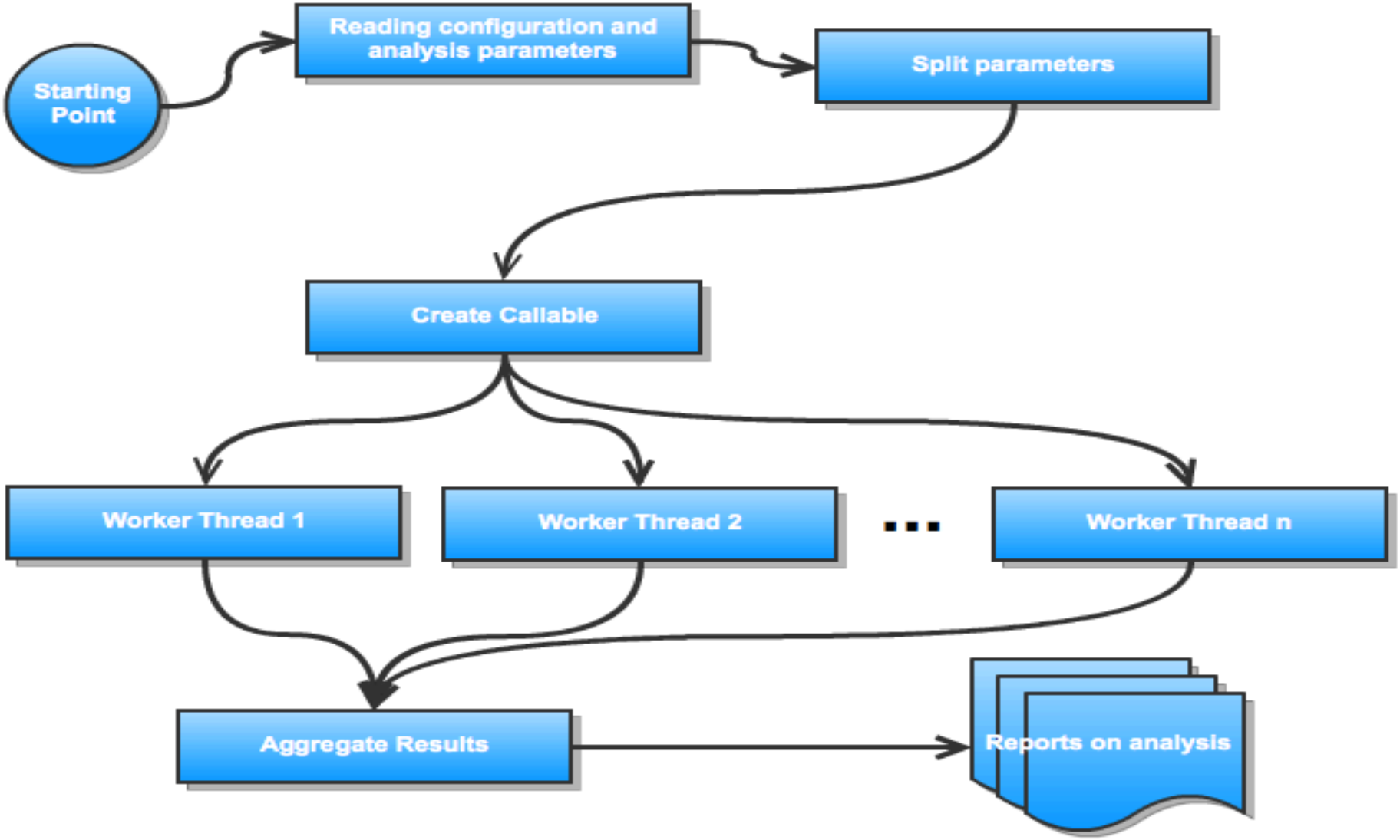
1. Threads Concurrency & Parallelism

```
import java.util.concurrent.*;
class PrintChar implements Runnable
{
   private char character;
   private int noOfTimes;
   PrintChar(char ch, int n) {
      character = ch;
      noOfTimes = n;
   }
   public void run() {
    for(int i=0; i<noOfTimes; i++)
   System.out.print(character+" ");
   } //end run
}  //end class
class PrintNum implements Runnable
{
   private int num;
   PrintNum(int num) {
      this.num = num;
   }
   public void run() {
      for(int i = 0; i < num; i++)
         System.out.print(i + " ");
   } //end run
}  //end class
```

Thread Pool

```
class ThreadPool_ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum
three threads
    ExecutorService executor =
Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();

    //Optional if main methods needs the
results
    executor.awaitTermination();
    System.out.println("\nMain Thread Ends");
  }
}
```

1. Threads Concurrency & Parallelism



1. Threads Concurrency & Parallelism

"Classic" Thread Pool, Executor Framework, Callable & Future

```
public class MyRunnable implements Runnable
{
  private final long countUntil;

  MyRunnable(long countUntil) {
    this.countUntil = countUntil;
  }

  @Override
  public void run() {
    long sum = 0;
    for (long i = 1; i < countUntil; i++) {
      sum += i;
    }
    System.out.println(sum);
  }
}
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

1. Threads Concurrency & Parallelism

"Classic" Thread Pool, Executor Framework, Callable & Future

```
import java.util.ArrayList;
import java.util.List;
public class ProgMain {
  public static void main(String[] args) {
    // We will store the threads so that we can check if they are done
    List<Thread> threads = new ArrayList<Thread>();
    for (int i = 0; i < 500; i++) {
      Runnable task = new MyRunnable(10000000L + i);
      Thread worker = new Thread(task);
      worker.setName(String.valueOf(i));
      // Start the thread, never call method run() direct
      worker.start();
      threads.add(worker);
    }
    int running = 0;
    do {
      running = 0;
      for (Thread thread : threads) {
        if (thread.isAlive()) { running++; }
      }
      System.out.println("We have " + running + " running threads. ");
    } while (running > 0);
  } //end main
} //end class
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

1. Threads Concurrency & Parallelism

"Classic" Thread Pool, Executor Framework, Callable & Future

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Runnable worker = new MyRunnable(10000000L + i);
      executor.execute(worker);
    }
    // This will make the executor accept no new threads
    // and finish all existing threads in the queue
    executor.shutdown();
    // Wait until all threads are finish
    executor.awaitTermination();
    System.out.println("Finished all threads");
  }
}
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

1. Threads Concurrency & Parallelism

"Classic" Thread Pool, Executor Framework, Callable & Future

```
//package de.vogella.concurrency.callables;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {

 @Override
 public Long call() throws Exception {
   long sum = 0;
   for (long i = 0; i <= 100; i++) {
       sum += i;
   }
   return sum;
 }
}
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

1. Threads Concurrency & Parallelism

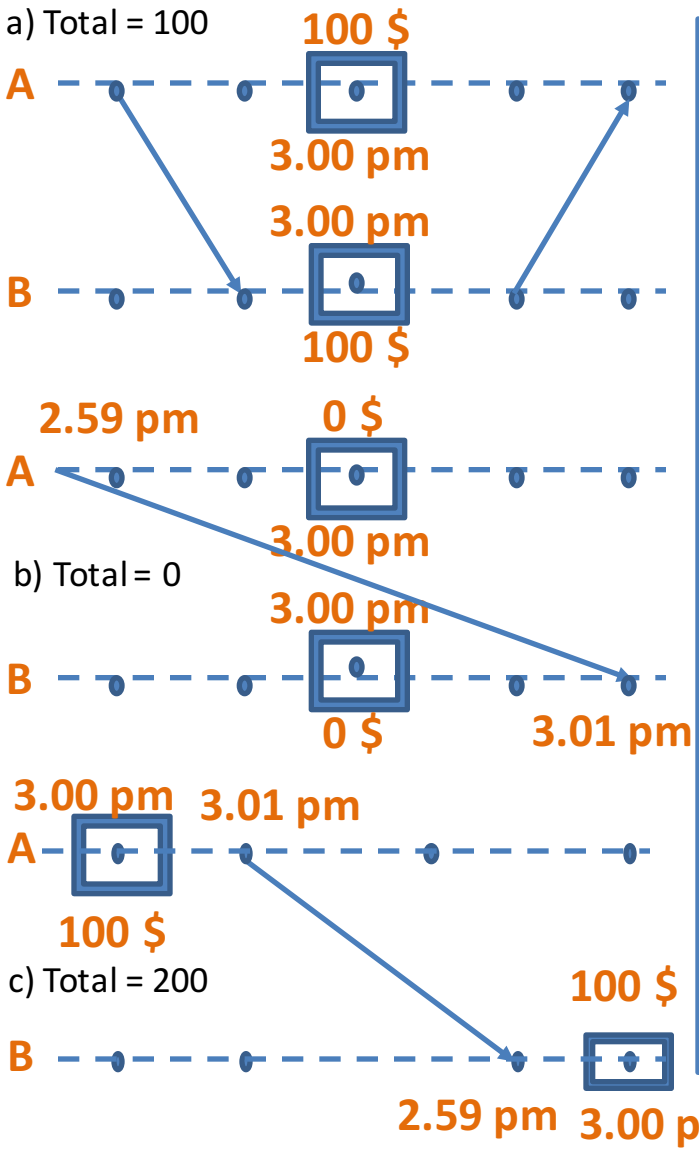
"Classic" Thread Pool, Executor Framework, Callable & Future

```
import java.util.*;
import java.util.concurrent.*;
public class CallableFutures {
  private static final int NTHREDS = 10;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
    long sum = 0;
    // now retrieve the result - System.out.println(list.size());
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum); executor.shutdown();
  }
}
```

http://www.vogella.com/tutorials/JavaConcurrency/article.html

1. Threads Concurrency – Synchronization

Analogy with multi-threading synch is the synch in distributed systems



Horizontal lines are the time axis and the points are the potential hours when an event may occur.

In the use-case a), the person with 2 bank accounts in the subsidiary A and B has in total, 200 $, at 3.00 pm => 100$ in subsidiary A nothing in subsidiary B. The total amount in the both subsidiaries A and B is 200 $. There are 2 assumptions:

1. Both servers clocks of the subsidiary A and subsidiary B are perfectly synchronized.
2. All the transactions between the both subsidiaries are all done fully before or after 3.00 pm.

In the use-case b), the servers' clocks are in sync, but a 100$ transfer transaction from the subsidiary A to B starts at 2.59 pm and it stops at 3.01. If the total will be calculated at 3.00 pm, then the overall amount is 0 $.

In the use-case c), the person will have the double amount of money (200 $), at 3.00 pm, because the servers clock are not synchronized.

1. Threads Concurrency – Synchronization

A shared resource may be corrupted if it is accessed simultaneously by multiple threads.

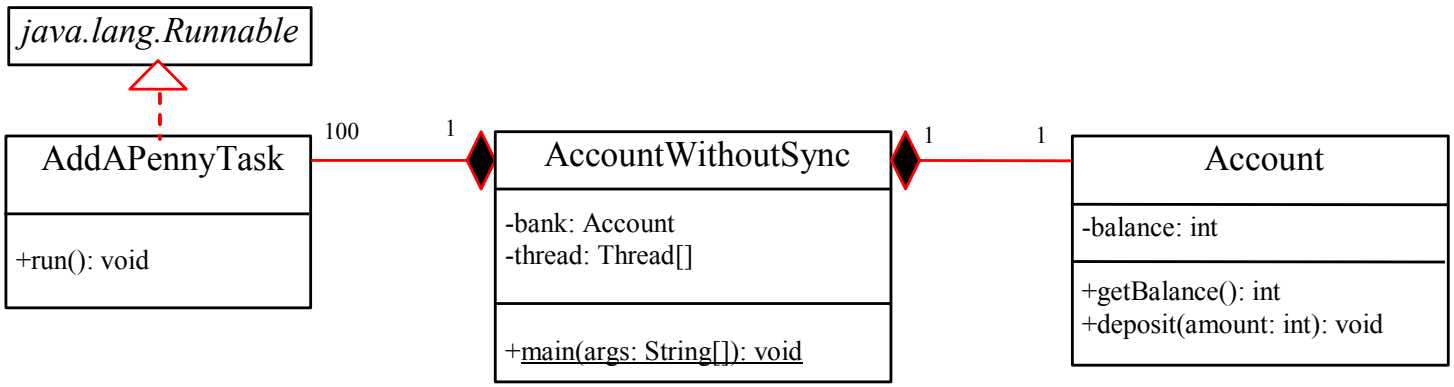
Example: two unsynchronized threads accessing the same bank account may cause conflict.

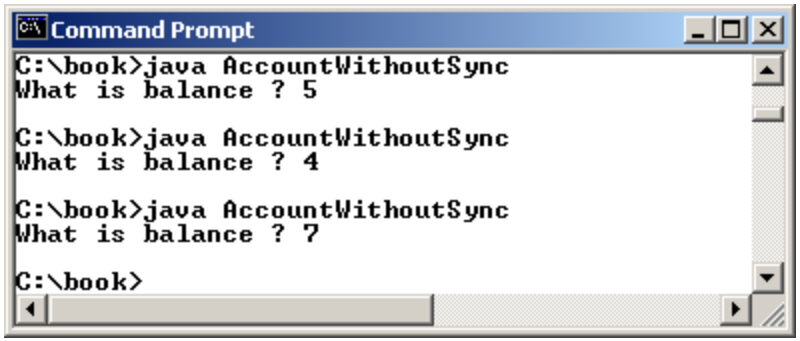
Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

1. Threads Concurrency – Synchronization

Example: Showing Resource Conflict – AccountWithoutSynch.java

- Objective: Write a program that demonstrates the problem of resource conflict. Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty.





Race Condition

What, then, caused the error in the example? Here is a possible scenario:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

- Effect: Task 1 did nothing (in Step 4 Task 2 overrides the result)
- Problem: Task 1 and Task 2 are accessing a common resource in a way that causes conflict.
- Known as a *race condition* in multithreaded programs.
- A *thread-safe* class does not cause a race condition in the presence of multiple threads.
- The Account class is not thread-safe.

1. Threads Concurrency – Synchronization

synchronized

- Problem: race conditions
- Solution: give exclusive access to one thread at a time to code that manipulates a shared object.
- Synchronization keeps other threads waiting until the object is available.
- The synchronized keyword synchronizes the method so that only one thread can access the method at a time.
- The critical region in the AccountWithoutSynch.java is the entire deposit method.
- One way to correct the problem in source code: make Account thread-safe by adding the synchronized keyword in deposit:

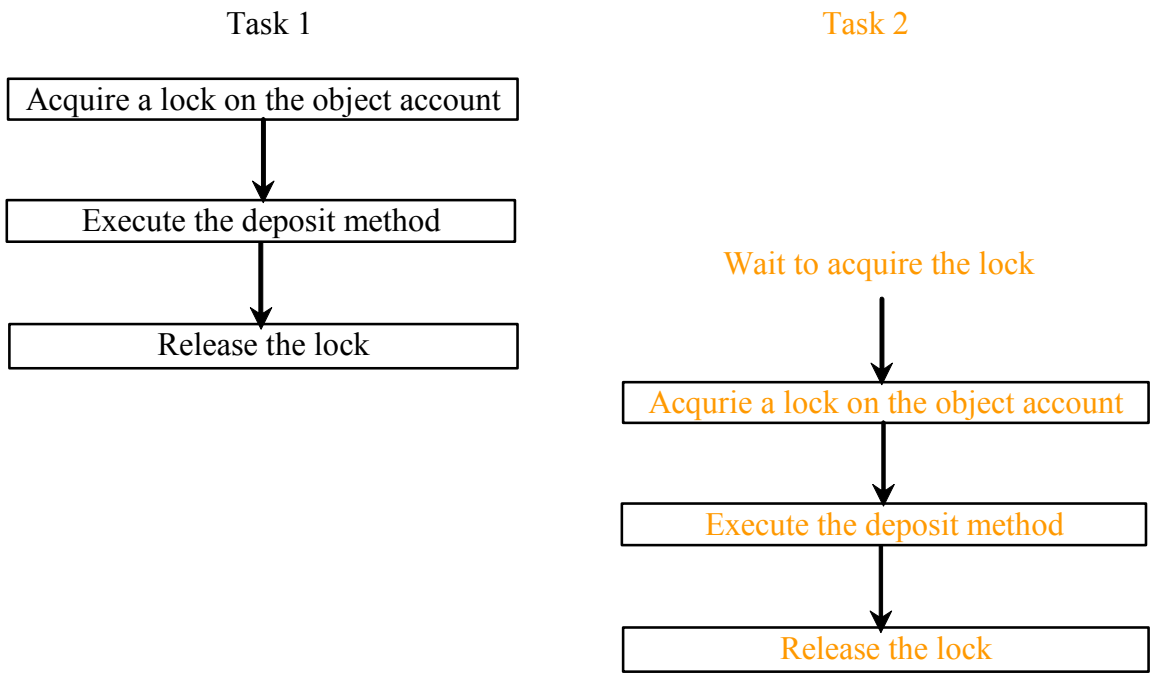
public synchronized void deposit(double amount)

Synchronizing Instance Methods and Static Methods

- A synchronized method acquires a lock before it executes.
- Instance method: the lock is on the object for which it was invoked.
- Static method: the lock is on the class.
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired, then the method is executed, and finally the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

Synchronizing Instance Methods and Static Methods

With the deposit method synchronized, the preceding scenario cannot happen. If Task 2 starts to enter the method, and Task 1 is already in the method, Task 2 is blocked until Task 1 finishes the method.



Synchronizing Statements

- Invoking a synchronized instance method of an object acquires a lock on the object.

- Invoking a synchronized static method of a class acquires a lock on the class.

- A *synchronized block* can be used to acquire a lock on any object, not just *this* object, when executing a block of code.

```
synchronized (expr) {
  statements;
}
```

- expr must evaluate to an object reference.

- If the object is already locked by another thread, the thread is blocked until the lock is released.

- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

1. Multi-threading in Java

Java Multi-threading Synchronization for Concurrent (Producer-Consumer) Programming

Critical region = is a section of the byte-code/executable machine code, which is impacting in a concurrent manner the same area of memory, from a multi-threading environment. The role of the synchronization in this case is to ensure the exclusive access to the common memory Resource.

The methods wait(), notify(), notifyAll() (e.g. in the producer-consumer sample) are inherited from the java.lang.Object class, NOT from Thread class. The wait() method can be called only from a synchronized method. Sometimes the boolean condition for waiting a thread may be associated with a if/while statement plus a flag showing if the should still wait or not the boolean condition (e.g. in the producer-consumer sample):

Produce:

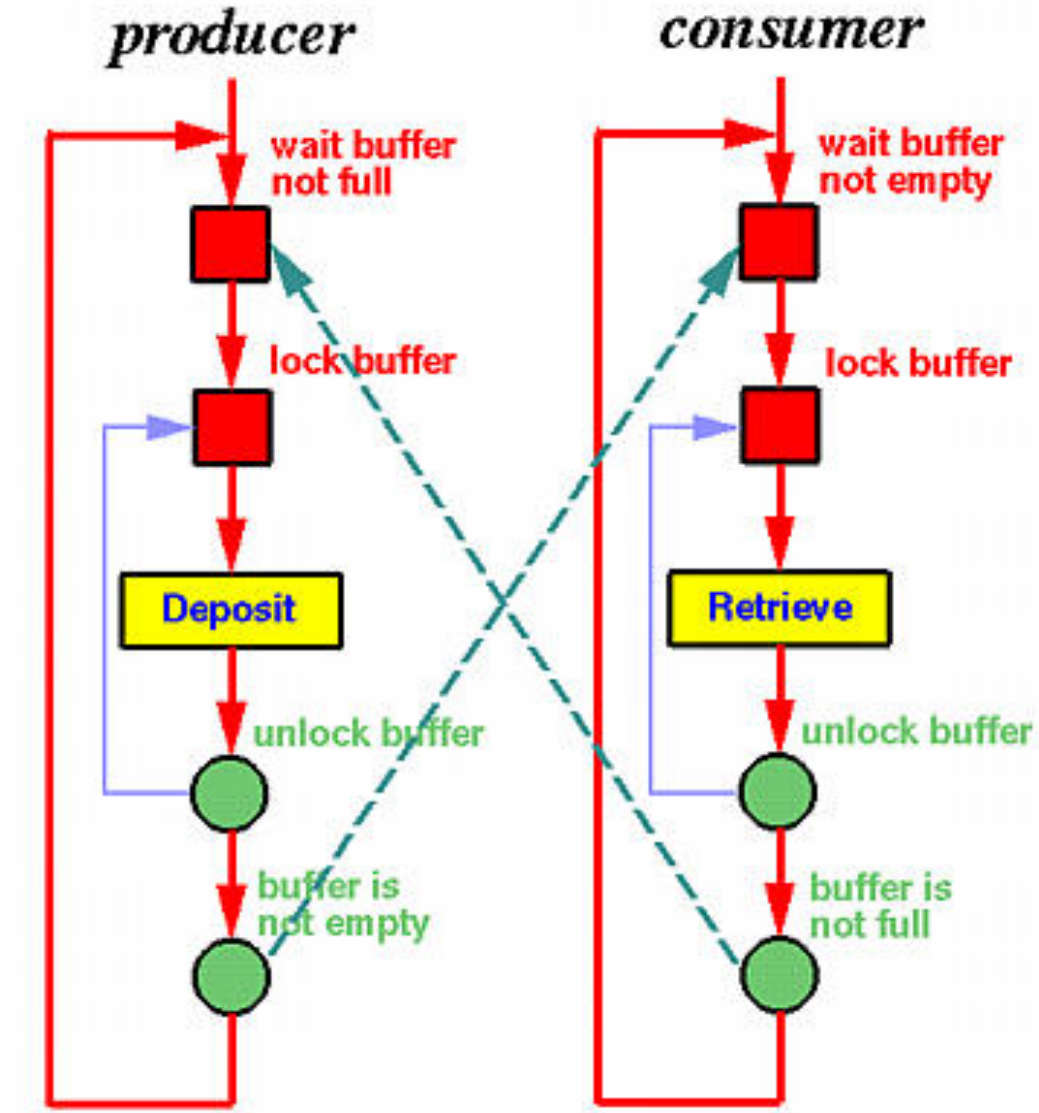
```
while (Condition) {
    wait();
}
availableFlag = true;
notifyAll();
```

Consume:

```
while (! Condition) {
    wait();
}
availableFlab = false;
notifyAll();
```

1. Multi-threading in Java

Java Multi-threading Synchronization for Concurrent (Producer-Consumer) Programming



1. Multi-threading in Java

Java Multi-threading Cooperation

The method yield() allows the JVM and OS to commute the control from the current thread to another thread. In the samples, in the lecture, are samples for showing the setPriority()/getPriority(), join(), join(…), isAlive() and interrupt() methods.

For the common resource that is in the memory, a **race condition** may appear in the concurrency created into an multi-threading environment and it is recommended to use **volatile** keyword for that object/variable from the memory, in order to instruct the JVM to write immediately in the memory, instead keeping into the register for the optimization.

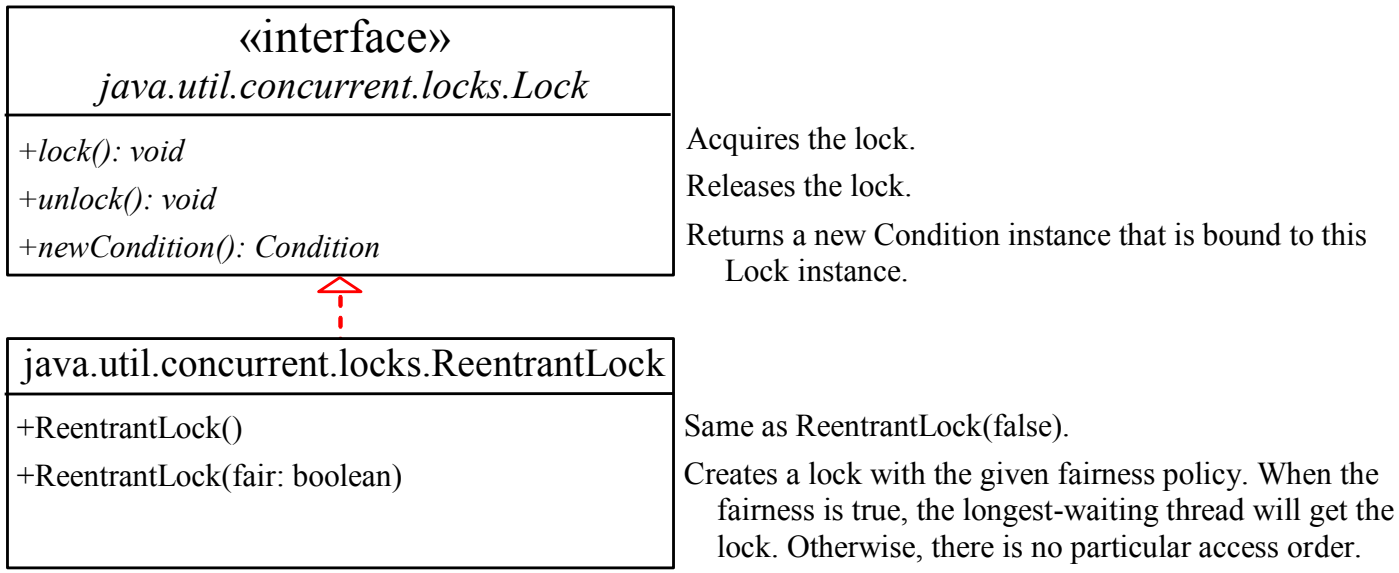
Java MUTEX ⇔ synchronized * is recommended to use it only at method level if the method is NOT into a class which implements Runnable or extends Thread.

What is the difference between the semaphore and mutex in JVM? What is the difference between semaphore and mutex in the POSIX Threads within Linux OS?

What objects/instances are thread-safe within JVM? – immutable, singleton, POJO?

1. Threads Concurrency – Synchronization using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- You can use locks explicitly to obtain more control for coordinating threads.
- A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks.
- newCondition() method creates Condition objects, which can be used for thread communication.



Fairness Policy

- ReentrantLock: concrete implementation of Lock for creating mutually exclusive locks.
- Create a lock with the specified fairness policy.
- True fairness policies guarantee the longest-wait thread to obtain the lock first.
- False fairness policies grant a lock to a waiting thread without any access order.
- Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

This example revises AccountWithoutSync.java in AccountWithoutSyncUsingLock.java to synchronize the account modification using explicit locks.

Cooperation Among Threads

- Conditions can be used for communication among threads.
- A thread can specify what to do under a certain condition.
- newCondition() method of Lock object.

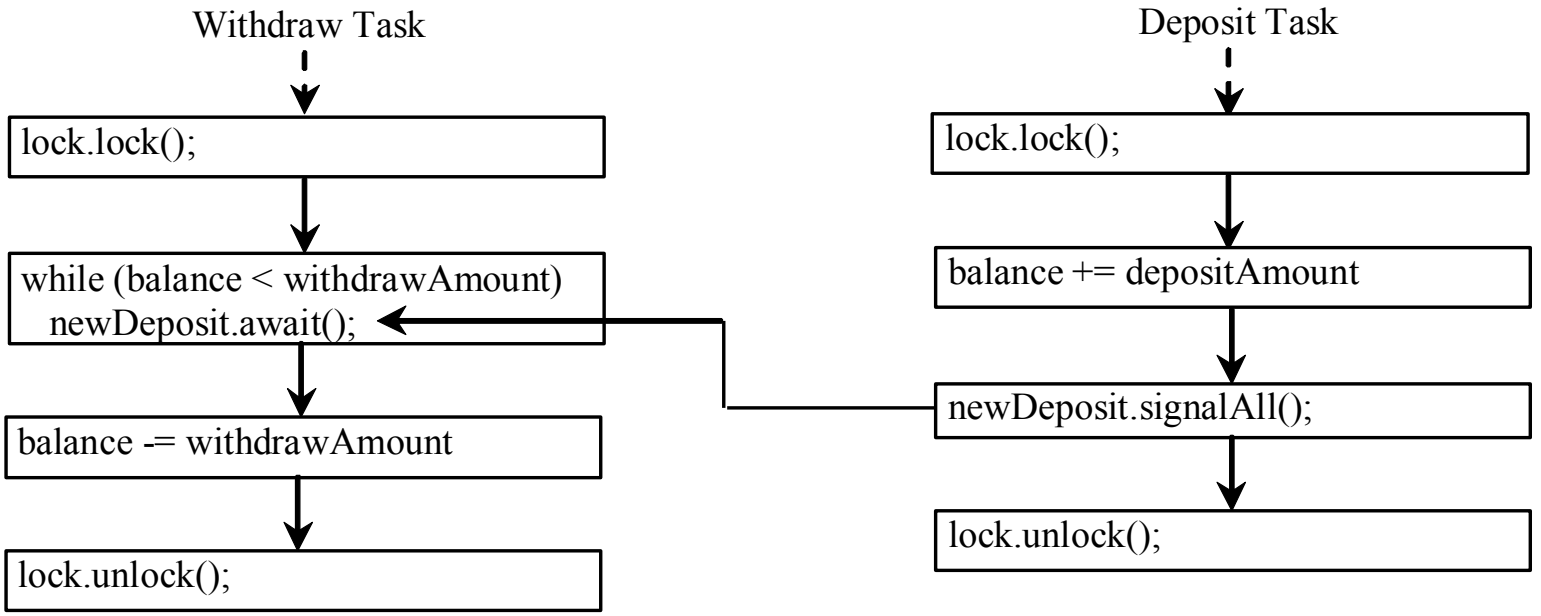
- Condition methods:
  - await() current thread waits until the condition is signaled
  - signal() wakes up a waiting thread
  - signalAll() wakes all waiting threads

«interface» *java.util.concurrent.Condition*	
+await(): void	Causes the current thread to wait until the condition is signaled.
+signal(): void	Wakes up one waiting thread.
+signalAll(): Condition	Wakes up all waiting threads.

Cooperation Among Threads

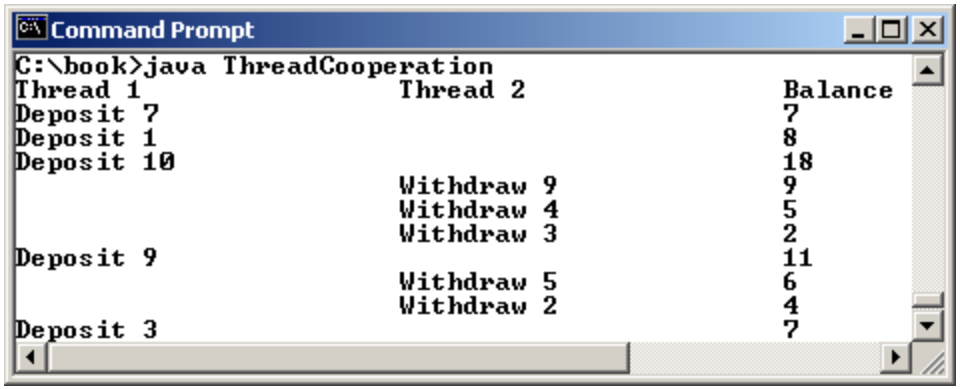
- Lock with a condition to synchronize operations: newDeposit
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.

- Interaction between the two tasks:



Example: Thread Cooperation

Write a program that demonstrates thread cooperation (ThreadCooperation.java). Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.



Java's Built-in Monitors

- Locks and conditions are new starting with Java 5.
- Prior to Java 5, thread communications were programmed using object's built-in monitors.

- Locks and conditions are more powerful and flexible than the built-in monitor.
- A *monitor* is an object with mutual exclusion and synchronization capabilities.

- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock (synchronized keyword on method / block) on the monitor and exits by releasing the lock.
- A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

- *Any object can be a monitor*. An object becomes a monitor once a thread locks it.

wait(), notify(), and notifyAll()

Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

1. Threads Synchronization - Monitors

Example: Using Monitor

Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

   // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```

resume

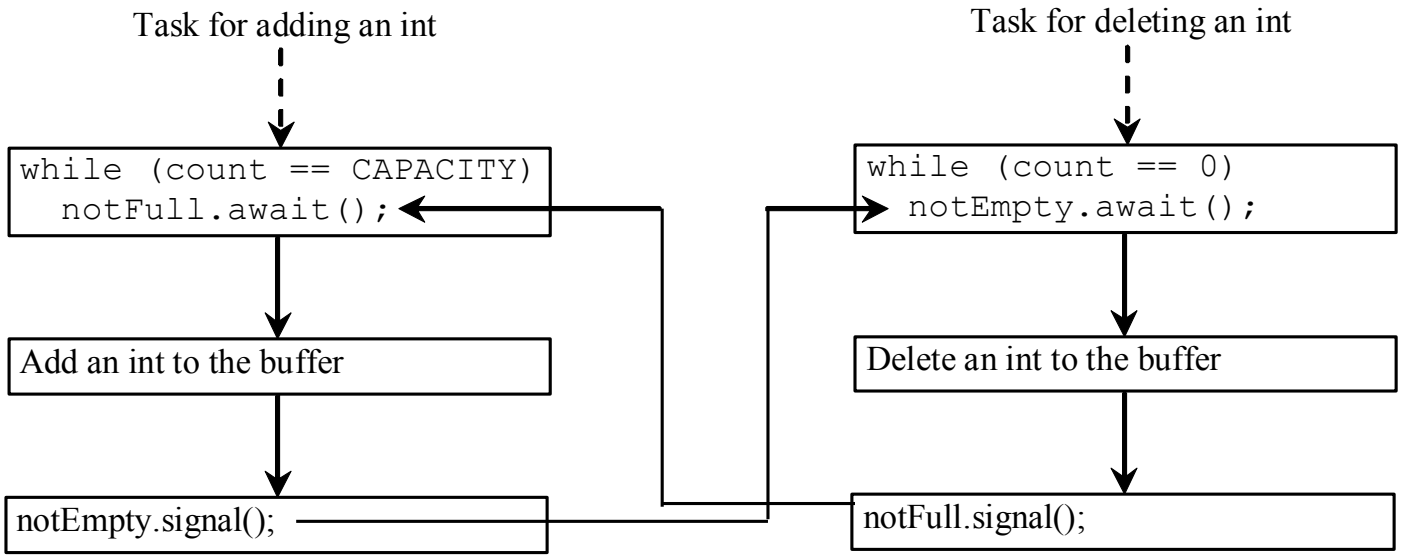
Task 2

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or  anObject.notifyAll();
  ...
}
```

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.
- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.
- The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Case Study: Producer/Consumer

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: notEmpty (i.e., buffer is not empty) and notFull (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition. The interaction between the two tasks is shown:



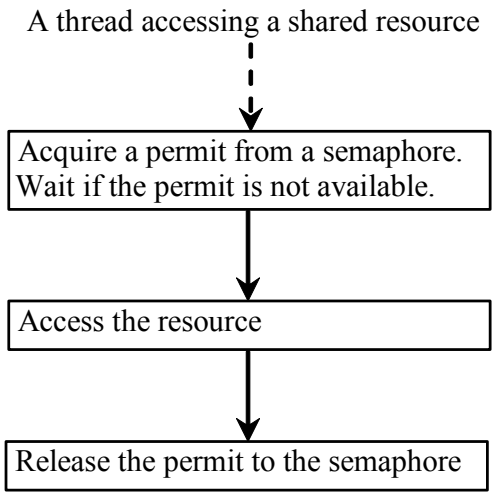
Case Study: Producer/Consumer

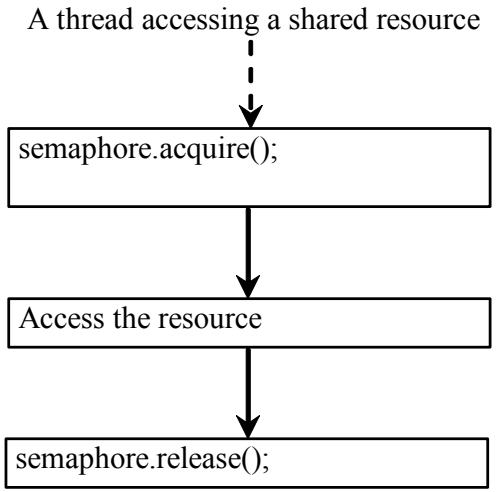
ConsumerProducer.java presents the complete program. The program contains the Buffer class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The write(int) method (line 58) adds an integer to the buffer. The read() method (line 75) deletes and returns an integer from the buffer.

For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

Semaphores

Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown:





1. Threads Concurrency on Shared Resource

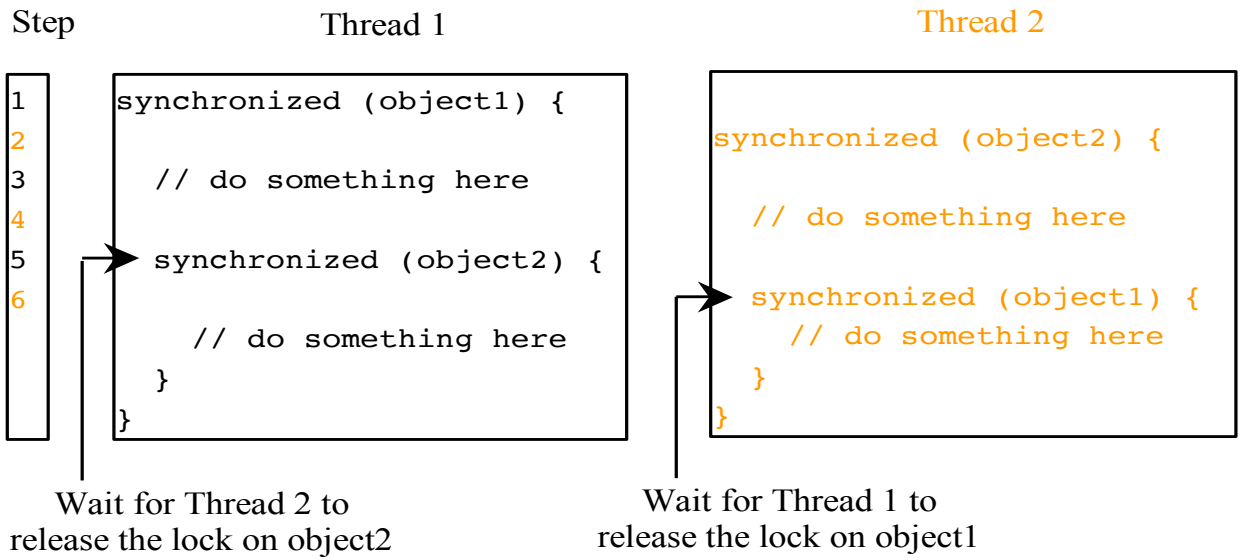
Creating Semaphores

To create a semaphore, you have to specify the number of permits with an optional fairness policy (see the figure). A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- In the figure below, the two threads wait for each other to release the in order to get a lock, and neither can continue to run.



Preventing Deadlock

- Deadlock can be easily avoided by resource ordering.
- With this technique, assign an order on all the objects whose locks must be acquired and ensure that the locks are acquired in that order.
- How does this prevent deadlock in the previous example?

Synchronized Collections

- The classes in the Java Collections Framework are not thread-safe.
- Their contents may be corrupted if they are accessed and updated concurrently by multiple threads.
- You can protect the data in a collection by locking the collection or using synchronized collections.

The Collections class provides six static methods for creating *synchronization wrappers*.

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Returns a synchronized collection.
+synchronizedList(list: List): List	Returns a synchronized list from the specified list.
+synchronizedMap(m: Map): Map	Returns a synchronized map from the specified map.
+synchronizedSet(s: Set): Set	Returns a synchronized set from the specified set.
+synchronizedSortedMap(s: SortedMap): SortedMap	Returns a synchronized sorted map from the specified sorted map.
+synchronizedSortedSet(s: SortedSet): SortedSet	Returns a synchronized sorted set.

Vector, Stack, and Hashtable

Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) { return c.add(o); }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized. These are old classes introduced in JDK 1.0. In JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.

Fail-Fast

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, suppose you want to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.

GUI Event Dispatcher Thread

GUI event handling and painting code executes in a single thread, called the *event dispatcher thread*. This ensures that each event handler finishes executing before the next one executes and the painting isn't interrupted by events.

GUI Event Dispatcher Thread

invokeLater and invokeAndWait

In certain situations, you need to run the code in the event dispatcher thread to avoid possible deadlock. You can use the static methods, invokeLater and invokeAndWait, in the javax.swing.SwingUtilities class to run the code in the event dispatcher thread. You must put this code in the run method of a Runnable object and specify the Runnable object as the argument to invokeLater and invokeAndWait. The invokeLater method returns immediately, without waiting for the event dispatcher thread to execute the code. The invokeAndWait method is just like invokeLater, except that invokeAndWait doesn't return until the event-dispatching thread has executed the specified code.

GUI Event Dispatcher Thread

Launch Application from Main Method

So far, you have launched your GUI application from the main method by creating a frame and making it visible. This works fine for most applications. In certain situations, however, it could cause problems. To avoid possible thread deadlock, you should launch GUI creation from the event dispatcher thread as follows:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      // Place the code for creating a frame and setting it properties
    }
  });
}
```

Section Conclusion

Fact: **Multi-threading vs. Multi-process**

In few **samples** it is easy to understand:

- Necessity: Parallelism vs. Concurrency
- Multi-Process vs. Multi-Threading
- Atomic operations (counter++)
- Multi-threading model mapping

Java 8 Multi-Threading Lambda and Functional Interface in HPC Use Cases | Same concepts from the previous sections are translated into Lambda expressions for the multi-threading

Java 8 Lambda Multi-Threading Intro

2. Java 8 – Lambda Threads & Executors

The Concurrency API was first introduced with the release of Java 5 and then progressively enhanced with every new Java release.

This section is dedicating for Multi-threading Programming using Java 8 Lambda Expressions and Runnable Functional Interface.

All modern operating systems support concurrency both via processes and threads. Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently and/or parallel (e.g. Java Fork-Join mechanism), so we can make the most out of the available cores of the CPU.

2. Java 8 – Lambda Threads & Executors – Runnable/Thread

Java supports Threads since JDK 1.0. Before starting a new thread you have to specify the code to be executed by this thread, often called the *task*. This is done by implementing Runnable - a functional interface defining a single void no-args method run() as demonstrated in the following example:

```
Runnable taskJava7 = new Runnable () {
        @Override
        public void run() {
                String threadName = Thread.currentThread().getName()
                System.out.println("Hello " + threadName);
        }
};

taskJava7.run();

Thread threadJ7 = new Thread(taskJava7);
threadJ7.start();

Runnable taskJava8 = () -> {
        String threadName = Thread.currentThread().getName();
        System.out.println("Hello " + threadName);
};

taskJava8.run();

Thread threadJ8 = new Thread(taskJava8);
threadJ8.start();
```

2. Java 8 – Lambda Threads & Executors - Executors

Working with the Thread class can be very tedious and error-prone. Due to that reason the **Concurrency API** has been introduced back in 2004 with the release of Java 5. The API is located in package **java.util.concurrent** and contains many useful classes for handling concurrent programming. Since that time the **Concurrency API** has been enhanced with every new Java release and even Java 8 provides new classes and methods for dealing with concurrency.

The Concurrency API introduces the concept of an **ExecutorService** as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually. All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

```
ExecutorService executor = Executors.newSingleThreadExecutor();

executor.submit( () -> {
                        String threadName = Thread.currentThread().getName();
                        System.out.println("Hello " + threadName);
}); // => Hello pool-1-thread-1
```

2. Java 8 – Lambda Threads & Executors - Executors

The result looks similar to the first Runnable lambda sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

An ExecutorService provides two methods for that purpose: shutdown() waits for currently running tasks to finish while shutdownNow() interrupts all running tasks and shut the executor down immediately. Typically shutdown executors statements:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.err.println("tasks interrupted");
} finally {
    if (!executor.isTerminated()) { System.err.println("cancel non-finished tasks"); }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

The executor shuts down softly by waiting a certain amount of time for termination of currently running tasks. After a maximum of five seconds the executor finally shuts down by interrupting all running tasks.

2. Java 8 – Lambda Threads & Executors – Future-Callable

In addition to **Runnable executors** support another kind of task named **Callable**. Callables are functional interfaces just like Runnable, but instead of being void, they return a value.
This lambda expression defines a callable returning an integer after sleeping for one second:

```
Callable<Integer> task = () -> {
   try { TimeUnit.SECONDS.sleep(1); return 123; }
   catch (InterruptedException e) { throw new IllegalStateException("task interrupted", e); }
};
```

Callables can be submitted to executor services just like runnables. But what about the callables result? Since submit() doesn't wait until the task completes, the executor service cannot return the result of the callable directly. Instead the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);
System.out.println("future  done? " + future.isDone());
Integer result = future.get();
System.out.println("future  done? " + future.isDone());
System.out.print("result: " + result);
```

After submitting the callable to the executor we first check if the future has already been finished execution via isDone(). This isn't the case since the above callable sleeps for one second before returning the integer. Calling the method get() blocks the current thread and waits until the callable completes before returning the actual result 123.

2. Java 8 – Lambda Threads & Executors – Future-Callable

Timeouts

Any call to future.get() will block and wait until the underlying callable has been terminated. In the worst case a callable runs forever - thus making your application unresponsive. You can simply counteract those scenarios by passing a timeout:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit( ( ) -> {
    try { TimeUnit.SECONDS.sleep(2); return 123; }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});
future.get(1, TimeUnit.SECONDS);
```

Executing the above code results in a TimeoutException:
Exception in thread "main" java.util.concurrent.TimeoutException at java.util.concurrent.FutureTask.get(FutureTask.java:205)

You might already have guessed why this exception is thrown: We specified a maximum wait time of one second but the callable actually needs two seconds before returning the result.

2. Java 8 – Lambda Threads & Executors – Future-Callable

InvokeAll - see Lambda and Java 8 Processing Streams

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 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);
```

In this example we utilize Java 8 functional streams in order to process all futures returned by the invocation of invokeAll. We first map each future to its return value and then print each value to the console.

2. Java 8 – Lambda Threads & Executors – Future-Callable

InvokeAny

Another way of batch-submitting callables is the method **invokeAny**() which works slightly different to invokeAll(). ***Instead of returning future objects this method blocks until the first callable terminates and returns the result of that callable.***
In order to test this behavior we use this helper method to simulate callables with different durations. The method returns a callable that sleeps for a certain amount of time until returning the given result:

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

We use this method to create a bunch of callables with different durations from one to three seconds. Submitting those callables to an executor via **invokeAny**() returns the string result of the fastest callable - in that case task2:

```
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
```

The above example uses yet another type of executor created via **newWorkStealingPool**(). This factory method is part of Java 8 and returns an executor of type **ForkJoinPool** which works slightly different than normal executors. Instead of using a fixed size thread-pool ForkJoinPools are created for a given parallelism size which per default is the number of available cores of the hosts CPU.

2. Java 8 – Lambda Threads & Executors – Future-Callable

Scheduled Executors

We've already learned how to submit and run tasks once on an executor. In order to periodically run common tasks multiple times, we can utilize scheduled thread pools.
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 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: %s ms", remainingDelay);
```

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.

2. Java 8 – Lambda Threads & Executors – Future-Callable

In order to schedule tasks to be executed periodically, executors provide the two methods **scheduleAtFixedRate**() and **scheduleWithFixedDelay**(). The first method is capable of executing tasks with a fixed time rate, e.g. once every second as demonstrated in this example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

Additionally this method accepts an initial delay which describes the leading wait time before the task will be executed for the first time. Please keep in mind that **scheduleAtFixedRate**() doesn't take into account the actual duration of the task. So if you specify a period of one second but the task needs 2 seconds to be executed then the thread pool will working to capacity very soon. In that case you should consider using **scheduleWithFixedDelay**() instead. This method works just like the counterpart described above. The difference is that the wait time period applies between the end of a task and the start of the next task. For example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
Runnable task = () -> {
  try { TimeUnit.SECONDS.sleep(2); System.out.println("Scheduling: " + System.nanoTime()); }
  catch (InterruptedException e) { System.err.println("task interrupted"); }
};
executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

This example schedules a task with a fixed delay of one second between the end of an execution and the start of the next execution. The initial delay is zero and the tasks duration is two seconds. So we end up with an execution interval of 0s, 3s, 6s, 9s and so on. As you can see **scheduleWithFixedDelay**() is handy if you cannot predict the duration of the scheduled tasks.

2. Java 8 – Lambda Thread: Synchronization & Locks

When writing multi-threaded code you have to pay particular attention when accessing shared mutable variables concurrently from multiple threads. Let's just say we want to increment an integer which is accessible simultaneously from multiple threads.

We define a field count with a method increment() to increase count by one:

```
int count = 0;
void increment() {
    count = count + 1;
}
```

When calling this method concurrently from multiple threads we're in serious trouble:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 10000).forEach(i -> executor.submit(this::increment));
// stop(executor); // procedure of shuting-down the executor-service
System.out.println(count); // 9965
```

Instead of seeing a constant result count of 10000 the actual result varies with every execution of the above code. The reason is that we share a mutable variable upon different threads without synchronizing the access to this variable which results in a race condition.

2. Java 8 – Lambda Thread: Synchronization & Locks

Java supports thread-synchronization since the early days via the synchronized keyword. We can utilize synchronized to fix the above race conditions when incrementing the count:

```
synchronized void incrementSync() { count = count + 1; }
```

When using incrementSync() concurrently we get the desired result count of 10000. No race conditions occur any longer and the result is stable with every execution of the code:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 10000).forEach(i -> executor.submit(this::incrementSync));
// stop(executor); // procedure of shuting-down the executor-service
System.out.println(count); // 10000
```

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.

All implicit monitors implement the *reentrant* characteristics. Reentrant means that locks are bound to the current thread. A thread can safely acquire the same lock multiple times without running into deadlocks (e.g. a synchronized method calls another synchronized method on the same object).

2. Java 8 – Lambda Thread: Synchronization & Locks

Locks

Instead of using implicit locking via the synchronized keyword the Concurrency API supports various explicit locks specified by the Lock interface. Locks support various methods for finer grained lock control thus are more expressive than implicit monitors. Multiple lock implementations are available in the standard JDK which will be demonstrated in the following sections.

ReentrantLock

The class ReentrantLock is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities. As the name suggests this lock implements reentrant characteristics just as implicit monitors. Let's see how the above sample looks like using ReentrantLock:

```
ReentrantLock myLock = new ReentrantLock();
int count = 0;
void increment() {
    myLock.lock();
    try { count++; }
    finally { myLock.unlock(); }
}
```

A lock is acquired via lock() and released via unlock(). It's important to wrap your code into a try/finally block to ensure unlocking in case of exceptions. This method is thread-safe just like the synchronized counterpart. If another thread has already acquired the lock subsequent calls to lock() pause the current thread until the lock has been unlocked. Only one thread can hold the lock at any given time.

2. Java 8 – Lambda Thread: Synchronization & Locks

Locks support various methods for fine grained control as seen in the next sample:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
ReentrantLock myLock = new ReentrantLock();
executor.submit(() -> {
    myLock.lock();
    try { sleep(2); }
    finally { myLock.unlock(); }
});
executor.submit(() -> { System.out.println("Locked: " + myLock.isLocked());
System.out.println("Held by me: " + myLock.isHeldByCurrentThread());
boolean locked = myLock.tryLock();
System.out.println("Lock acquired: " + locked); });
//stop(executor);
```

While the first task holds the lock for one second the second task obtains different information about the current state of the lock:

```
Locked: true
Held by me: false
Lock acquired: false
```

The method tryLock() as an alternative to lock() tries to acquire the lock without pausing the current thread. The boolean result must be used to check if the lock has actually been acquired before accessing any shared mutable variables.

2. Java 8 – Lambda Thread: Synchronization & Locks

ReadWriteLock

The interface ReadWriteLock specifies another type of lock maintaining a pair of locks for read and write access. The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. So the read-lock can be held simultaneously by multiple threads as long as no threads hold the write-lock. This can improve performance and throughput in case that reads are more frequent than writes.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();
executor.submit(() -> {
    lock.writeLock().lock();
    try { sleep(1); map.put("foo", "bar"); }
    finally { lock.writeLock().unlock(); } });
```

The above example first acquires a write-lock in order to put a new value to the map after sleeping for one second. Before this task has finished two other tasks are being submitted trying to read the entry from the map and sleep for one second:

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try { System.out.println(map.get("foo")); sleep(1); }
    finally { lock.readLock().unlock(); } };
executor.submit(readTask); executor.submit(readTask); // do NOT forget to stop(executor);
```

When you execute this code sample you'll notice that both read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel and print the result simultaneously to the console. They don't have to wait for each other to finish because read-locks can safely be acquired concurrently as long as no write-lock is held by another thread.

2. Java 8 – Lambda Thread: Synchronization & Locks

StampedLock

Java 8 ships with a new kind of lock called StampedLock which also support read and write locks just like in the example above. In contrast to ReadWriteLock the locking methods of a StampedLock return a stamp represented by a long value. You can use these stamps to either release a lock or to check if the lock is still valid. Additionally stamped locks support another lock mode called *optimistic locking*.

Let's rewrite the last example code to use StampedLock instead of ReadWriteLock:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock myLock = new StampedLock();
executor.submit(() -> {
    long stamp = myLock.writeLock();
    try { sleep(1); map.put("foo", "bar"); }
    finally { myLock.unlockWrite(stamp); } });
Runnable readTask = () -> {
    long stamp = myLock.readLock();
    try { System.out.println(map.get("foo")); sleep(1); }
    finally { myLock.unlockRead(stamp); } };
executor.submit(readTask); executor.submit(readTask);
//stop(executor);
```

Obtaining a read or write lock via readLock() or writeLock() returns a stamp which is later used for unlocking within the finally block. Keep in mind that stamped locks don't implement reentrant characteristics. Each call to lock returns a new stamp and blocks if no lock is available even if the same thread already holds a lock. So you have to pay particular attention not to run into deadlocks.

Just like in the previous ReadWriteLock example both read tasks have to wait until the write lock has been released. Then both read tasks print to the console simultaneously because multiple reads doesn't block each other as long as no write-lock is held.

2. Java 8 – Lambda Thread: Mutex vs. Semaphores

What is mutex and semaphore in Java ? What is the main difference?

"Semaphore can be counted, while mutex can only count to 1.
Suppose you have a thread running which accepts client connections. This thread can handle 10 clients simultaneously. Then each new client sets the semaphore until it reaches 10. When the Semaphore has 10 flags, then your thread won't accept new connections. Mutex are usually used for guarding stuff. Suppose your 10 clients can access multiple parts of the system. Then you can protect a part of the system with a mutex so when 1 client is connected to that sub-system, no one else should have access. You can use a Semaphore for this purpose too."

"This is not quite true. The same thread can enter the same mutex more than once, so a count needs to be maintained to ensure entries `& exits are balanced."

Unfortunately everyone has missed the most important difference between the semaphore and the mutex; the concept of "ownership".

Semaphores have no notion of ownership, this means that any thread can release a semaphore (this can lead to many problems in itself but can help with "death detection"). Whereas a mutex does have the concept of ownership (i.e. you can only release a mutex you have acquired). Ownership is incredibly important for safe programming of concurrent systems. I would always recommend using mutex in preference to a semaphore (but there are performance implications).

Mutexes also may support priority inheritance (which can help with the priority inversion problem) and recursion (eliminating one type of deadlock). It should also be pointed out that there are "binary" semaphores and "counting/general" semaphores. Java's semaphore is a counting semaphore and thus allows it to be initialized with a value greater than one (whereas, as pointed out, a mutex can only a conceptual count of one). The usefulness of this has been pointed out in other posts. So to summarize, unless you have multiple resources to manage, I would always recommend the mutex over the semaphore.

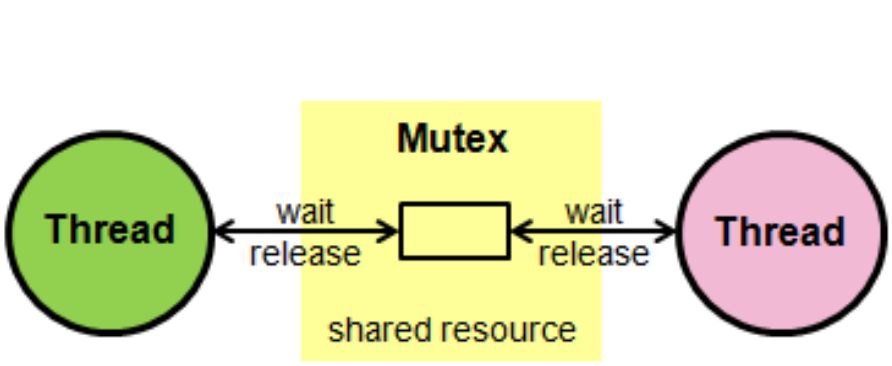
2. Java 8 – Lambda Thread: Mutex vs. Semaphores

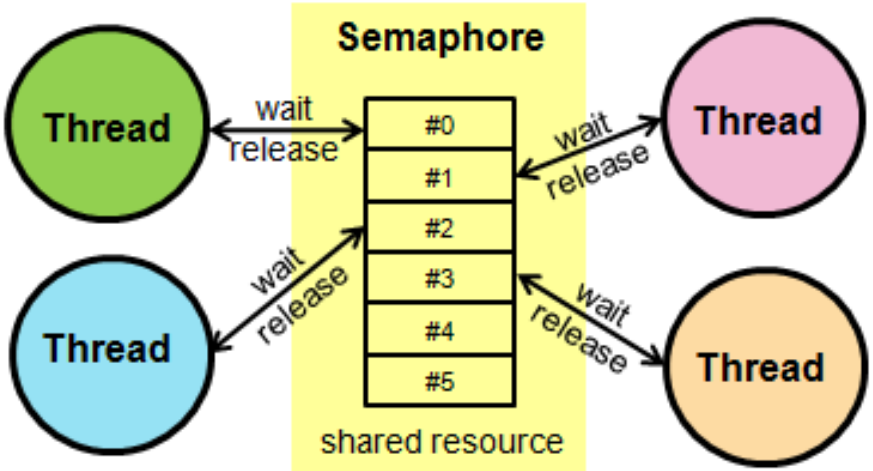
Java multi threads example to show you how to use Semaphore and Mutex to limit the number of threads to access resources.

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.

In addition to locks the Concurrency API also supports counting semaphores. Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits. This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.





2. Java 8 – Lambda Thread: Semaphores

Here's an example how to limit access to a long running task simulated by sleep(5):

```
ExecutorService executor = Executors.newFixedThreadPool(10);
Semaphore semaphore = new Semaphore(5);
Runnable longRunningTask = () -> {
 boolean permit = false;
 try {
    permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
    if (permit) { System.out.println("Semaphore acquired"); sleep(5); }
    else { System.out.println("Could not acquire semaphore"); }
 } catch (InterruptedException e) {
    throw new IllegalStateException(e);
 } finally { if (permit) { semaphore.release(); } } }

IntStream.range(0, 10).forEach( i ->executor.submit(longRunningTask) );
//stop(executor);
```

The executor can potentially run 10 tasks concurrently but we use a semaphore of size 5, thus limiting concurrent access to 5. It's important to use a try/finally block to properly release the semaphore even in case of exceptions.

2. Java 8 – Lambda Thread: Synchronization & Locks

Executing the previous code results in the following output:

```
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
```

The semaphores permits access to the actual long running operation simulated by sleep(5) up to a maximum of 5. Every subsequent call to tryAcquire() elapses the maximum wait timeout of one second, resulting in the appropriate console output that no semaphore could be acquired.

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

AtomicInteger

The package java.concurrent.atomic contains many useful classes to perform atomic operations. An operation is atomic when you can safely perform the operation in parallel on multiple threads without using the synchronized keyword or locks as shown in the previous sections. Internally, the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs. Those instructions usually are much faster than synchronizing via locks. So my advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently. Now let's pick one of the atomic classes for a few examples: AtomicInteger

```
AtomicInteger atomicInt = new AtomicInteger(0);
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 1000).forEach(i -> executor.submit(atomicInt::incrementAndGet));

//stop(executor); // helper method for stopping the Executor-Service
System.out.println(atomicInt.get()); // => 1000
```

By using AtomicInteger as a replacement for Integer we're able to increment the number concurrently in a thread-safe manor without synchronizing the access to the variable. The method incrementAndGet() is an atomic operation so we can safely call this method from multiple threads.

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

AtomicInteger supports various kinds of atomic operations. The method updateAndGet() accepts a lambda expression in order to perform arbitrary arithmetic operations upon the integer:

```
AtomicInteger atomicInt = new AtomicInteger(0);
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 1000).forEach(i -> {
   Runnable task = () -> atomicInt.updateAndGet(n -> n + 2);
   executor.submit(task); });
//stop(executor);
System.out.println(atomicInt.get()); // => 2000
```

The method accumulateAndGet() accepts another kind of lambda expression of type IntBinaryOperator.

Other useful atomic classes are:
AtomicBoolean, AtomicLong and AtomicReference.

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

LongAdder

The class LongAdder as an alternative to AtomicLong can be used to consecutively add values to a number.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
LongerAdder adder = new LongerAdder();
IntStream.range(0, 1000).forEach(i -> executor.submit(adder::increment));
// stop(executor);
System.out.println(adder.sumThenReset()); // => 1000
```

LongAdder provides methods add() and increment() just like the atomic number classes and is also thread-safe. But instead of summing up a single result this class maintains a set of variables internally to reduce contention over threads. The actual result can be retrieved by calling sum() or sumThenReset().
This class is usually preferable over atomic numbers when updates from multiple threads are more common than reads. This is often the case when capturing statistical data, e.g. you want to count the number of requests served on a web server. *The drawback of LongAdder is higher memory consumption because a set of variables is held in-memory.* For other details, please see **LongAccumulator** as well.

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

ConcurrentMap

The interface ConcurrentMap extends the map interface and defines one of the most useful concurrent collection types. Java 8 introduces functional programming by adding new methods to this interface.

In the next code snippets we use the following sample map to demonstrates those new methods:

```
ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

The method forEach() accepts a lambda expression of type BiConsumer with both the key and value of the map passed as parameters. It can be used as a replacement to for-each loops to iterate over the entries of the concurrent map. The iteration is performed sequentially on the current thread.

```
map.forEach((key, value) -> System.out.printf("%s = %s\n", key, value));
```

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

The method putIfAbsent() puts a new value into the map only if no value exists for the given key. At least for the ConcurrentHashMap implementation of this method is thread-safe just like put() so you don't have to synchronize when accessing the map concurrently from different threads:

```
String value = map.putIfAbsent("c3", "p1");
System.out.println(value); // p0
```

The method getOrDefault() returns the value for the given key. In case no entry exists for this key the passed default value is returned:

```
String value = map.getOrDefault("hi", "there");
System.out.println(value); // there
```

The method replaceAll() accepts a lambda expression of type BiFunction. BiFunctions take two parameters and return a single value. In this case the function is called with the key and the value of each map entry and returns a new value to be assigned for the current key:

```
map.replaceAll((key, value) -> "r2".equals(key) ? "d3" : value);
System.out.println(map.get("r2")); // d3
```

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

Instead of replacing all values of the map compute() let's us transform a single entry. The method accepts both the key to be computed and a bi-function to specify the transformation of the value.

```
map.compute("foo", (key, value) -> value + value);
System.out.println(map.get("foo")); // barbar
```

In addition to compute() two variants exist: computeIfAbsent() and computeIfPresent(). The functional parameters of these methods only get called if the key is absent or present respectively.

Finally, the method merge() can be utilized to unify a new value with an existing value in the map. Merge accepts a key, the new value to be merged into the existing entry and a bi-function to specify the merging behavior of both values:

```
map.merge("foo", "boo", (oldVal, newVal) -> newVal + " was " + oldVal);
System.out.println(map.get("foo")); // boo was foo
```

ConcurrentHashMap

All those methods above are part of the ConcurrentMap interface, thereby available to all implementations of that interface.

In addition the most important implementation ConcurrentHashMap has been further enhanced with a couple of new methods to perform parallel operations upon the map. Just like parallel streams those methods use a special ForkJoinPool available via ForkJoinPool.commonPool() in Java 8. This pool uses a preset parallelism which depends on the number of available cores. Four CPU cores are available on my machine which results in a parallelism of three:

```
System.out.println(ForkJoinPool.getCommonPoolParallelism()); // 3
```

This value can be decreased or increased by setting the following JVM parameter:
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

We use the same example map for demonstrating purposes but this time we work upon the concrete implementation ConcurrentHashMap instead of the interface ConcurrentMap, so we can access all public methods from this class:

```
ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

Java 8 introduces three kinds of parallel operations: forEach, search and reduce. Each of those operations are available in four forms accepting functions with keys, values, entries and key-value pair arguments.

All of those methods use a common first argument called parallelismThreshold. This threshold indicates the minimum collection size when the operation should be executed in parallel. E.g. if you pass a threshold of 500 and the actual size of the map is 499 the operation will be performed sequentially on a single thread. In the next examples we use a threshold of one to always force parallel execution for demonstrating purposes.

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

ForEach

The method forEach() is capable of iterating over the key-value pairs of the map in parallel. The lambda expression of type BiConsumer is called with the key and value of the current iteration step. In order to visualize parallel execution we print the current threads name to the console. Keep in mind that in my case the underlying ForkJoinPool uses up to a maximum of 3 threads.

```
map.forEach(1,
    (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n", key, value,
    Thread.currentThread().getName())
);
// key: r2; value: d2; thread: main
// key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1
// key: han; value: solo; thread: ForkJoinPool.commonPool-worker-2
// key: c3; value: p0; thread: main
```

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

Search

The method search() accepts a BiFunction returning a non-null search result for the current key-value pair or null if the current iteration doesn't match the desired search criteria. As soon as a non-null result is returned further processing is suppressed. Keep in mind that ConcurrentHashMap is unordered. The search function should not depend on the actual processing order of the map. If multiple entries of the map match the given search function the result may be non-deterministic.

```
String result = map.search(1, (key, value) -> {
    System.out.println(Thread.currentThread().getName());
    if ("foo".equals(key)) { return value; } return null; });
System.out.println("Result: " + result);
// ForkJoinPool.commonPool-worker-2
// main
// ForkJoinPool.commonPool-worker-3
// Result: bar
```

Here's another example searching solely on the values of the map:

```
String result = map.searchValues(1, value -> {
    System.out.println(Thread.currentThread().getName());
    if (value.length() > 3) { return value; } return null; });
System.out.println("Result: " + result);
// ForkJoinPool.commonPool-worker-2
// main
// main
// ForkJoinPool.commonPool-worker-1
// Result: solo
```

2. Java 8 – Lambda Thread: Atomic Variables & ConcurrentMap

Reduce

The method reduce() already known from Java 8 Streams accepts two lambda expressions of type BiFunction. The first function transforms each key-value pair into a single value of any type. The second function combines all those transformed values into a single result, ignoring any possible null values.

```
String result
    = map.reduce(1, (key, value) -> {
        System.out.println("Transform: " + Thread.currentThread().getName());
        return key + "=" + value;
    },
    (s1, s2) -> {
        System.out.println("Reduce: " + Thread.currentThread().getName());
        return s1 + ", " + s2;
    });
System.out.println("Result: " + result);
// Transform: ForkJoinPool.commonPool-worker-2
// Transform: main
// Transform: ForkJoinPool.commonPool-worker-3
// Reduce: ForkJoinPool.commonPool-worker-3
// Transform: main
// Reduce: main
// Reduce: main
// Result: r2=d2, c3=p0, han=solo, foo=bar
```

https://github.com/winterbe/java8-tutorial

Java 8 Multi-Threading
for easy sharing

Section Conclusions

Java 8 Lambda Expressions for the Multi-Threading programming:

- Concurrency API / Threads and Executors
- Future-Callable
- Concurrency API / Fork-Join mechanisms
- Synchronization and Locks/Mutex
- Mutex vs. Semaphores
- Atomic Variables
- ConcurrentMap and Thread-safe data-structures (Java Collection Framework)

Share knowledge, Empowering Minds

Communicate & Exchange Ideas

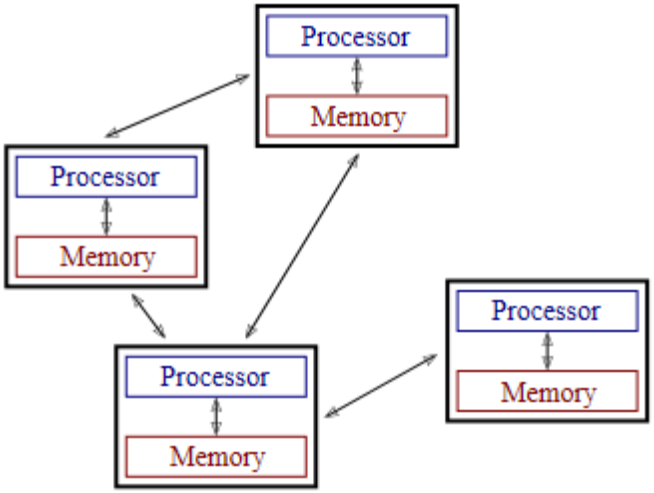
Questions & Answers!

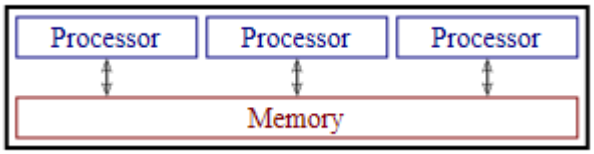
But wait…

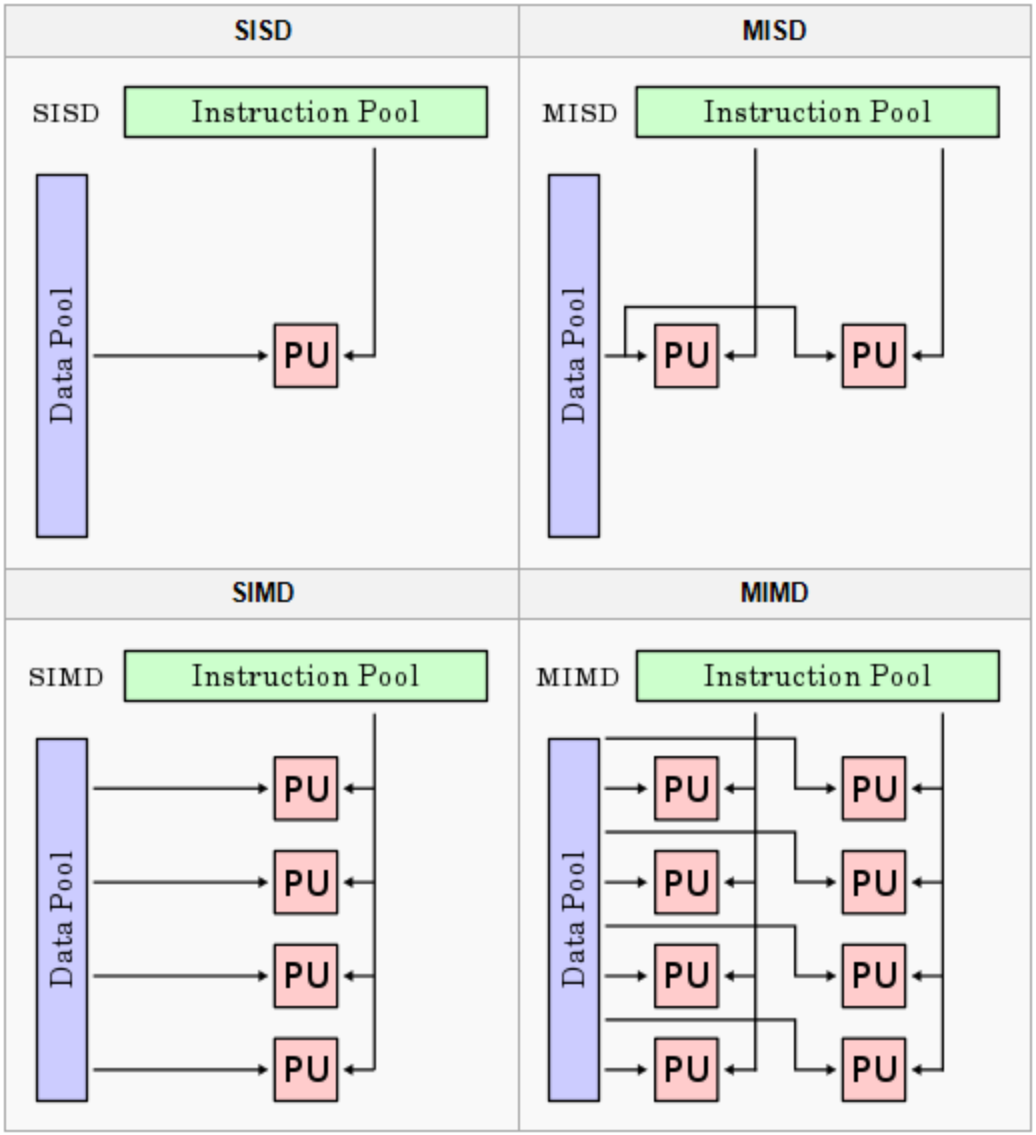
There's More!

Flynn Taxonomy
Parallel vs. Distributed Systems

Parallel vs. Distributed Computing / Algorithms







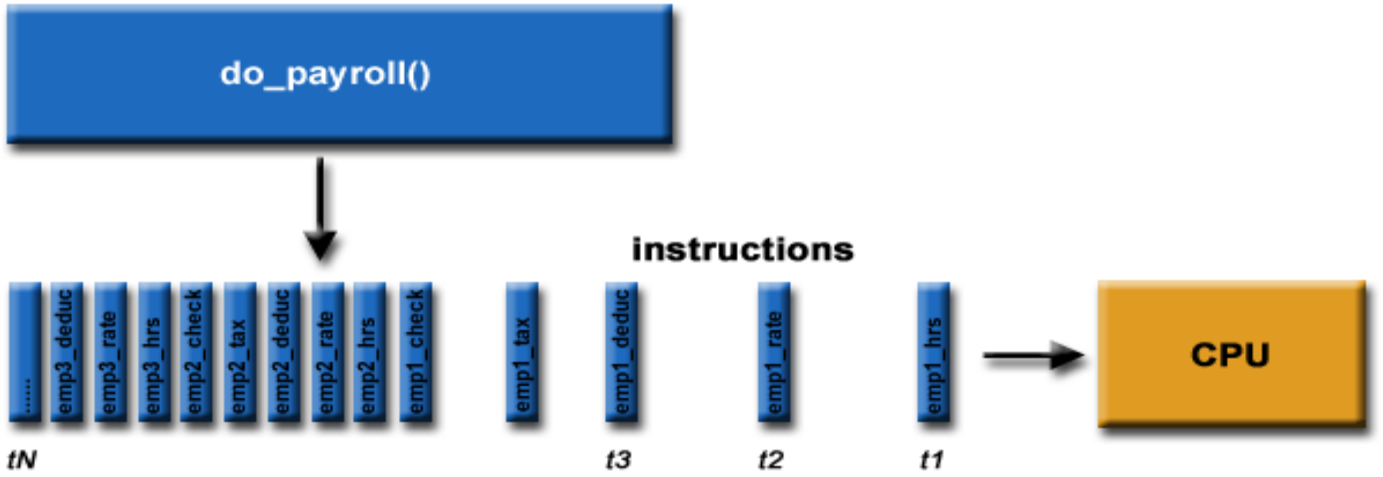
http://en.wikipedia.org/wiki/Distributed_computing

http://en.wikipedia.org/wiki/Flynn's_taxonomy

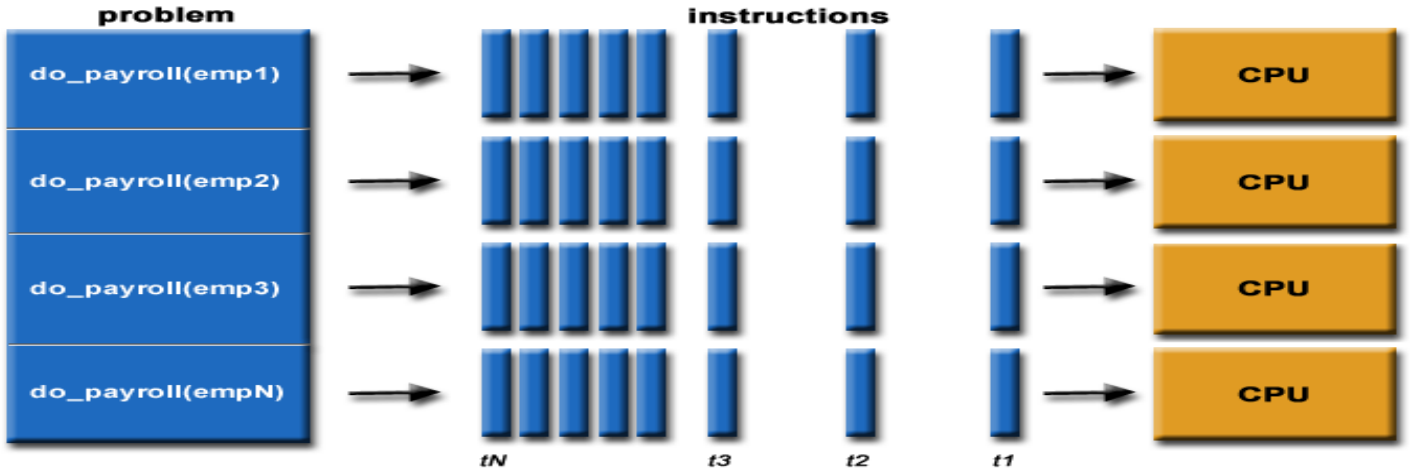
Parallel Computing Hint – OpenMP / OpenCL

https://computing.llnl.gov/tutorials/parallel_comp/

Serial Computing



Parallel Computing



Thanks!

Java Programming – Software App Development
End of Lecture 9 – Java SE Advanced
Multithreading