

Advanced Java Programming Course

# MultiThreading



Faculty of Information Technologies  
Industrial University of Ho Chi Minh City

# Session objectives

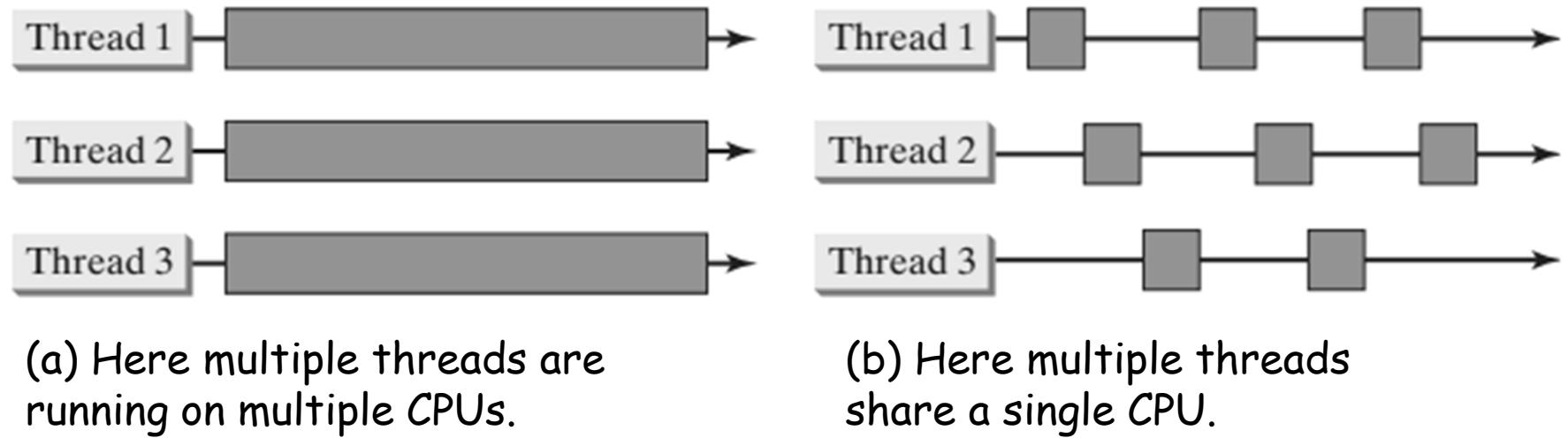
- ◊ Introduction
- ◊ Creating thread
- ◊ Callables and Futures
- ◊ Thread class and Thread behaviors
- ◊ Thread properties
- ◊ Thread pooling
- ◊ *Thread synchronization*
- ◊ Deadlocks
- ◊ *Thread and GUI*



# Introduction

- ◊ A program may consist of many tasks that can run concurrently.
- ◊ A thread is the flow of execution, from beginning to end, of a task.
- ◊ It provides the mechanism for running a task.
- ◊ With Java, you can launch multiple threads from a program concurrently.
- ◊ These threads can be executed simultaneously in multiprocessor systems

# Introduction



(a) Here multiple threads are running on multiple CPUs.

(b) Here multiple threads share a single CPU.

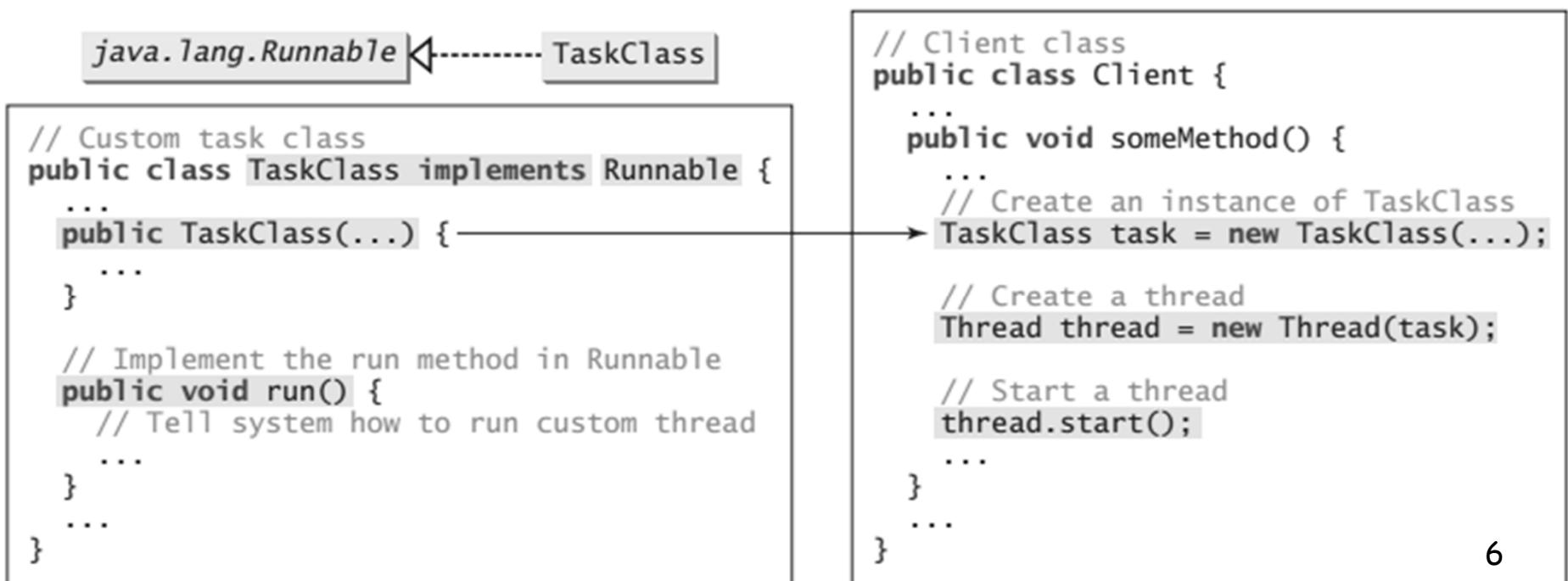
- ◊ In single-processor systems, the multiple threads share CPU time known as time sharing, and the operating system is responsible for scheduling and allocating resources to them.

# Java programs execution and Thread

- ◊ When Java programs execute, there is always one thread running and that is the main thread.
  - ✧ It is this thread from which child threads are created.
  - ✧ Program is terminated when main thread stops execution.(\*)
  - ✧ Main thread can be controlled through Thread objects.
  - ✧ Reference of the main thread can be obtained by calling the `currentThread()` method of the Thread class.

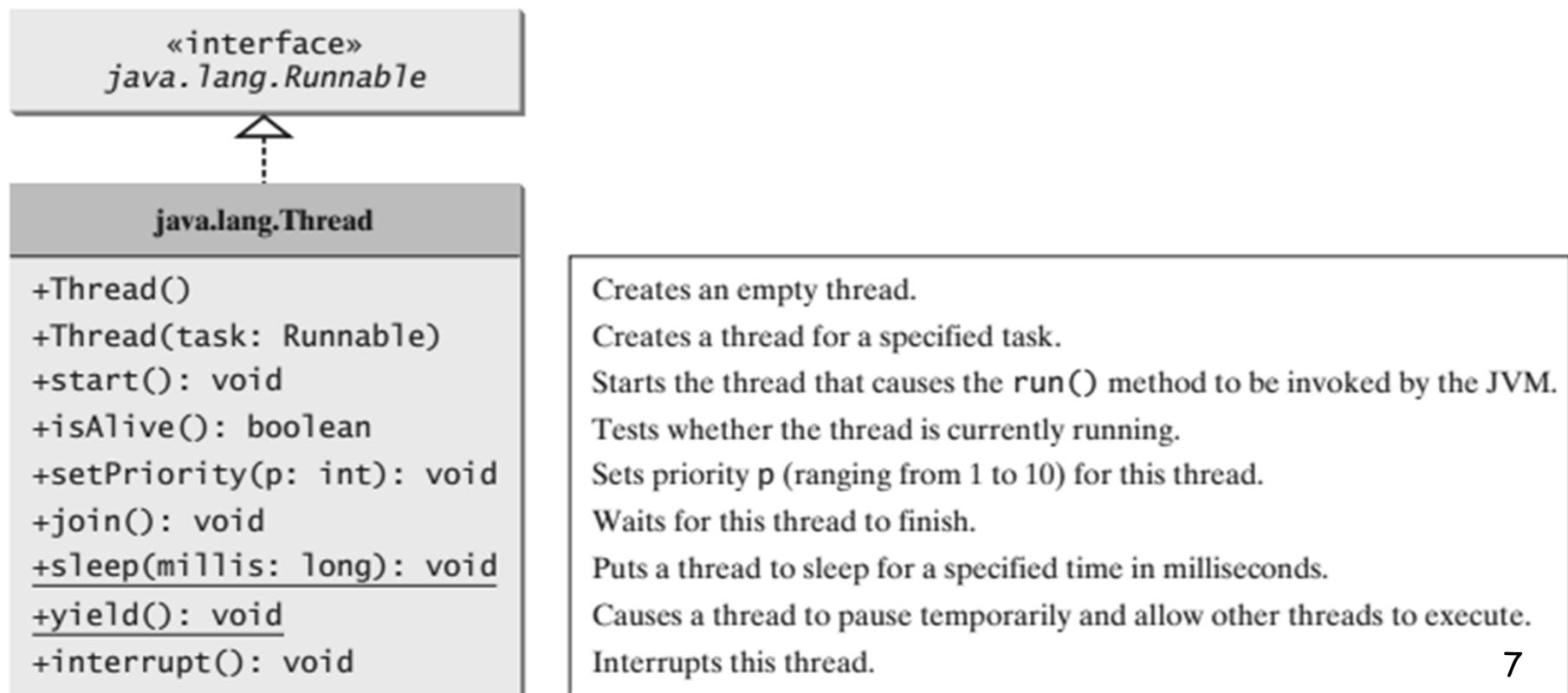
# Creating Tasks and Threads

- ◊ Tasks are objects. To create tasks, you have to first define a class for tasks. A task class must implement the *Runnable* interface
- ◊ You need to implement *run* method to tell the system how your thread is going to run



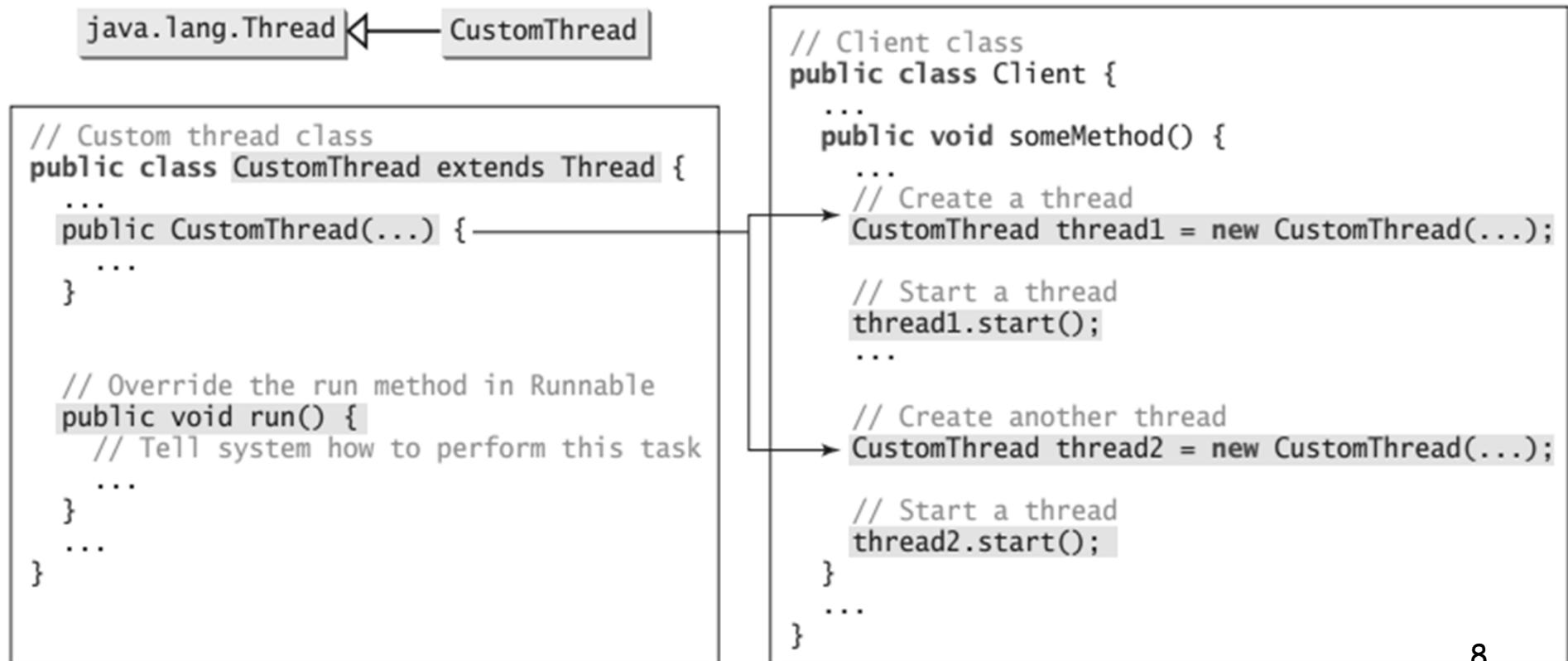
# Thread class

- ◊ The Thread class contains the constructors for *creating* threads for tasks, and the methods for *controlling* threads.



# Another way to create thread

This approach is **not recommended**, because it mixes the task and the mechanism of running the task. Separating the task from the thread is a preferred design.



# Callables and Futures

## Introduction

- ◊ A Runnable encapsulates a task that runs asynchronously; you can think of it as an asynchronous method with no parameters and no return value.
- ◊ Drawback of Runnable:
  - ✧ Cannot return any type (of run method)
  - ✧ No parameters (of run method)
  - ✧ Processing exception locally.
- ◊ So, we need another mechanic: Callable
- ◊ The Callable interface is a parameterized type, with a single method call:

```
1  public interface Runnable {  
2      public abstract void run();  
3  }
```

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

## Callables and Futures (cont)

### Future object

- ◊ A Future object holds the result of an asynchronous computation.
- ◊ You use a Future object so that you can start a computation, give the result to someone, and forget about it.
- ◊ The owner of the Future object can obtain the result when it is ready.

```
1 public interface Future<V>
2 {
3     V get() throws . . .;
4     V get(long timeout, TimeUnit unit) throws . . .;
5     void cancel(boolean mayInterrupt);
6     boolean isCancelled();
7     boolean isDone();
8 }
```

# Callables and Futures

## Example

- ◊ The FutureTask wrapper is a convenient mechanism for turning a Callable into both a Future and a Runnable it implements both interfaces.

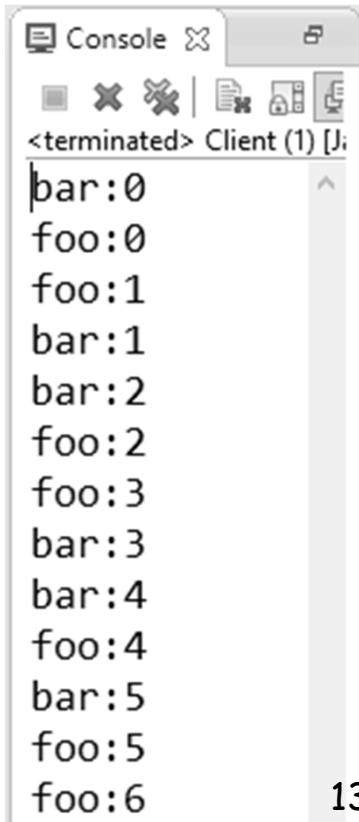
```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // it's a Runnable
t.start();
. . .
Integer result = task.get(); // it's a Future
```

```
1 ComputationTask.java
2
3 import java.util.concurrent.Callable;
4
5 public class ComputationTask implements Callable<Long>{
6     private String taskName;
7     public ComputationTask(String taskName) {
8         this.taskName = taskName;
9     }
10
11    @Override
12    public Long call() throws Exception {
13        Long result=0L;
14        for (int i = 0; i < 1000; i++) {
15            result+=i;//simple for testing purpose
16            System.out.println(taskName + " #" + i);
17            Thread.sleep(10);
18        }
19        return result;
20    }
21 }
22
23 public static void main(String[] args) throws Exception{
24     Callable<Long> call=new ComputationTask("long-last-computaion");
25     FutureTask<Long> task = new FutureTask<>(call);
26     new Thread(task).start();
27
28     //Waits if necessary for the computation to complete,
29     //and then retrieves its result.
30     long result=task.get();
31     System.out.println("Result:" + result);
32 }
33 }
```

## Thread behaviors - The *sleep()* method

- ◊ The `sleep(long millis)` method puts the thread to sleep for the specified time in milliseconds to allow other threads to execute.

```
@Override  
public void run() {  
    try {  
        for (int i = 0; i < times; i++) {  
            System.out.println(task + ":" + i);  
            Thread.sleep(10);  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```



The screenshot shows a Java IDE's console window titled "Console". The title bar includes standard icons for close, minimize, and maximize. Below the title bar, there are tabs for "Console" and "File". The main area of the window displays the output of a program. The output consists of alternating lines of "bar" and "foo" followed by a colon and an integer value, starting from 0 and increasing by 1 up to 6. The lines are separated by newlines. The window has a scroll bar on the right side.

```
bar:0  
foo:0  
foo:1  
bar:1  
bar:2  
foo:2  
foo:3  
bar:3  
bar:4  
foo:4  
bar:5  
foo:5  
foo:6
```

## Thread behaviors - The *yield()* method

- ◊ the *yield()* method causes the currently executing thread to yield. If there are other runnable threads with a priority at least as high as the priority of this thread, they will be scheduled next.

```
@Override  
public void run() {  
    try {  
        for (int i = 0; i < times; i++) {  
            System.out.println(task + ":" + i);  
            Thread.yield();  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

## Thread behaviors - The `join()` method

- ◊ Causes the current thread to wait until the thread on which it is called terminates.
- ◊ Allows specifying the maximum amount of time that the program should wait for the particular thread to terminate.
- ◊ It throws `InterruptedException` if another thread interrupts it.
- ◊ The calling thread waits until the specified thread terminates.

# Thread with join

The screenshot shows an IDE interface with three tabs at the top: Client.java, YourTask.java, and OtherTask.java. The OtherTask.java tab is active, displaying the following Java code:

```
1 package prepare.thread;
2
3 public class OtherTask implements Runnable{
4     @Override
5     public void run() {
6         try {
7             Thread t=new Thread(
8                 new YourTask("Your Thread", 8));
9             t.start();
10            for (int i = 0; i < 10; i++) {
11                System.out.println("Other Thread #"+i);
12                if(i==7)
13                    t.join(); //join t with current thread
14            }
15        } catch (InterruptedException e) {
16            e.printStackTrace();
17        }
18    }
19 }
```

To the right of the code is a Console window showing the output of the program. The output consists of two sets of interleaved threads. The first set, labeled "Other Thread", prints numbers from 0 to 9. The second set, labeled "Your Thread", also prints numbers from 0 to 9. The "Your Thread" set starts after the "Other Thread" set reaches number 7, indicating that the main thread has joined the thread that was running "Your Thread".

```
Console >
<terminated> Client (1) [Java Application]
Other Thread #0
Other Thread #1
Other Thread #2
Other Thread #3
Other Thread #4
Other Thread #5
Other Thread #6
Other Thread #7
Your Thread:0
Your Thread:1
Your Thread:2
Your Thread:3
Your Thread:4
Your Thread:5
Your Thread:6
Your Thread:7
Other Thread #8
Other Thread #9
```

# Interrupting threads

- ◊ There is no longer a way to force a thread to terminate.
- ◊ The *interrupt()* method can be used to request termination of a thread.
- ◊ Checking one thread is interrupted:

*Thread.currentThread().isInterrupted()*

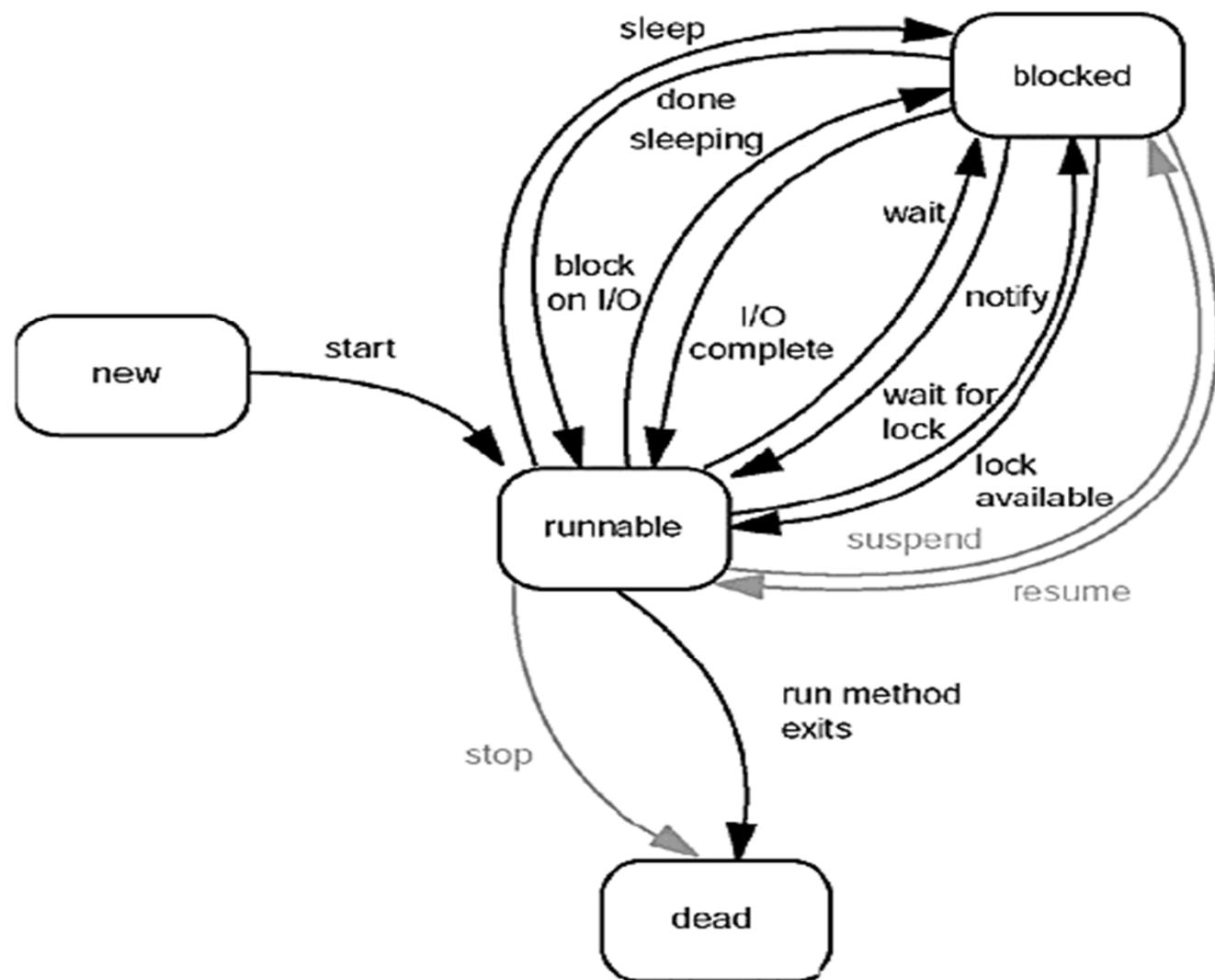
- ◊ If a thread is blocked, it cannot check the interrupted status.  
This is where the *InterruptedException* comes in.

# Interrupting threads (cont.)

```
public void run()
{
    try
    {
        . . .
        while (more work to do)
        {
            do more work
        }
    }
    catch (InterruptedException exception)
    {
        // thread was interrupted during sleep or wait
    }
    finally
    {
        cleanup, if required
    }
    // exit run method and terminate thread
}
```

## Pattern for interrupting an thread

# Thread states



# Managing threads: Priorities (1)

- ◊ In Java, thread scheduler can use the thread priorities in the form of integer value to each of its thread to determine the execution schedule of threads .
- ◊ Thread gets the ready-to-run state according to their priorities. The thread scheduler provides the CPU time to thread of highest priority during ready-to-run state.

Constant	Description
Thread.MAX_PRIORITY	The maximum priority of any thread (10)
Thread.MIN_PRIORITY	The minimum priority of any thread (1)
Thread.NORM_PRIORITY	The normal priority of any thread (5)

## Managing threads: Priorities (2)

- ◊ When a Java thread is created, it inherits its priority from the thread that created it.
- ◊ At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution.
- ◊ In the implementation of threading scheduler usually applies one of the two following strategies:
  - ✧ **Preemptive scheduling:** If the new thread has a higher priority than current running thread leaves the runnable state and higher priority thread enter to the runnable state.
  - ✧ **Time-Sliced (Round-Robin) Scheduling:** A running thread is allowed to execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

## Managing threads: Priorities (3)

- ◊ The highest-priority runnable thread keeps running until:
  - ✧ It yields by calling the `yield()` method
  - ✧ It ceases to be runnable (either by dying or by entering the blocked state)
  - ✧ A higher-priority thread has become runnable.
- ◊ We can use follow method to set priority of Thread

```
void setPriority(int newPriority)
```

# Daemon threads

- ◊ Two types of threads in Java:
  1. User threads:
    - Created by the user
  2. Daemon threads:
    - Threads that work in the background providing service to other threads (e.g. - the garbage collector thread)
- ◊ When user thread exits, JVM checks to find out if any other thread is running.
  - ✧ If there are, it will schedule the next thread.
  - ✧ If the only executing threads are daemon threads, it exits.
- ◊ We can set a thread to be a Daemon if we do not want the main program to wait until a thread ends.

```
2 public class DaemonThread extends Thread {  
3     public void run() {  
4         System.out.println("Entering run method");  
5         try {  
6             System.out.println("In run Method: currentThread() is"  
7                     + Thread.currentThread());  
8             while (true) {  
9                 try {  
10                     Thread.sleep(500);  
11                 } catch (InterruptedException x) {  
12                     }  
13                     System.out.println("In run method: woke up again");  
14                 }  
15             } finally {  
16                 System.out.println("Leaving run Method");  
17             }  
18         }  
19     public static void main(String[] args) throws Exception{  
20         System.out.println("Entering main Method");  
21         DaemonThread t = new DaemonThread();  
22         t.setDaemon(true);  
23         t.start();  
24         Thread.sleep(3000);  
25         System.out.println("Leaving main method");  
26     }  
27 }
```

# Thread Pools

- ◊ A thread pool is ideal to manage the number of tasks executing concurrently.
- ◊ Java provides the *Executor* interface for executing tasks in a thread pool and the *ExecutorService* interface for managing and controlling tasks

```
java.util.concurrent.Executors  
+newFixedThreadPool(numberOfThreads:  
    int): ExecutorService  
  
+newCachedThreadPool():  
    ExecutorService
```

```
«interface»  
java.util.concurrent.ExecutorService  
  
+shutdown(): void  
  
+shutdownNow(): List<Runnable>  
  
+isShutdown(): boolean  
+isTerminated(): boolean
```

## Thread Pools - The Executor interface (1/2)

- ◊ To create an Executor object, use the static methods in the Executor class.
- ◊ The newFixedThreadPool(int) method creates a fixed number of threads in a pool.
  - ✧ If a thread completes executing a task, it can be reused to execute another task.
  - ✧ If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it
  - ✧ If all the threads in the pool are not idle and there are tasks waiting for execution.

## Thread Pools - The Executor interface (2/2)

- ◊ The newCachedThreadPool() method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution.
  - ✧ A thread in a cached pool will be terminated if it has not been used for 60 seconds.
  - ✧ A cached pool is efficient for many short tasks.

# Thread Pools - The ExecutorService interface

- ◊ The shutdown() method shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.
- ◊ The shutdownNow() method shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
- ◊ The isShutdown() method returns true if the executor has been shut down.
- ◊ The isTerminated() method returns true if all tasks in the pool are terminated

# Thread Synchronization

- ◊ What happens if two threads have access to the same object and each calls a method that modifies the state of the object?
- ◊ In such a case, data may become inconsistent.
- ◊ Situation is often called a *race condition*.
- ◊ To avoid simultaneous access of a shared object by multiple threads, you must learn how to synchronize the access.

# Thread Synchronization

- ◊ Thread Communication Without Synchronization

View follow example: UnsynchBankTest.java

- ◊ There are some things wrong in this Bank.

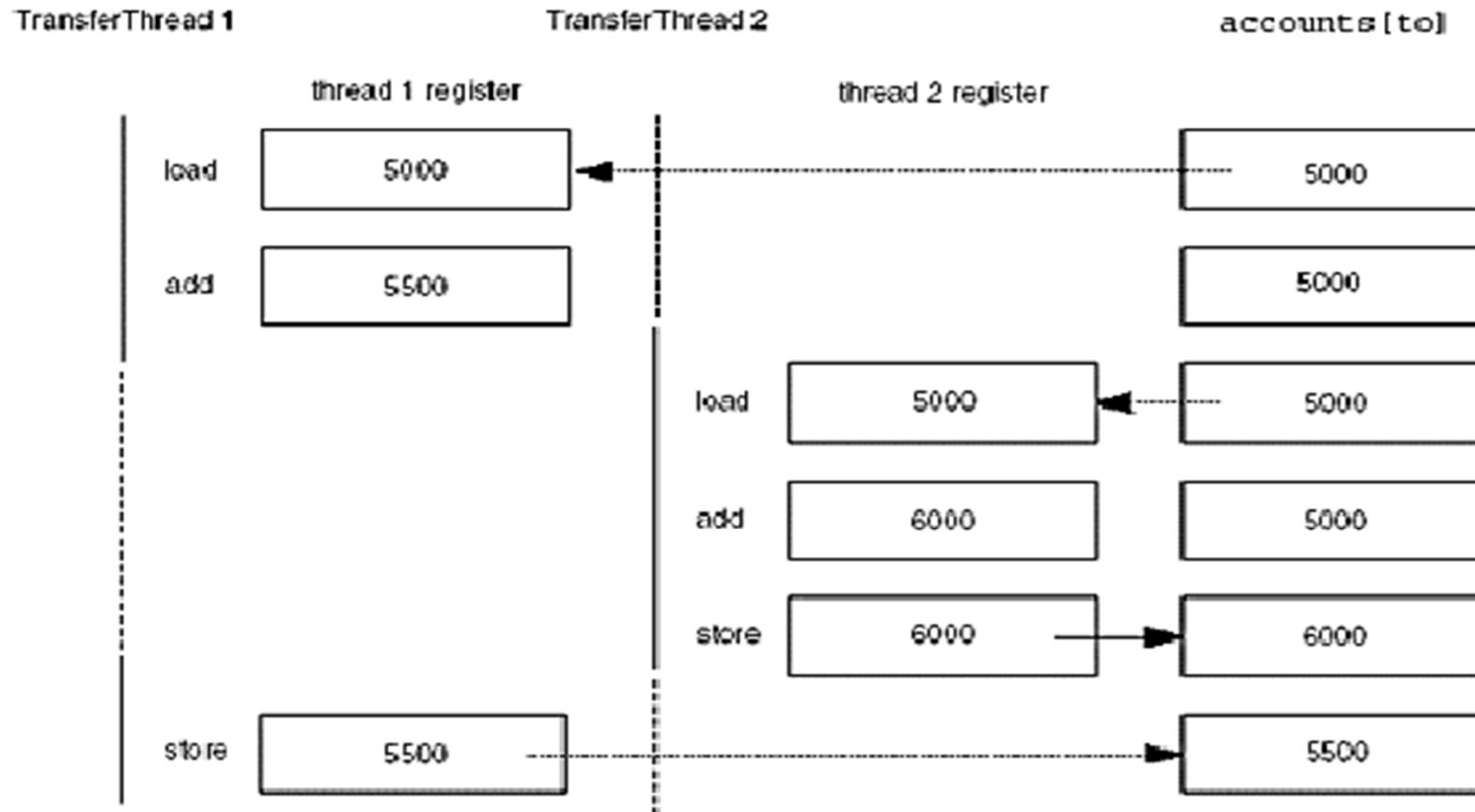
- ◊ The Race Condition Explained:

- ✧ The problem is that these are not atomic operations. View follow figure

- ✧ The real problem is that the work of the transfer method can be interrupted in the middle. If we could ensure that the method runs to completion before the thread loses control, then the state of the bank account object would not be corrupted.

# Thread Synchronization

## Simultaneous access by two threads



# Thread Synchronization

- ◊ Synchronization is based on the concept of monitor.
  - ✧ A monitor is an object that is used as a mutually exclusive lock.
- ◊ Only one thread can enter a monitor:
  - ✧ When one thread enters the monitor, it means that the thread has acquired a lock
  - ✧ All other threads must wait till that thread exits the monitor.
- ◊ For a thread to enter the monitor of an object:
  - ✧ The programmer may invoke a method created using the synchronized keyword (implicit synchronize).
  - ✧ Or using explicit lock objects.

# Thread Synchronization - 1<sup>st</sup> approach

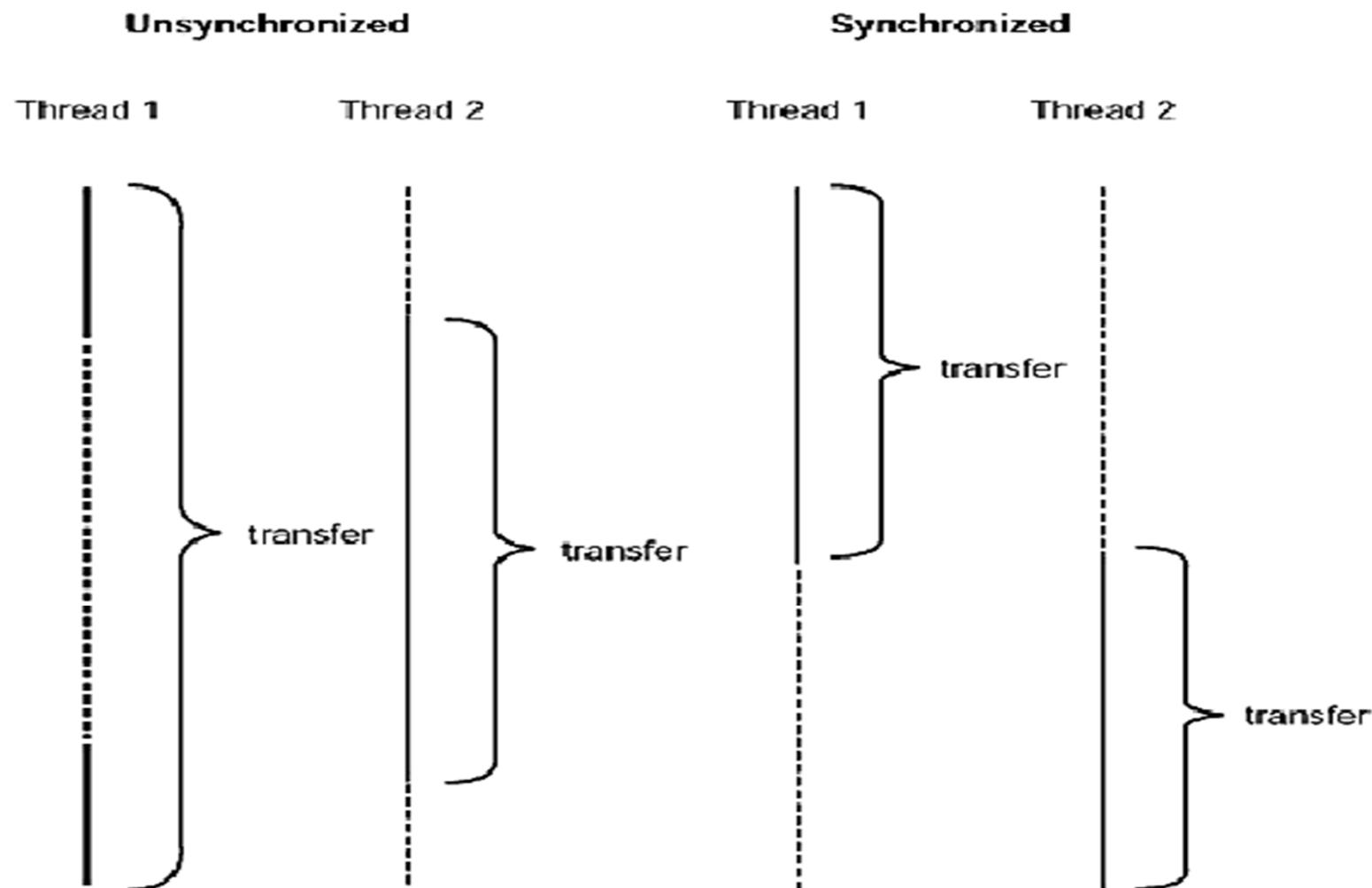
## ◊ Concurrency mechanism:

- ✧ Simply tag any operation that should not be interrupted as *synchronized*, for example :

```
public synchronized void transfer(int from, int to,int amount)
```

- ✧ When one thread calls a *synchronized* method, it is guaranteed that the method will finish before another thread can execute any *synchronized* method on the same object.

# Comparison of unsynchronized and synchronized threads



## Thread Synchronization - 1<sup>st</sup> approach (cont.) how it work?

- ◊ When a thread calls a synchronized method, the object becomes "locked."
- ◊ Periodically, the thread scheduler activates the threads that are waiting for the lock to open.
- ◊ Other threads are still free to call unsynchronized methods on a locked object.
- ◊ When a thread leaves a synchronized method by throwing an exception, it still relinquishes the object lock.
- ◊ If a thread owns the lock of an object and it calls another synchronized method of the same object, then that method is automatically granted access. The thread only relinquishes the lock when it exits the last synchronized method.

## Thread Synchronization - 1<sup>st</sup> approach (cont.)

### The 'wait - notify' mechanism

- ✧ This mechanism ensures that there is a smooth transition of a particular resource between two competitive threads.
- ✧ It also oversees the condition in a program where one thread is:
  - Allowed to wait for the lock.
  - Notified to end its waiting state and get the lock
- ✧ When a thread executes a call to *wait*, it surrenders the object lock and enters a wait list for that object.
- ✧ To remove a thread from the wait list, some other thread must make a call to *notifyAll* or *notify*, on the same object.

# Thread Synchronization – 1<sup>st</sup> approach (cont.)

## The ‘*wait - notify*’ mechanism (cont.)

### **notify()**

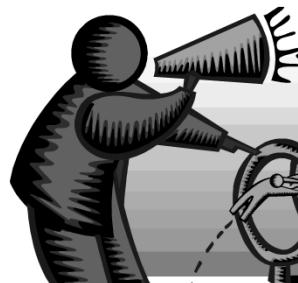


**notify()**  
wakes up or  
notifies the  
first thread.

First thread



### **notifyAll()**



**notifyAll()**  
wakes up or  
notifies all the  
threads that  
called `wait()` on  
the same object.

Thread 1



Thread 2



Thread 3



# Thread Synchronization - 1<sup>st</sup> approach (cont.)

## The 'wait - notify' mechanism example

```
class MyQueue {  
    int n;  
    synchronized int get() {  
        System.out.println("Got: " + n);  
        return n;  
    }  
    synchronized void put(int n) {  
        this.n = n;  
        System.out.println("Put: " + n);  
    }  
}
```

```
class Producer implements Runnable {  
    MyQueue q;  
    Producer(MyQueue q) {  
        this.q = q;  
        new Thread(this, "Producer").start();  
    }  
    public void run() {  
        int i = 0;  
        while(true) { q.put(i++);}  
    }  
}
```

```
class Consumer implements Runnable {  
    MyQueue q;  
    Consumer(MyQueue q) {  
        this.q = q;  
        new Thread(this, "Consumer").start();  
    }  
    public void run() {  
        while(true) { q.get();}  
    }  
}
```

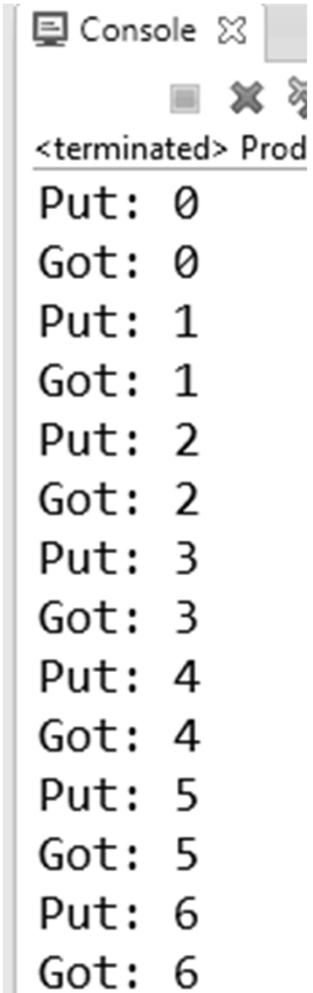
```
public class Producer_Consumer_Demo {  
    public static void main(String args[]){  
        MyQueue q = new MyQueue();  
        new Producer(q);  
        new Consumer(q);  
    }  
}
```

An incorrect implementation of a producer and consumer

# Thread Synchronization - 1<sup>st</sup> approach (cont.)

## The 'wait - notify' mechanism example

```
class MyQueue {  
    int n;  
    boolean valueSet = false;  
    synchronized int get() {  
        if(!valueSet)  
            try { wait(); } catch(InterruptedException e) {}  
        System.out.println("Got: " + n);  
        valueSet = false;  
        notify();  
        return n;  
    }  
    synchronized void put(int n) {  
        if(valueSet)  
            try { wait(); } catch(InterruptedException e) {}  
        this.n = n;  
        valueSet = true;  
        System.out.println("Put: " + n);  
        notify();  
    }  
}
```



Console <terminated> Prod  
Put: 0  
Got: 0  
Put: 1  
Got: 1  
Put: 2  
Got: 2  
Put: 3  
Got: 3  
Put: 4  
Got: 4  
Put: 5  
Got: 5  
Put: 6  
Got: 6

# Thread Synchronization - 1<sup>st</sup> approach (cont.)

## Synchronized Blocks

- ✧ Syntax : **synchronized** (object) {  
                  //do your work  
}

- ✧ Example :

```
1. public void run()  
2. {  
3.     //. . .  
4.     synchronized (bank) // lock the bank object  
5.     {  
6.         if (bank.getBalance(from) >= amount)  
7.             bank.transfer(from, to, amount);  
8.     }  
9.     //. . .  
10. }
```

## Thread Synchronization - 1<sup>st</sup> approach (cont.)

### Synchronized static method

- ✧ If one thread calls a synchronized static method of a class, all synchronized static methods of the class are blocked until the first call returns.
- ✧ Example :

```
public static synchronized Singleton getInstance()
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Lock Objects

- ◊ The basic outline for protecting a code block with a ReentrantLock is:

```
myLock.lock(); // a ReentrantLock object
try{
    //critical section
}
finally{
    myLock.unlock();
    // make sure the lock is unlocked even if an exception is thrown
}
//...
private Lock bankLock = new ReentrantLock();
// ReentrantLock implements the Lock interface
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Lock Objects (cont.)

- ◊ This construct guarantees that only one thread at a time can enter the critical section.
- ◊ As soon as one thread locks the lock object, no other thread can get past the lock statement.
- ◊ When other threads call lock, they are blocked until the first thread unlocks the lock object.

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Lock Objects (cont.)

- ◊ Imagine we have a very simple case where we need to synchronize access to a pair of variables. One is a simple value and another is derived based on some lengthy calculation.

```
01 public class Calculator {  
02     private int calculatedValue;  
03     private int value;  
04  
05     public synchronized void calculate(int value) {  
06         this.value = value;  
07         this.calculatedValue = doMySlowCalculation(value);  
08     }  
09  
10    public synchronized int getCalculatedValue() {  
11        return calculatedValue;  
12    }  
13  
14    public synchronized int getValue() {  
15        return value;  
16    }  
17 }
```

Simple, but if we have a lot of contention or if we perform a lot of reads and few writes?

## Using of `ReadWriteLock`

```
01 public class Calculator {  
02     private int calculatedValue;  
03     private int value;  
04     private ReadWriteLock lock = new ReentrantReadWriteLock();  
05  
06     public void calculate(int value) {  
07         lock.writeLock().lock();  
08         try {  
09             this.value = value;  
10             this.calculatedValue = doMySlowCalculation(value);  
11         } finally {  
12             lock.writeLock().unlock();  
13         }  
14     }  
15  
16     public int getCalculatedValue() {  
17         lock.readLock().lock();  
18         try {  
19             return calculatedValue;  
20         } finally {  
21             lock.readLock().unlock();  
22         }  
23     }  
24  
25     public int getValue() {  
26         lock.readLock().lock();  
27         try {  
28             return value;  
29         } finally {  
30             lock.readLock().unlock();  
31         }  
32     }  
33 }
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects

- ◊ See code below:

```
if (bank.getBalance(from) >= amount)  
    bank.transfer(from, to, amount);
```

- ◊ It is entirely possible that the current thread will be deactivated between the successful outcome of the test and the call to transfer:

```
if (bank.getBalance(from) >= amount)  
    // thread might be deactivated at this point  
    bank.transfer(from, to, amount);
```

- ◊ By the time the thread is running again, the account balance may have fallen below the withdrawal amount.

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects (cont.)

- ◊ You must make sure that the thread cannot be interrupted

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // wait
            . . .
        }
        // transfer funds
        . . .
    }
    finally
    {
        bankLock.unlock();
    }
}
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects (cont.)

- ◊ What do we do when there is not enough money in the account?
- ◊ We wait until some other thread has added funds. But this thread has just gained exclusive access to the bankLock, so no other thread has a chance to make a deposit.
- ◊ The solution is : condition objects

```
class Bank
{
    public Bank()
    {
        . . .
        sufficientFunds = bankLock.newCondition();
    }
    . . .
    private Condition sufficientFunds;
}
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects (cont.)

- ◊ If the Transfer method finds that sufficient funds are not available, it calls

```
sufficientFunds.await();
```

=>The current thread is now blocked and gives up the lock. This lets in another thread that can, we hope, increase the account balance

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects (cont.)

- ◊ There is an essential difference between a thread that is waiting to acquire a lock and a thread that has called await.
- ◊ Once a thread calls the await method, it enters a wait set for that condition.
- ◊ Thread is not unblocked when the lock is available.
- ◊ Instead, it stays blocked until another thread has called the signalAll method on the same condition.
- ◊ The signalAll method call unblocks all threads that are waiting for the condition.
- ◊ When the threads are removed from the wait set, they are again runnable and the scheduler will eventually activate them again.

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Condition Objects (cont.)

```
1 void transfer(int from, int to, int amount)
2 {
3     bankLock.lock();
4     try
5     {
6         while (accounts[from] < amount)
7             sufficientFunds.await();
8         // transfer funds
9         .
10        sufficientFunds.signalAll();
11    }
12    finally
13    {
14        bankLock.unlock();
15    }
16 }
```

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Fairness

- ◊ A fair lock favors the thread that has been waiting for the longest time.
- ◊ By default, locks are not required to be fair.
- ◊ You can specify that you want a fair locking policy:

```
Lock fairLock = new ReentrantLock(true);
```

- ◊ Fair locks are a lot slower than regular locks.
- ◊ You should only enable fair locking if you have a specific reason why fairness is essential for your problem.

## Thread Synchronization - 2<sup>nd</sup> approach (cont.)

### Lock Testing and Timeouts

- ◊ The `tryLock` method tries to acquire a lock and returns true if it was successful. Otherwise, it immediately returns false.

```
1  if (myLock.tryLock())
2      // now the thread owns the lock
3      try { . . . }
4      finally { myLock.unlock(); }
5  else
6      // do something else
```

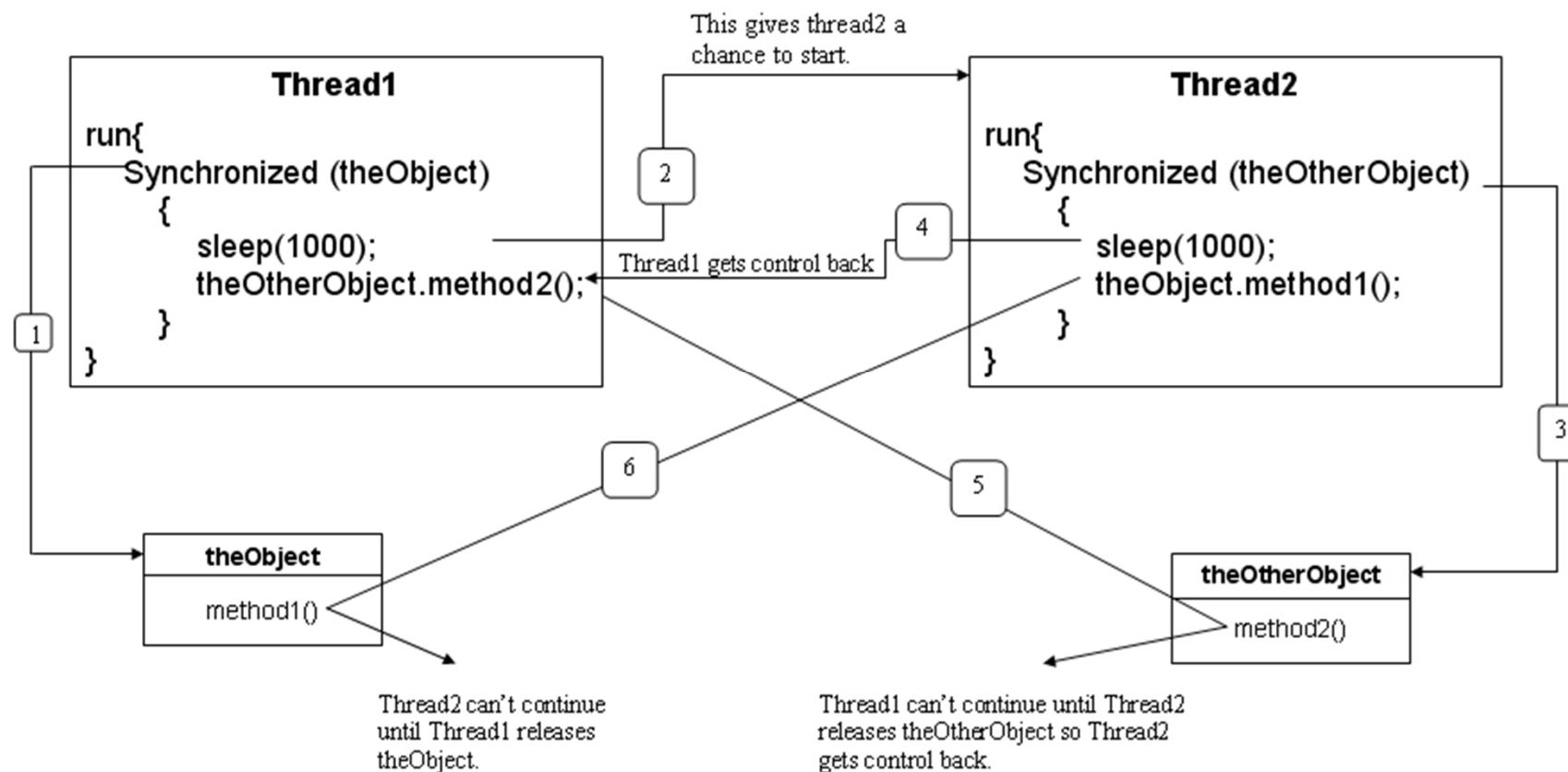
- ◊ You can call `tryLock` with a timeout parameter, like this:

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

- ◊ `TimeUnit` is an enumeration with values `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, and `NANOSECONDS`.

# Deadlocks

## Analyzing following situation



## Deadlocks(cont.)

- ◊ If all threads in an application are blocked. The system has deadlocked.
- ◊ Unfortunately, there is nothing in the Java programming language to avoid or break these deadlocks.
- ◊ You must design your threads to ensure that a deadlock situation cannot occur.
- ◊ Notify/notifyAll method can unblock thread(s).

# GUI Event Dispatch Thread

- ◊ GUI event handling and painting code executes on a special thread called the *event dispatch thread*.
- ◊ Most of Swing methods are not thread-safe. Invoking them from multiple threads may cause conflicts.
- ◊ You need to run the code in the event dispatch thread to avoid possible conflicts.
- ◊ You can use the static methods, *invokeLater* and *invokeAndWait* in the `javax.swing.SwingUtilities` class to run the code in the event dispatch thread.

## Thread and Swing - SwingWorker

- ◊ All Swing GUI events are processed in a single event dispatch thread.
- ◊ If an event requires a long time to process, the thread cannot attend to other tasks in the queue.
- ◊ To solve this problem, you should run the time-consuming task for processing the event in a separate thread.
- ◊ You can define a task class that extends SwingWorker, run the time-consuming task and update the GUI using the results produced from the task.

# Thread and Swing - SwingWorker

«interface»  
java.lang.Runnable



javax.swing.SwingWorker<T, V>

```
#doInBackground(): T  
#done(): void  
  
+execute(): void  
+get(): T  
  
+isDone(): boolean  
+cancel(): boolean  
#publish(data V...): void  
  
#process(data: java.util.List<V>): void  
  
#setProgress(progress: int): void  
#getProgress(): void
```

Performs the task and returns a result of type T.  
Executed on the event dispatch thread after `doInBackground` is finished.  
Schedules this `SwingWorker` for execution on a worker thread.  
Waits if necessary for the computation to complete, and then retrieves its result (i.e., the result returned `doInBackground`).  
Returns true if this task is completed.  
Attempts to cancel this task.  
Sends data for processing by the `process` method. This method is to be used from inside `doInBackground` to deliver intermediate results for processing on the event dispatch thread inside the `process` method. Note that `V...` denotes variant arguments.  
Receives data from the `publish` method asynchronously on the event dispatch thread.  
Sets the progress bound property. The value should be from 0 to 100.  
Returns the progress bound property.

# Thread and Swing - SwingWorker

- ◊ Since the task is run on a separate thread, you can continue to use the GUI.
- ◊ If the task is executed on the event dispatch thread, the GUI is frozen

```
//...
computeWithSwingWorkerButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Execute SwingWorker
        new LongTimeComputing(resultTextField).execute();
    }
});
```



# *That's all for this session!*

- ◊ Thread is a special and interesting property of Java
  - ✧ For building a single program to perform more than one task at the same time (*multithreading program*)
  - ✧ Thread synchronization
- ◊ Other advanced technique to use multithreading is Callable
  - ✧ The best technique to handling multithreading.

## **Thank you all for your attention and patient !**