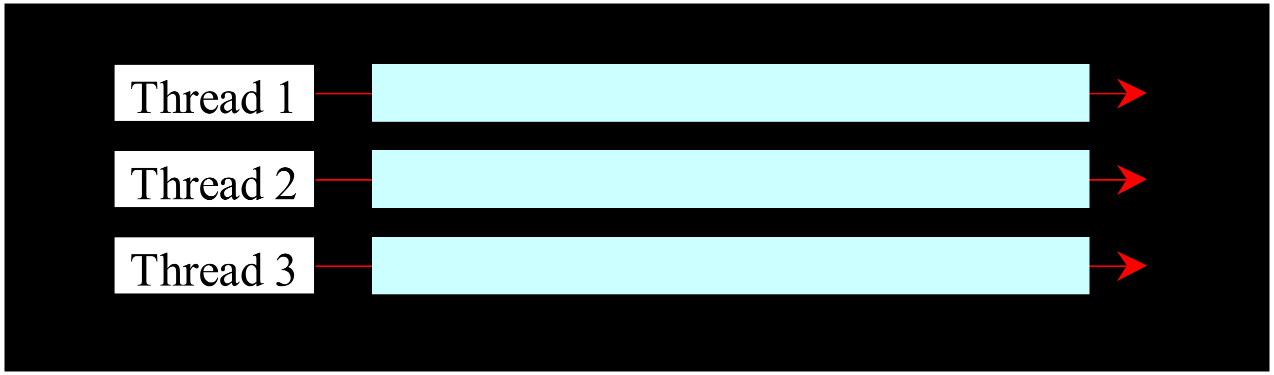
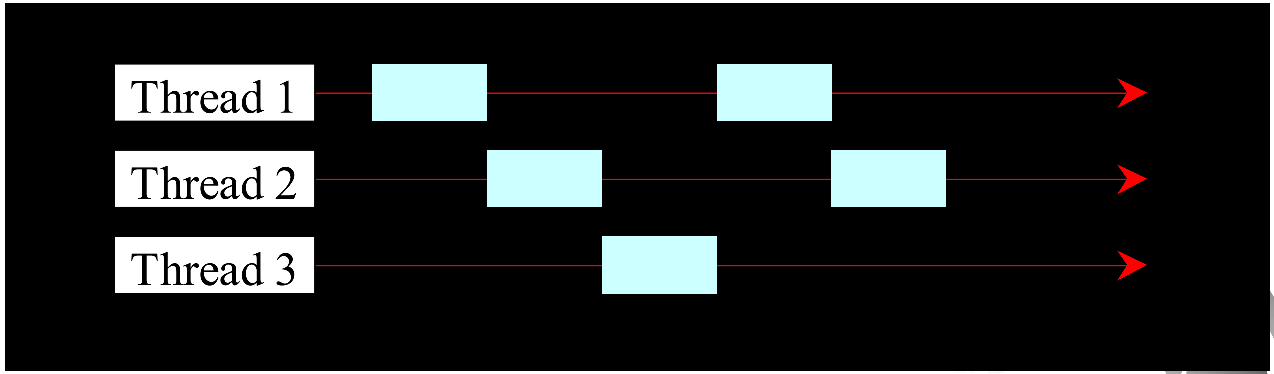
Multithreading in Java

Threads Concept

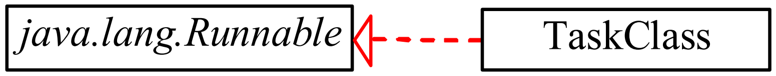
Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



Creating Tasks and Threads



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

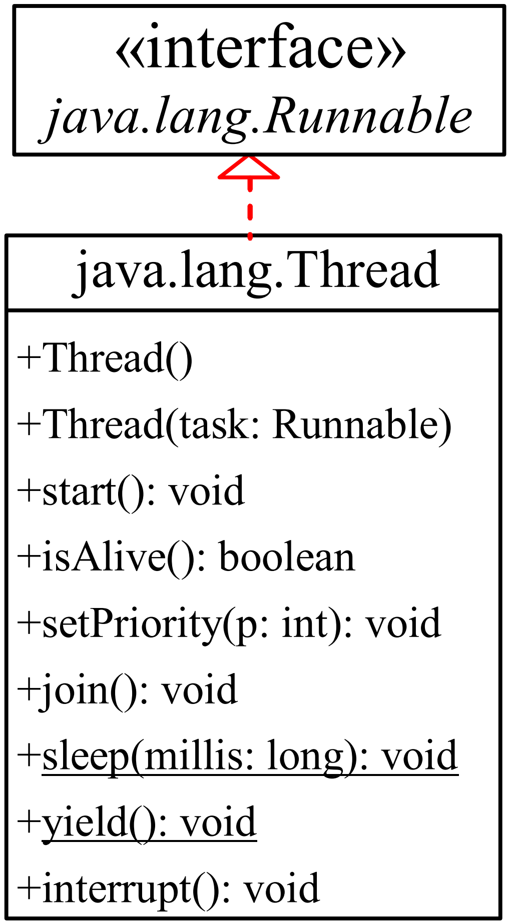
```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Using the `Runnable` Interface to Create and Launch Threads
Example - *TaskThreadDemo*

Objective: Create and run three threads:

- The first thread prints the letter *a* 100 times.
- The second thread prints the letter *b* 100 times.
- The third thread prints the integers 1 through 100.

The Thread Class



+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

The Static yield() Method

You can use the yield() method to temporarily release time for other threads.

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the *print100* thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds.

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

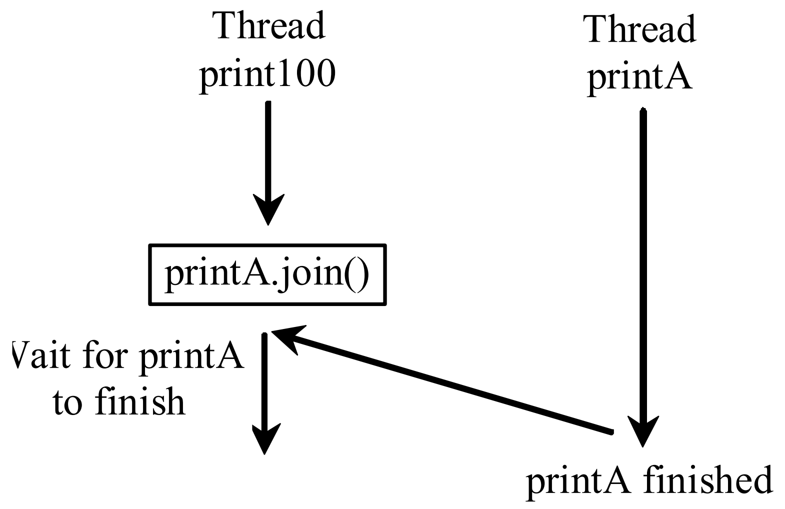
Every time a number (>= 50) is printed, the *print100* thread is put to sleep for 1 millisecond.

The join() Method

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

For example, suppose you modify the code in *TaskThreadDemo.java* as follows:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



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

isAlive(), interrupt(), and isInterrupted()

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

- The interrupt() method interrupts a thread in the following way:
  - if 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.

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

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`. You can reset the priority using `setPriority(int priority)`.

- Some constants for priorities include

```
Thread.MIN_PRIORITY
Thread.MAX_PRIORITY
Thread.NORM_PRIORITY
```

Thread Synchronization

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

For 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);