

## Threads

- ♦ A *thread* is a single sequential flow of control within a program.
- ♦ A *multi-threaded* program is one has multiple threads running simultaneously.
- ♦ Multi-threaded programs are also known as concurrent programs
- ♦ A multi-threaded program can run on a single-processor or multi-processor computer.
- ♦ Java provides language support for multi-threading.
- ♦ The Java runtime is multi-threaded.

## Multi-Threaded Programming

- ♦ Most conventional programming languages are single-threaded.
- ♦ Advantages:
  - ♦ reactive systems: control systems
  - ♦ responsiveness: GUI
  - ♦ availability: server
- ♦ Multi-threaded programming are more difficult:
  - ♦ shared access to objects
  - ♦ non-determinism
  - ♦ overhead: thread creation, context switching, and synchronization
  - ♦ race hazard

## A Bank Account

```
public class Account {
    // ...
    public boolean withdraw(long amount) {
        if (amount <= balance) {
            long newbalance = balance - amount;
            balance = newbalance;
            return true;
        } else
            return false;
    }
    private long balance;
}
```

## A "Perfect" Crime

Assume the initial balance is \$1,000,000. Two withdraw requests are made almost simultaneously.

balance	withdraw 1	withdraw 2
1,000,000	amount<=balance	
1,000,000		amount<=balance
1,000,000	newbalance=...	
1,000,000		newbalance=...
0	balance=...	
0		balance=...
0	return true;	
0		return true;

## Creating Threads

### Method A:

- ◆ Subclass the Thread class.
- ◆ Override the run( ) method.
- ◆ Create a thread with new MyThread( ... ).
- ◆ Start the thread by calling the start( ) method.

### Method B:

- ◆ Implement the Runnable interface.
- ◆ Override the run( ) method.
- ◆ Create a thread with new Thread(runnable).
- ◆ Start the thread by calling the start( ) method.

## A Simple Counter

```
public class Counter1 extends Thread {  
  
    protected int count, inc, delay;  
  
    public Counter1(int init, int inc, int delay) {  
        this.count = init;  
        this.inc = inc;  
        this.delay = delay;  
    }  
    public void run() {  
        try {  
            for (;;) {  
                System.out.print(count + " ");  
                count += inc;  
                sleep(delay);  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

## A Simple Counter (cont'd)

(class Counter1 continued.)

```
public static void main(String[] args) {  
    new Counter1(0, 1, 33).start();  
    new Counter1(0, -1, 100).start();  
}  
}
```

### Output:

```
0 0 1 2 -1 3 4 5 -2 6 7 8 -3 9 10 -4 11 12 13  
-5 14 15 16 -6 17 18 -7 19 20 21 -8 22 23 24 -9  
25 26 -10 27 28 -11 29 30 31 -12 32 33 34 -13  
35 36 37 -14 38 39 -15 40 41 42 -16 43 44 45
```

## A Simple Counter II

```
public class Counter2 implements Runnable {  
  
    protected int count, inc, delay;  
  
    public Counter2(int init, int inc, int delay) {  
        this.count = init;  
        this.inc = inc;  
        this.delay = delay;  
    }  
    public void run() {  
        try {  
            for (;;) {  
                System.out.print(count + " ");  
                count += inc;  
                Thread.sleep(delay);  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

## A Simple Counter II (cont'd)

(class Counter2 continued.)

```
public static void main(String[] args) {
    new Thread(new Counter2(0, 1, 33)).start();
    new Thread(new Counter2(0, -1, 100)).start();
}
}
```

Output:

```
0 0 1 2 -1 3 4 5 -2 6 7 8 -3 9 10 -4 11 12 13 -5
14 15 16 -6 17 18 -7 19 20 21 -8 22 23 24 -9 25
26 -10 27 28 -11 29 30 31 -12 32 33 34 -13 35 36 -
14 37 38 39 -15 40 41 42 -16 43 44 45 -17 46 47
```

## Controlling Threads

- ♦ `start()`: start running the thread
- ♦ `isAlive()`: return `true` if the thread has been started and not terminated.
- ♦ `stop()`: abruptly and irrevocably stop a thread.  
**(Deprecated in 1.2)**
- ♦ `sleep()`: sleep a given amount of time and wake up automatically.
- ♦ `suspend()` and `resume()`: suspend and resume a thread.  
**(Deprecated in 1.2)**
- ♦ `join()`: wait until the target thread is finished.
- ♦ `interrupt()`: if the thread is blocked wake up the thread with an `InterruptedException`, otherwise, set the interrupted flag.
- ♦ `yield()`: give other threads of the same priority a chance to run.

## Thread Priority and Scheduling

- ♦ Java virtual machine implements a very simple scheduling strategy:
- ♦ Each thread has a priority between `MIN_PRIORITY` and `MAX_PRIORITY`.
- ♦ A live thread may be *runnable* or *blocked*.
- ♦ The runnable thread of the highest *priority* will be selected to run.
- ♦ If there are more than one runnable threads of the same highest priority, one will be selected *arbitrarily*. There is no requirement of *fairness*.
- ♦ A thread of a higher priority will preempt a thread of a lower priority.

## Thread Priority and Scheduling (cont'd)

A thread that is currently running will relinquish the control when one of the following happens:

- ♦ It yields, i.e., `yield()` is invoked.
- ♦ It becomes blocked.
- ♦ A thread with a higher priority becomes runnable.
- ♦ Its time-slice has expired.

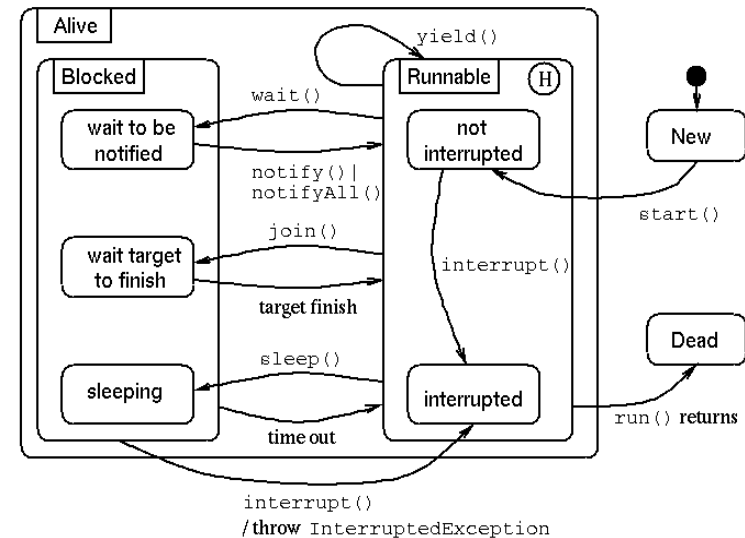
## Controlling Priorities

```
public class Thread {
    public static final int MAX_PRIORITY = ...;
    public static final int MIN_PRIORITY = ...;
    public static final int NORM_PRIORITY = ...;

    public final void setPriority(int newPriority)
        {...}
    public final int getPriority() {...}
    // ...
}
```

- By default, a new thread has the same priority as the one that creates it.
- The priority of a thread can be changed at any time.
- Only use priorities to tune the performance of programs. The correctness of programs should not depend on the priorities and the scheduling policies.

## Thread States



## Consistency of Object States

- While an object is being modified, it may be in many intermediate states that are *inconsistent* or *invalid*.
- If a thread that is modifying an object is interrupted, it may leave the object in an inconsistent state.
- A class is said to be *thread safe*, if it ensures the consistency of its objects in the presence of multiple threads.
- To maintain the consistency of object states, a thread should not be interrupted while it is in certain regions, called *critical regions*.
- An operation that can not be interrupted is called an *atomic operation*.

## Synchronization

### Mutual exclusion of threads.

- Each synchronized method or statement is guarded by an object.
- When entering a synchronized method or statement, the object will be locked until the method is finished.
- When the object is locked by another thread, the current thread must wait.

## Synchronized Method

```
public class Account {
    // ...
    public synchronized boolean
        withdraw(long amount) { ... }
}
```

## Synchronized Statement

```
public class Account {
    // ...
    public boolean withdraw(long amount) {
        synchronized (this) {
            if (amount <= balance) {
                long newbalance = balance - amount;
                balance = newbalance;
                return true;
            } else
                return false;
        }
    }
}
```

## When The Queue Is Empty

Consider the following queue class.

```
public class Queue {
    public synchronized Element dequeue() {
        if (!isEmpty())
            return ...;
        else
            // what to do?
    }
    public synchronized void enqueue(Element e) {
        if (!isFull())
            // ;
        else
            // what to do?
    }
    // ...
}
```

## Cooperation Among Threads

- ◆ Synchronization only address the issue of exclusion not cooperation.
- ◆ Cooperation involves
- ◆ `wait()`: when a thread is unable to continue, let other threads to proceed.
- ◆ `notify()`: notify other threads that they may be able to proceed.
- ◆ The `wait()`, `notify()`, and `notifyAll()` methods are defined in the `Object` class.

## **wait() and notify()**

```
public class Queue {
    public synchronized Element dequeue() {
        while (isEmpty())
            wait();
        notify();
        return ...;
    }
    public synchronized void enqueue() {
        while (Full())
            wait();
        notify();
        // ...
    }
    // ...
}
```