ID1217 Concurrent Programming
Lecture 7

ROYAL INSTITUTE
OF TECHNOLOGY

Tutorial: Threads, Locks and Conditions in Java SE SDK

Vladimir Vlassov

KTH/ICT/SCS

ROYAL INSTITUTE OF TECHNOLOGY

Additional readings

- The Java Tutorials. Lesson: Concurrency
  http://docs.oracle.com/javase/tutorial/essential/concurrency/
- Java Concurrency Utilities
  http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/index.html

ROYAL INSTITUTE OF TECHNOLOGY

Multithreading in Java

- A Java thread is a light-weight process represented by an object of the **`Thread`** (sub)class that includes **`start`** and **`run`** methods
  - Stack, execution context
  - Accesses all variables in its scope
- Each thread has a method **`void run()`**
  - Executes when it starts
  - Thread vanishes when it returns
  - You must implement this method
- Classes for multithreading:
  - **`public class Thread`**
  - **`public class ThreadGroup`**
  - **`public interface Runnable`**

ROYAL INSTITUTE OF TECHNOLOGY

First Way to Program and Create a Java Thread

1. Extend the **`Thread`** class
   - Override the **`run`** method and define other methods if needed;
   - Create and start a thread:
     - Instantiate the **`Thread`** subclass;
     - Call the **`start`** method on the thread object – creates a thread context and invokes **`run`** to be executed in a separate thread

Another Way to Program and Create Java Threads

2. Implement the **`Runnable`** interface in a class that represent a class of ***tasks*** to be execute in a thread
   - Implement the **`run`** method;
   - Create and start a thread with the **`Runnable`** object, i.e. the thread is given a **`Runnable`** task to execute
     - Create a **`Runnable`** object;
     - Create a thread to execute that task by passing the **`Runnable`** object to a **`Thread`** constructor
     - Call the **`start`** method on the thread object to start the thread.

ROYAL INSTITUTE OF TECHNOLOGY

`Thread` Class and `Runnable` Interface

```
public class Thread extends Object implements Runnable {
     public Thread();
     public Thread(Runnable target);
     public Thread(String name);
     public Thread(Runnable target, String name);
        ...
     public synchronized native void start();
     public void run();
        ...
   }
public interface Runnable{
   public void run();
}
```

ROYAL INSTITUTE OF TECHNOLOGY

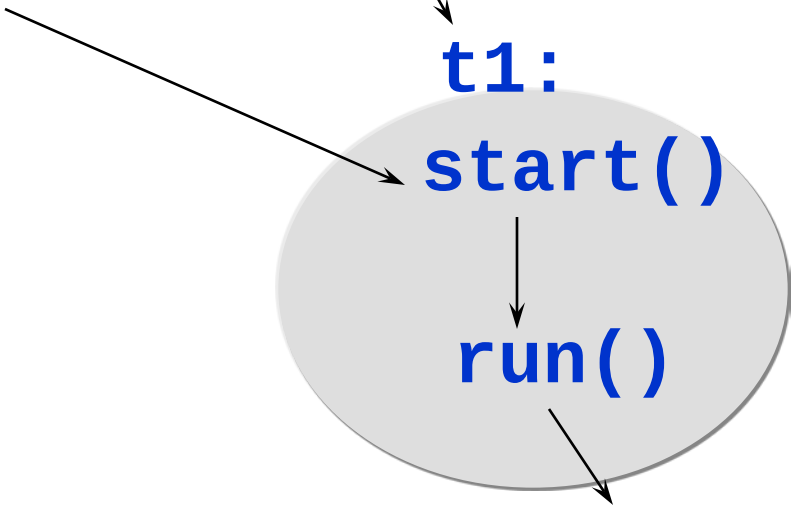
Example 1: Extending **Thread**

```
public class RunThreads {
   public static void main(String[] args) {
     OutputThread t1 = new OutputThread("One");
     OutputThread t2 = new OutputThread("Two");
     t1.start();
     t2.start();
   }
}
class OutputThread extends Thread {
   OutputThread(String name){ super(name); }
   public void run() {
        for (int i = 0; i < 3; i++) {
               System.out.println(getName());
               yield();
        }
   }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Starting a Thread

```
OutputThread t1 = new OutputThread("One");
t1.start();
```




```
for (int i = 0; i < 3; i++) {
  System.out.println(getName());
  yield();
}
```

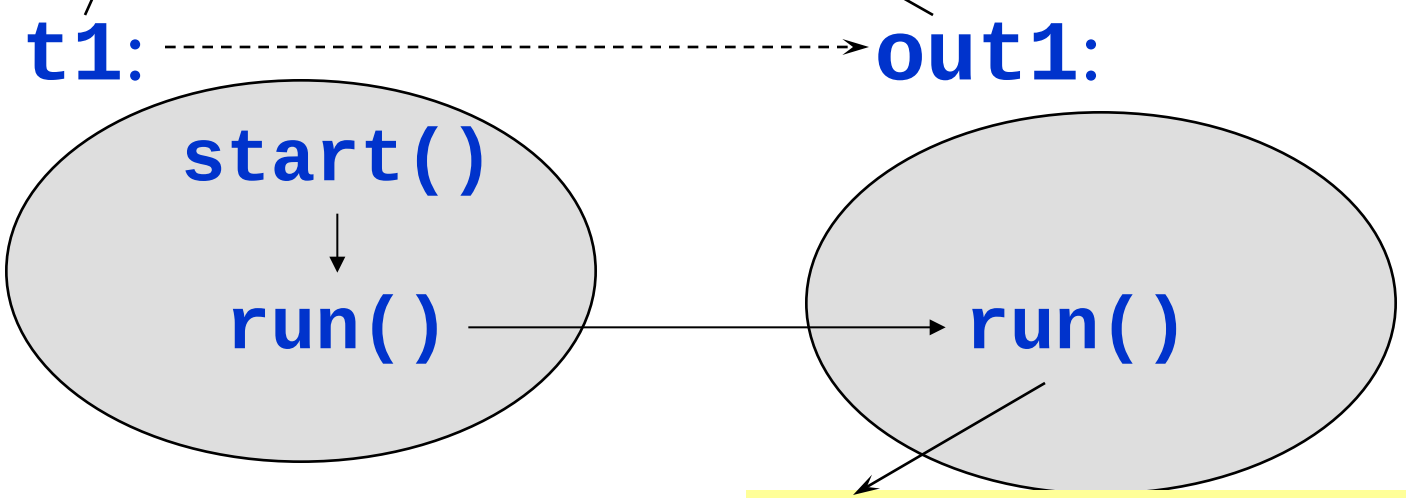
Example 2. Implementing **Runnable**

```
public class RunThreads1 {
   public static void main(String[] args) {
        OutputClass out1 = new OutputClass("One");
        OutputClass out2 = new OutputClass("Two");
        Thread t1 = new Thread(out1);
        Thread t2 = new Thread(out2);
        t1.start();
        t2.start();
   }
}
class OutputClass implements Runnable {
   String name;
   OutputClass(String s) { name = s; }
   public void run() {
        for ( int i=0; i<3; i++ ) {
                System.out.println(name);
                Thread.currentThread().yield();

        }
   }
}
```

Thread with a **Runnable** Task

```
OutputClass out1 = new OutputClass("One");
Thread t1 = new Thread(out1);
t1.start();
```




```
for ( int i=0; i<3; i++ ) {
   System.out.println(name);
   yield();
}
```

Hello Example: A **Runnable** Class

```
public class Hello implements Runnable {
  String message;
  public Hello(m) {
    message = m;
  }
  public void run() {
    System.out.println(message);
  }
}
```

"Art of Multiprocessor Programming" by Maurice Herlihy, and Nir Shavit

ROYAL INSTITUTE OF TECHNOLOGY

A **Runnable** Class

```
public class Hello implements Runnable {
  String message;
  public Hello(m) {
    message = m;
  }
  public void run() {
    System.out.println(message);
  }
}
```

Runnable interface

"Art of Multiprocessor Programming" by Maurice Herlihy, and Nir Shavit

Creating a Thread (1/3)

```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```


“Art of Multiprocessor Programming” by Maurice Herlihy, and Nir Shavit

ROYAL INSTITUTE OF TECHNOLOGY

Creating a Thread (2/3)

```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```

Create a **Runnable** object

“Art of Multiprocessor Programming” by Maurice Herlihy, and Nir Shavit

Creating a Thread (3/3)

```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```

Create the thread

“Art of Multiprocessor Programming” by Maurice Herlihy, and Nir Shavit

ROYAL INSTITUTE OF TECHNOLOGY

Starting a Thread; Joining a Thread

```
t.start();
```

- Starts the new thread
- Caller returns immediately
- Caller & thread run in parallel

```
t.join();
```

- Blocks the caller
- Waits for the thread to finish
- Returns when the thread is done

"Art of Multiprocessor Programming" by Maurice Herlihy, and Nir Shavit

ROYAL INSTITUTE OF TECHNOLOGY

Thread Constructors

- A thread is constructed with a name, belongs to a thread group and has a priority.
- The constructors include

```
Thread()
Thread(String)
Thread(ThreadGroup,String)
Thread(Runnable)
Thread(ThreadGroup,Runnable)
Thread(Runnable,String)
Thread(ThreadGroup,Runnable,String)
```

ROYAL INSTITUTE OF TECHNOLOGY

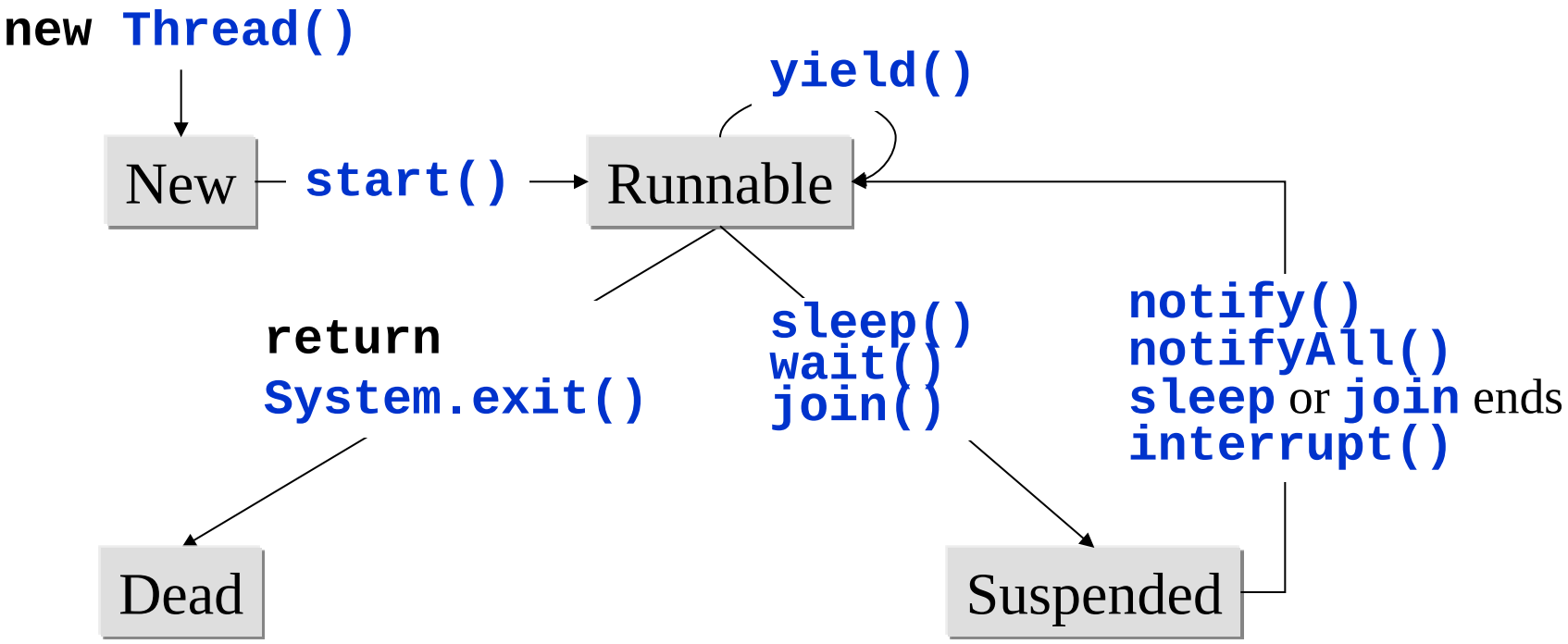
Thread Attributes

- Thread attributes can be set/get by appropriate methods:
  - Name, priority, group, type (daemon or not)
  - A group and type cannot be changed during the thread lifetime.
- Priority levels (constants defined in the Thread class):
  - **`Thread.MAX_PRIORITY`**
  - **`Thread.MIN_PRIORITY`**
  - **`Thread.NORM_PRIORITY`**

Some Methods of the **`Thread`** Class

- **`run()`**
  - Should be overridden (the code of thread is placed here), otherwise does nothing and returns;
  - Should not be invoked directly but rather calling start().
- **`start()`**
  - Start the thread; JVM invokes the run method of this thread.
- **`join()`**
  - Wait for this thread to to die.
- **`yield()`**
  - Causes a context switch.
- **`sleep(long)`**
  - The thread pauses for the specified number of milliseconds.
- **`interrupt()`**
  - Interrupt this thread.
- Get / set / check thread attributes:
  - **`setPriority(int)`**, **`getPriority()`**,
  - **`setName(String)`**, **`getName()`**,
  - **`setDaemon(boolean)`**, **`isDaemon()`**

Thread State Diagram



- IO operations affect states Runnable and Suspended in the ordinary way

ROYAL INSTITUTE OF TECHNOLOGY

Time Slicing

- There is no time slicing in the JVM run time system.
- Make it yourself if needed:

```
class TimeSlicer extends Thread {
    private int interval;
    TimeSlicer(int interval) {
         this.interval = interval;
         setPriority (Thread.MAX_PRIORITY);
         setDaemon(true);
          }
    public void run () {
         while (true) {
            try { sleep(interval); }
            catch (InterruptedException e){ }
                        }
          }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Thread Interactions

- Threads in Java execute concurrently – at least conceptually.
- Threads can be program to communicate and interact with each other
  - **Via shared objects;**
  - By calling methods and accessing variables of each other like ordinary objects;
  - Via pipes
- An object is **shared** when concurrent threads invoke its methods or access its variables.

`synchronized` Methods and Blocks

- A shared object may have **synchronized** methods or code blocks to be executed with mutual exclusion
- The `synchronized` modifier defines mutual exclusion for an entire method or a code block

- Synchronized methods and blocks, and Java concurrent utilities will be studied later in the course.
- Now we will look at *explicit* locks and conditions in Java

Locks and Conditions in Java

- **`java.util.concurrent.locks`**
  - Classes and interfaces for locking and waiting for conditions
- **`ReentrantLock`** class
  - Represents a reentrant mutual exclusion lock
  - Allows to create conditions with the `Condition` interface to wait on
  - **A**llows blocking on a condition rather than spinning
- **`Condition`** interface
  - Represents a condition variable associated with a lock
  - Allows one thread to suspend execution releasing the lock until notified by another thread
  - The suspended thread releases the lock
- Threads:
  - acquire and release locks
  - wait on conditions

The Java Lock Interface (1/5)

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Acquire lock

The Java Lock Interface (2/5)

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Release lock

The Java Lock Interface (3/5)

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Try for lock, but not too hard

The Java Lock Interface (4/5)

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Create condition to wait on

The Java Lock Interface (5/5)

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Guess what this method does?

Lock Conditions (1/4)

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Lock Conditions (2/4)

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Release lock and wait on condition

Lock Conditions (3/4)

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Wake up one waiting thread

Lock Conditions (4/4)

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
}
```

Wake up all waiting threads

Await, Signal and Signal All

```
q.await()
```

- Releases lock associated with `q`
- Sleeps (gives up processor)
- Awakens (resumes running) when signaled by `Signal` or `SignalAll`
- Reacquires lock & returns

```
q.signal();
```

- Awakens **one** waiting thread
  - Which will reacquire lock associated with `q`

```
q.signalAll();
```

- Awakens **all** waiting threads
  - Which will each reacquire lock associated with `q`

Example: Lock-Based Blocking Bounded Buffer

```
public class BoundedBuffer {

  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items;
  int rear, front, count, n;

  public BoundedBuffer(int n) {
     this.n = n;
     items = new Object[n];
  }
```

Buffer's lock and two conditions

```
  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == n) notFull.await();
      items[rear] = x; rear = (rear + 1) % n; count++;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0) notEmpty.await();
      Object x = items[front];
      front = (front + 1)% n; count--;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

The Executor Framework in `java.util.concurrent`

- For scheduling, execution, and control of asynchronous **tasks using a thread pool**
- Allows creating an executor (a pool of threads) and assigning tasks to the executor to execute
- An **`Executor`** object executes submitted tasks (**`Runnable`** objects)
- For example:

```
Executor e =
        Executors.newFixedThreadPool(numThreads);
e.execute(new RunnableTask1());
e.execute(new RunnableTask2());
```

ROYAL INSTITUTE OF TECHNOLOGY

Executor Interfaces

- An executor can have one of the following interfaces:
- **`Executor`**
  - A simple interface o launch void Runnable tasks
  - **`execute(Runnable)`**
- **`ExecutorService`**
  - Executor subinterface with additional features to manage lifecycle
  - To launch and control void Runnable tasks and Callable tasks, which return results
  - **`submit(Runnable)`** , **`submit(Callable<T>)`**, **`shutdown()`**, **`invokeAll(...)`**, **`awaitTermination(…)`**
  - **`Future<V>`** represents the result of an asynchronous computation
- **`ScheduledExecutorService`**
  - ExecutorService subinterface with support for future or periodic execution
  - For scheduling Runnable and Callable tasks

Example: Using an Executer (a Thread Pool)

```
import java.io.*;
import java.net.*;
public class Handler implements Runnable {
    private Socket socket;
    public Handler(Socket socket) {  this.socket = socket; }
    public void run() {
       try {
          BufferedReader rd = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
          PrintWriter wr = new PrintWriter(socket.getOutputStream());
          String str;
          while ((str = rd.readLine()) != null) {
             for ( int i=str.length(); i > 0; i-- ) wr.print(str.charAt(i-1));
                 wr.println();
                 wr.flush();
             }
             socket.close();
          } catch ( IOException e ) {;}
     }
}
```

```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
public class ReverseServer {
    public static void main(String[] args) throws IOException {
        int poolSize = 3, port = 4444;
        ServerSocket serverSocket = null;
        try {
           if (args.length >1) poolSize = Integer.parseInt(args[1]);
           if (args.length >0) port = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
           System.out.println("USAGE: java ReverseServer [poolSize] [port]");
           System.exit(1);
        }
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            System.out.println("Can not listen on port: " + port);
            System.exit(1);
        }
        ExecutorService executor = Executors.newFixedThreadPool(poolSize);
        while (true) {
            Socket socket = serverSocket.accept();
            executor.execute( new Handler(socket) );
        }
    }
}
```