

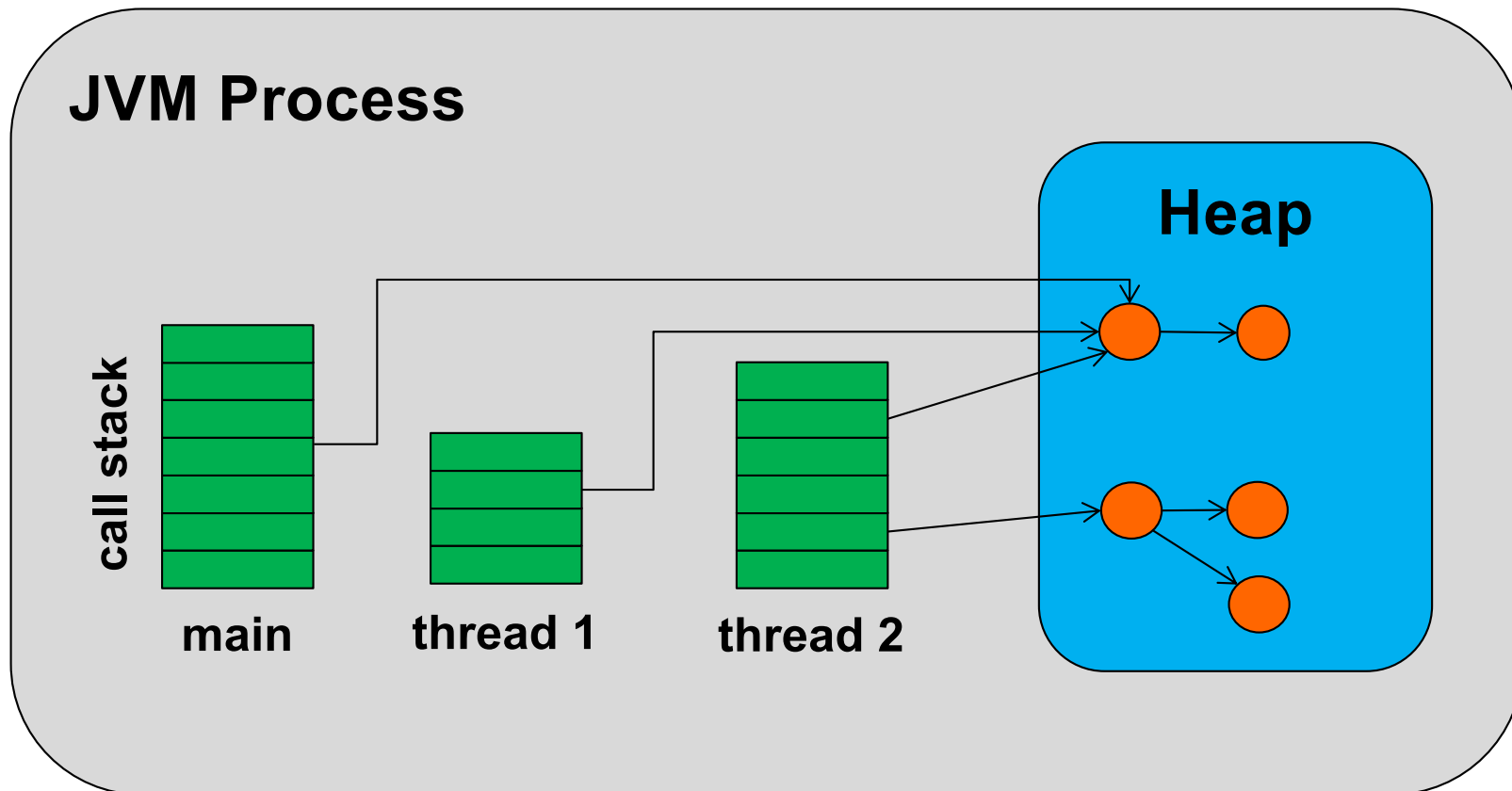
# 01\_Threads

- **Processes and Threads**
- **Java Threads**

# Processes and Threads

- **A process is an executable program loaded in memory**
  - has isolated address space allocated by OS
  - process switching is a rather expensive OS operation
  - communicates via OS (IPC), Files, Sockets
  - may contain multiple threads
- **A thread is a single sequential flow of control**
  - runs in the address space of a process
  - shares address space with other threads
  - has its own execution context
    - program counter
    - call stack
  - communicates via shared memory with other threads

# JVM Process Illustrated



# Threading Models

- **Kernel-Level-Threads** (1:1)
  - Kernel controls threads (and processes)
  - Threads are scheduled to available CPUs by kernel
  - Used by most current JVM implementations
- **User-Level-Threads** (1:n)
  - Threads implemented and managed in a library (green threads)
  - Efficient context switches (no kernel-mode privileges)
  - Application-specific scheduling may be supported
  - Different threads may not be scheduled on different processors
- **Hybrid-Threads** (m:n)
  - User-level threads are assigned to (a smaller number of) kernel threads

# Processes and Threads (Windows)

Windows Task Manager

File Options View Help

Applications Processes Services Performance Networking Users

Image Name	User Name	CPU	Memory (...)	Threads	Description
explorer.exe	dk	00	58'332 K	50	Windows ...
<b>eclipse.exe</b>	dk	00	224'608 K	43	<b>eclipse.exe</b>
chrome.exe	dk	00	36'392 K	21	Google C...
Dropbox.exe	dk	00	38'644 K	20	Dropbox
javaw.exe	dk	00	5'320 K	16	Java(TM) ...
csrss.exe		01	1'208 K	11	
taskhost.exe	dk	00	1'408 K	9	Host Proc...
POWERPNT.EXE	dk	00	31'312 K	9	Microsoft ...
EXCEL.EXE	dk	00	13'080 K	9	Microsoft ...
msseces.exe	dk	00	3'120 K	8	Microsoft ...
PrintScreenPro.exe	dk	00	3'000 K	8	Gadwin Pr...
wfcrun32.exe	dk	00	1'880 K	7	Citrix
VBoxTray.exe	dk	00	1'324 K	7	VirtualBox...
chrome.exe	dk	00	18'156 K	7	Google C...
chrome.exe	dk	00	13'464 K	7	Google C...

Show processes from all users End Process

Processes: 52 CPU Usage: 9% Physical Memory: 38%

# Processes and Threads (Linux)

1

[|||||

7.2%]

2

[|||||

3.9%]

Mem

[|||||||||||||||||||||||||||||||||||||||||5146/7935MB]

Swp

[

0/9577MB]

Tasks: 102, 252 thr, 58 kthr; 1 running

Load average: 0.13 0.19 0.12

Uptime: 04:23:34

PID	PPID	USER	S	CPU%	MEM%	TIME+	Command
2811	1	dk	S	0.0	0.2	0:00.00	`- gnome-terminal
2777	1	dk	S	0.0	0.0	0:00.00	`- /home/dk/tools/eclipse-3.7.1/eclipse
2791	2777	dk	S	0.0	6.1	0:39.59	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xms40m
2905	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2904	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2893	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2889	2777	dk	S	0.0	6.1	0:00.03	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2880	2777	dk	S	0.0	6.1	0:00.32	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2878	2777	dk	S	0.0	6.1	0:00.61	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2877	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2872	2777	dk	S	0.0	6.1	0:00.06	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2827	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2814	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2809	2777	dk	S	0.0	6.1	0:00.01	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2806	2777	dk	S	0.0	6.1	0:00.39	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2805	2777	dk	S	0.0	6.1	0:00.02	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2804	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm
2803	2777	dk	S	0.0	6.1	0:00.00	`- /home/dk/tools/jdk1.7.0_01/bin/java -Xm

# 01\_Threads

- **Processes and Threads**
- **Java Threads**

# java.lang.Thread

- **java.lang.Thread is tightly integrated into the JVM**

```
public class Thread implements Runnable {  
    /* Make sure registerNatives is the first thing <clinit> does.*/  
    private static native void registerNatives();  
    static { registerNatives(); }  
  
    public synchronized void start() {  
        ... start0(); ...  
    }  
  
    private native void start0();  
  
    public static native void sleep(long millis)  
                                   throws InterruptedException;  
  
    public static native void yield();  
  
    ...  
}
```



## java.lang.Thread (Native Part)

- **openjdk/hotspot/src/os/linux/vm/os\_linux.cpp**

```
...  
# include <pthread.h>  
...  
bool os::create_thread(Thread* thread, ThreadType thr_type,  
                        size_t stack_size) {  
    ...  
    OSThread* osthread = new OSThread(NULL, NULL);  
    if (osthread == NULL) {    return false;  }  
    ...  
    pthread_t tid;  
    int ret = pthread_create(&tid, &attr, (void* (*)(void*)) java_start,  
                             thread);  
    ...  
}
```

Source: <http://download.java.net/openjdk/jdk7/>

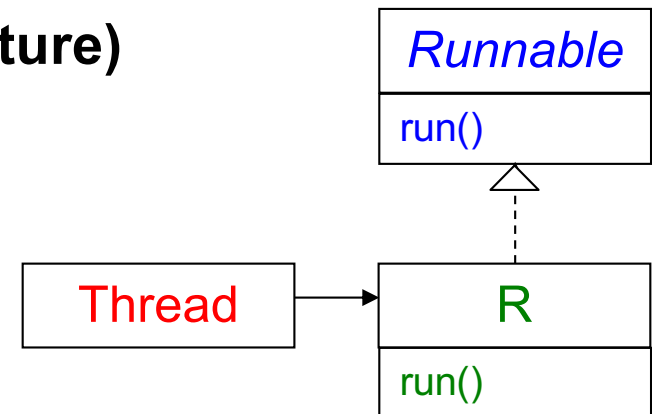
# Thread creation

- **java.lang.Thread** = Worker (Infrastructure)
- **java.lang.Runnable** = Work (Task)

```
public class R implements Runnable {  
    private int nr;  
    public R(int nr) { this.nr = nr; }  
    @Override  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println(  
                "Hello" + nr + " " + i);  
        }  
    }  
}
```

```
Runnable r = new R(1);  
Thread t = new Thread(r);
```

```
interface Runnable {  
    public void run();  
}
```



- run has no parameters
- run returns no result
- run does not declare any checked exception

# Starting the Thread

- **Thread t = new Thread(r);**
  - constructor initializes the thread object
- **t.start()**
  - activates the thread and returns immediately
  - concurrently executes the thread object's run method
  - when run() returns, the thread terminates

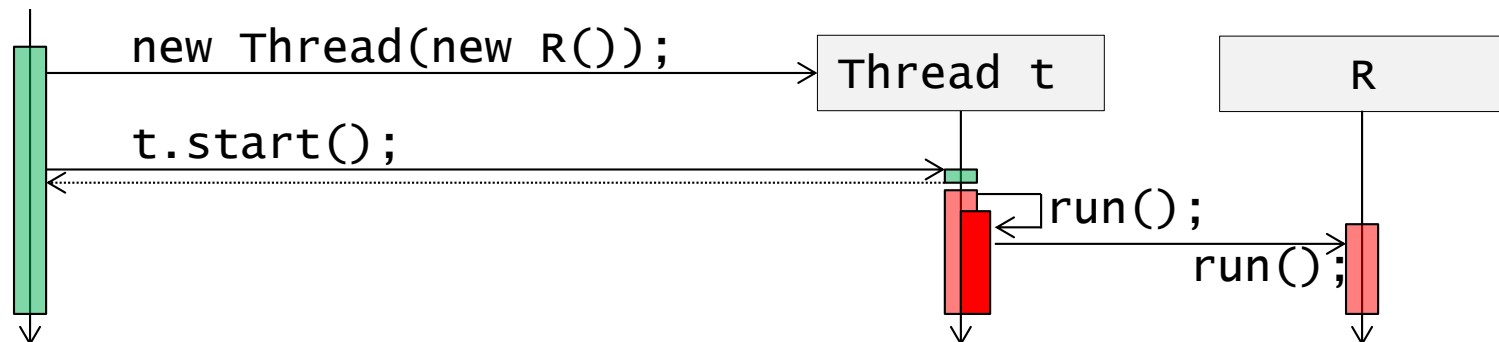
```
public class Test {  
    static void main(String[] args) {  
        Thread t1 = new Thread(new R(1));  
        Thread t2 = new Thread(new R(2));  
        t1.start();  
        t2.start();  
    }  
}
```

```
Hello1 0  
Hello2 0  
Hello1 1  
Hello2 1  
Hello2 2  
Hello2 3  
Hello2 4  
Hello2 5  
Hello2 6  
Hello2 7  
Hello2 8  
Hello2 9  
Hello1 2  
Hello1 3  
Hello1 4  
Hello1 5  
Hello1 6  
Hello1 7  
Hello1 8  
Hello1 9  
Process  
Exit...
```

# Thread.start() and Thread.run()

- **Thread.start()**

- starts a new thread of control to execute the run() method of the Thread object and returns immediately



- **Thread.run()**

- contains the code to be executed by this thread
- Default implementation in class Thread invokes run method on target runnable

```
public void run() {  
    if (target != null) {  
        target.run();  
    }  
}
```

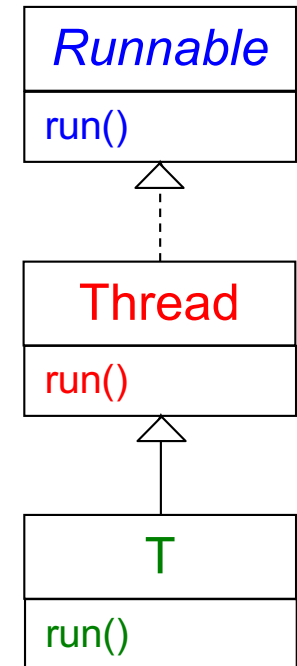
# Alternative Thread Definition

- **Class Thread implements Runnable**
  - Method **run** can be overridden in subclass of Thread

```
public class T extends Thread {  
    private int nr;  
    public T(int nr) { this.nr = nr; }  
    @Override  
    public void run() {  
        for (int i=0; i<10; i++) {  
            System.out.println("Hello" +  
                               nr + " " + i);  
        }  
    }  
}
```

- If a thread is created without an argument implementing *Runnable*, then the Thread's run method is used.

```
new T(1).start(); new T(2).start();
```



# Thread Definition using a Lambda

- **Runnable is a functional interface**

```
public class Test1 {  
  
    public static void main(String[] args) throws Exception {  
        Thread t1 = new Thread(() -> do(System.out, "1"));  
        Thread t2 = new Thread(() -> do(System.err, "2"));  
        t1.start();  
        t2.start();  
    }  
  
    private static void do(PrintStream p, String name) {  
        for (int i = 0; i < 1000; i++) {  
            p.println("Hello" + name + " " + i);  
        }  
    }  
}
```

# Extending Thread vs. Implementing Runnable

- **Extending Thread**

- Okay for simple experiments
- Easy access to Thread methods

```
getName();
```

- **Implementing Runnable separately**



- Separation of concerns
  - Task definition
  - Task execution
- Allows subclassing, i.e. no multiple subclassing problems
- You cannot miss to implement run
- Also easy access to Thread methods thanks to static imports



```
import static java.lang.Thread.*;  
currentThread().getName();
```

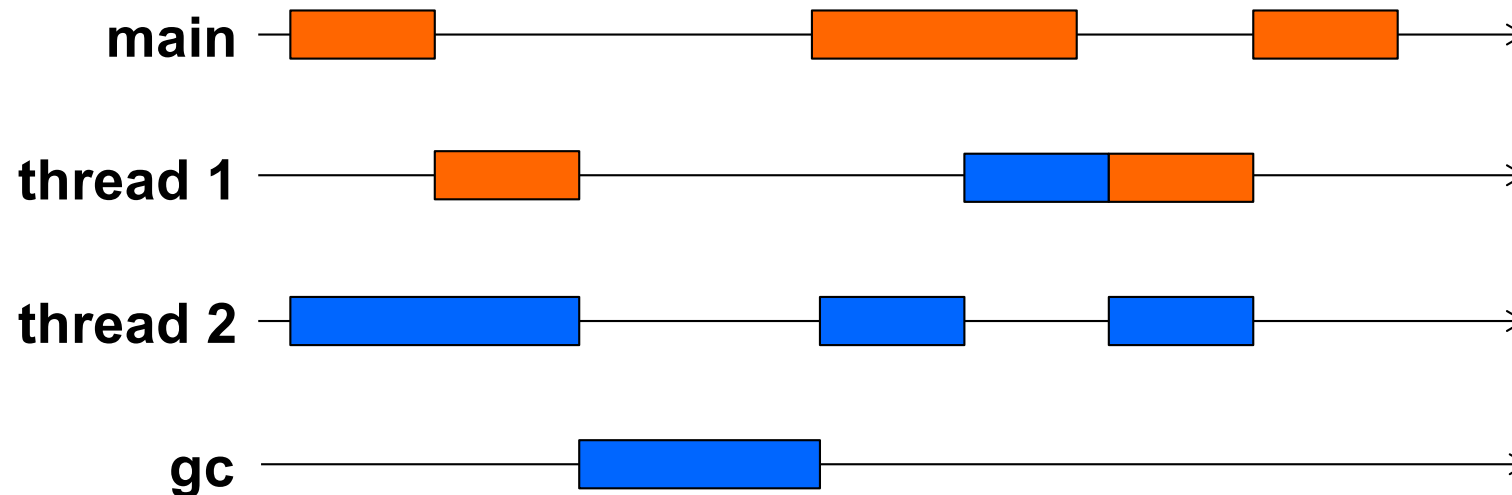
# Scheduling

- **Scheduling**
  - Per CPU core only one thread is running at any given time
  - **Scheduling = Allocation of CPU time to threads**
- **Threading Models**
  - Cooperative Threading
    - threads decide, when they should give up the processor to other threads  
*yield(); sleep(1000); blocking I/O Operations*
  - Preemptive Threading
    - JVM interrupts threads at any time (time sliced)
    - no thread can unfairly hog the processor
- **JVM specification does not mandate a threading model!**
  - Exotic Java implementations may implement cooperative scheduling
  - For the rest of the course, we assume a preemptive threading model



# Scheduling Illustrated

- **Four threads scheduled on two CPU cores**
  -  time on core 1       time on core 2



# Thread API: Controlling Threads

- **void start()**
  - Starts this thread
- **void run()**
  - The code to be executed concurrently by this thread
- **static void sleep(long millis[, int nanos])**
  - Causes the calling thread to sleep for the specified number of ms
- **static void yield( )**
  - A hint to the scheduler that the current thread is willing to yield its current use of a processor. **The scheduler is free to ignore this hint.**
- **void join([long millis])**
  - Waits (at most millis milliseconds) for this thread to terminate
- **void setDaemon(boolean)**
  - Marks this thread as either a daemon or a user thread

# Thread.join

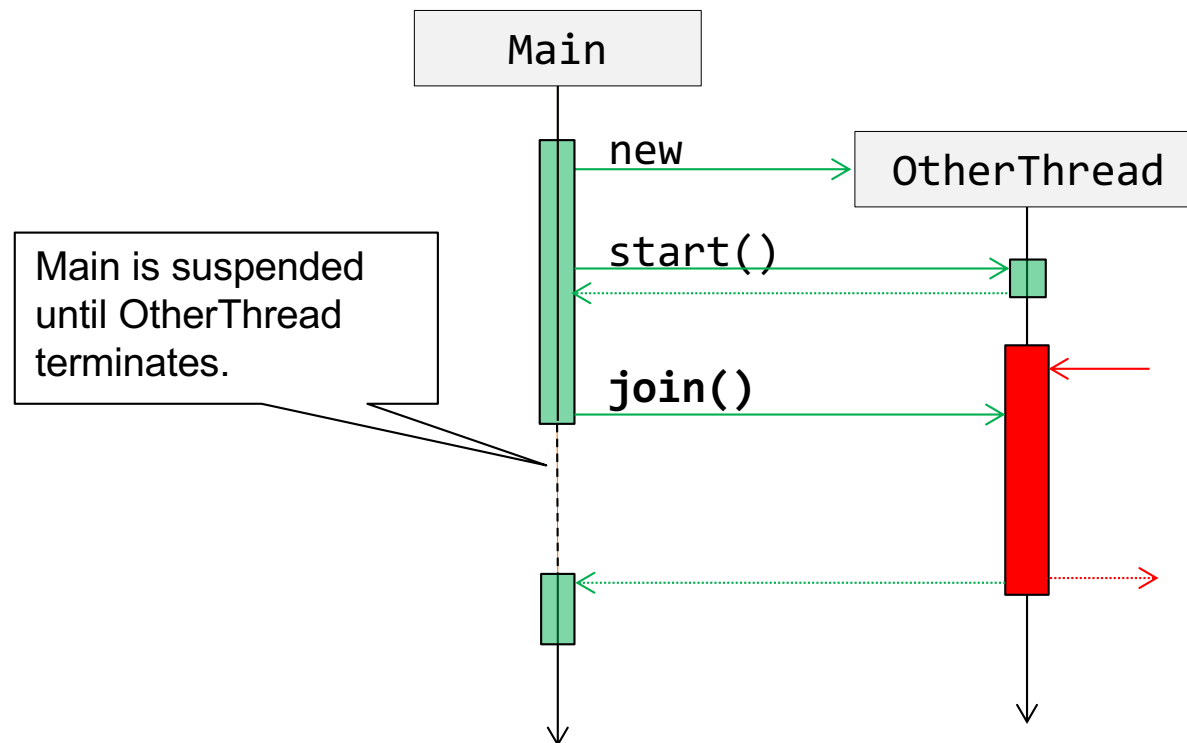
- **Example:**

```
public class Test1 {  
    static void main(String[] args){  
        Thread t1 = new Thread(new R(1));  
        Thread t2 = new Thread(new R(2));  
        t1.start();  
        t2.start();  
        System.out.println("done");  
    }  
}
```

- What is the output?

# Thread.join

- Illustrated



# Thread.join

- **Example:**

```
public class Test1 {  
    static void main(String[] args) throws Exception{  
        Thread t1 = new Thread(new R(1));  
        Thread t2 = new Thread(new R(2));  
        t1.start();  
        t2.start();  
        t1.join(); // waits until t1 has terminated  
        t2.join(); // waits until t2 has terminated  
        System.out.println("done");  
    }  
}
```

# Thread.setDaemon

- **Daemon Threads**

- Daemon threads run in the background
- If only daemon threads are active, the process stops
- setDaemon() must be called before start()

```
public class Test1 {  
    static void main(String[] args){  
        Thread t1 = new Thread(new R(1));  
        Thread t2 = new Thread(new R(2));  
        t2.setDaemon(true);  
        t1.start();  
        t2.start();  
    }  
}
```

- What is the output?

# JVM: Start and Termination

- **On JVM start up**
  - An initial non-daemon thread which calls the *main* method is started
- **The JVM continues to execute threads until either**
  - *Runtime.getRuntime().exit(n);* has been called and the security manager has permitted the exit operation to take place
  - All threads that are not daemon threads have died, either by
    - returning from the call to the *run* method
    - throwing an exception that propagates beyond the *run* method

# Uncaught Exception Handlers

- **Uncaught exception handlers**

- Allows to detect whether threads die due to an uncaught exception

```
interface UncaughtExceptionHandler {  
    void uncaughtException(Thread t, Throwable e);  
}
```

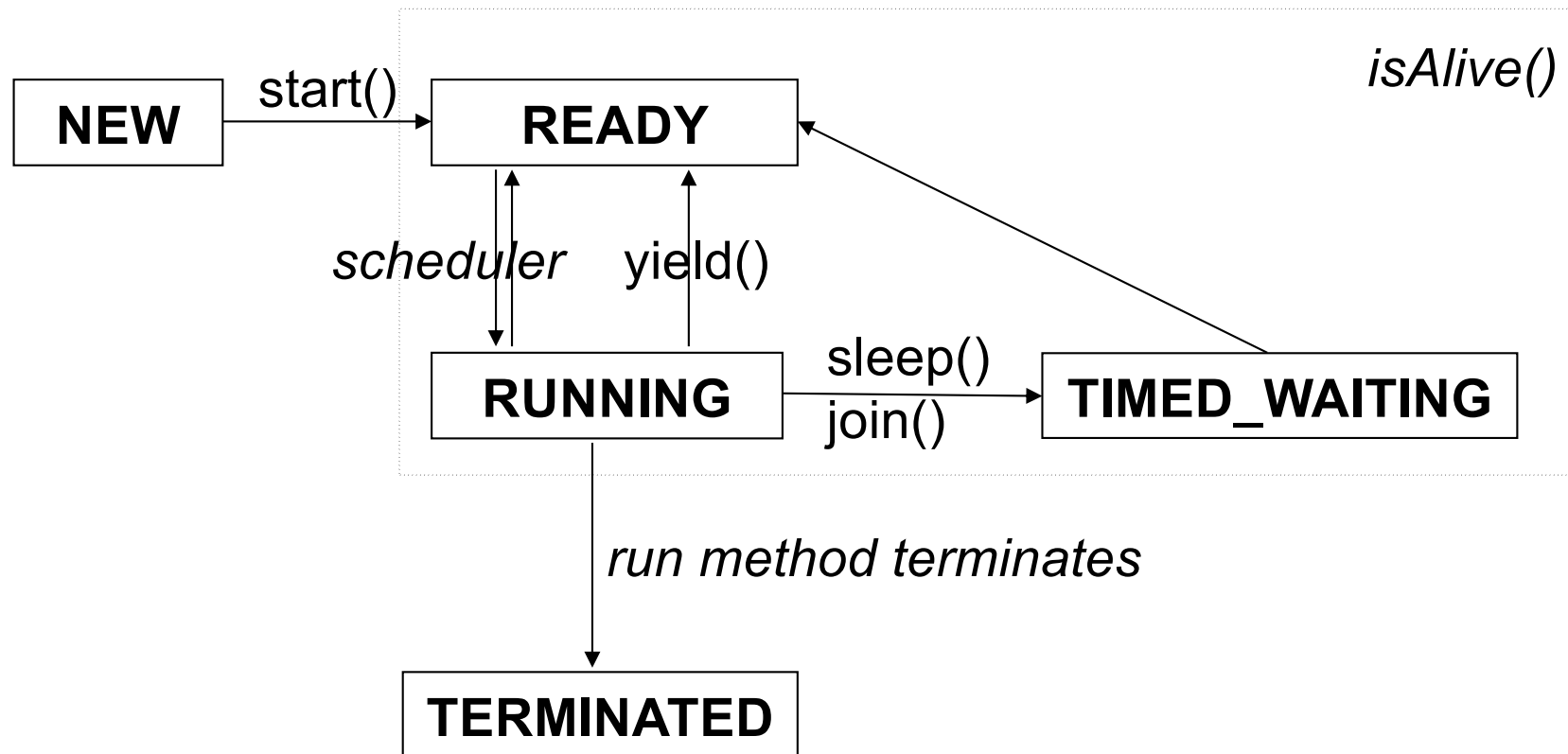
- Handling uncaught exceptions
  - Default behavior: print stack trace to System.err
  - Report message in application log
  - Try to restart the thread
  - Shut down the application
- Install Custom Exception Handlers (called before default behavior)
  - Thread.setUncaughtExceptionHandler [instance method]
  - Thread.setDefaultUncaughtExceptionHandler [static method]



# Thread API: Gathering Information

- **static Thread currentThread()**
  - Returns a reference to the currently executing thread object
- **long getId()**
  - Returns the unique identifier of this thread
- **String getName()**
  - Returns this thread's name
- **boolean isDaemon()**
  - Tests if this thread is a daemon thread
- **State getState()**
  - Returns the state of this thread
- **boolean isAlive()**
  - Tests if this thread is alive (i.e. started and not yet terminated)

# Thread State



# Thread Priority

- **Thread Priorities**

```
public void setPriority(int priority);  
public int  getPriority();
```

- A new thread inherits its priority from the thread that *created* it
- Must be in the range 1..10
  - Thread.MIN\_PRIORITY = 1; // minimum priority that a thread can have
  - Thread.NORM\_PRIORITY = 5; // default priority
  - Thread.MAX\_PRIORITY = 10; // maximum priority that a thread can have
- Priorities map on to some machine-specific values
  - Linux: "nice" values | Windows: local thread priorities
- Main issue:

**A JVM is free to implement priorities in any way it chooses, including ignoring the value!**

## Summary

- **Thread**
  - Mean to execute multiple control flows concurrently within a process
  - Communicates via shared memory with other threads
- **Thread scheduling is nondeterministic in common JVMs**
- **Thread construction is preferably be done by implementing `java.lang.Runnable` separately (not by subclassing `Thread`)**
- **Use `Thread.start()` to start a newly created thread**
- **Use `Thread.join()` to wait for a thread to terminate**