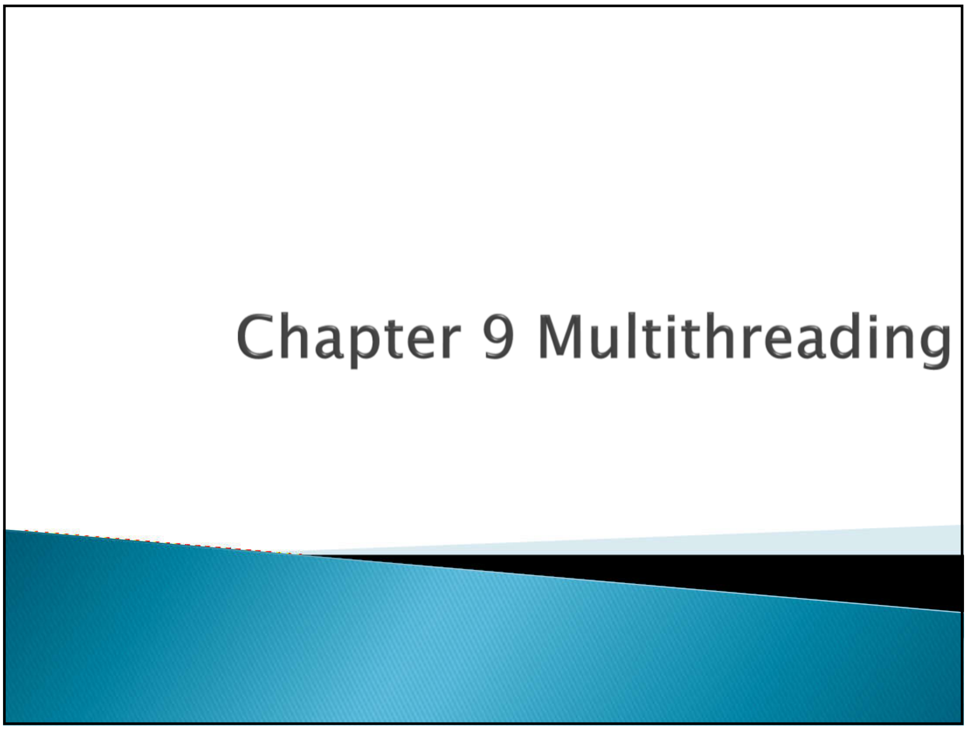
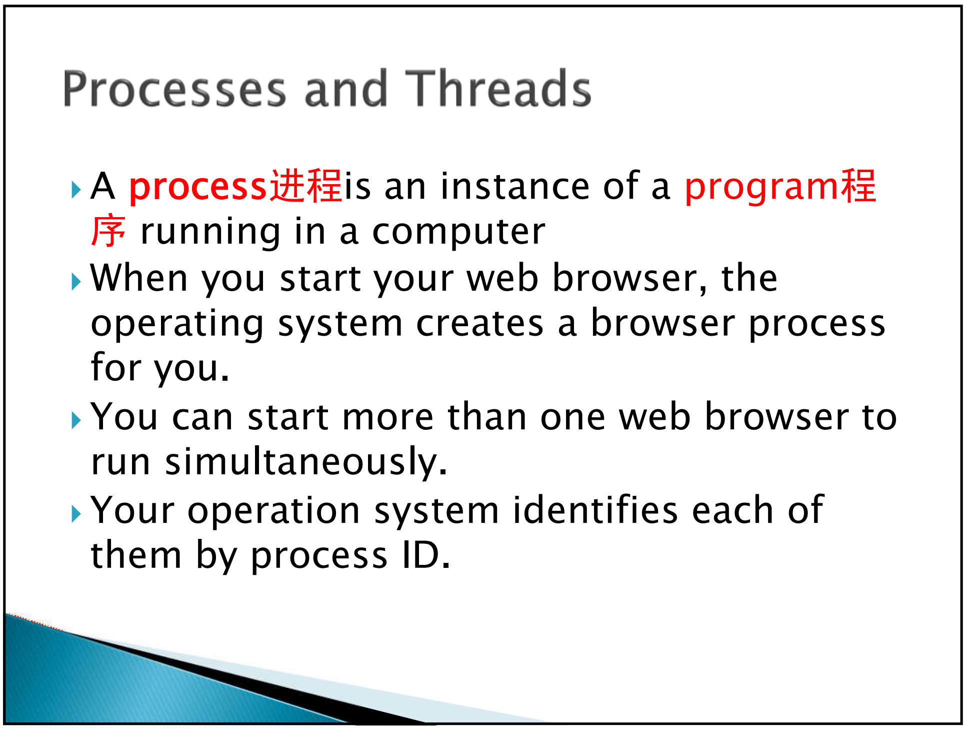
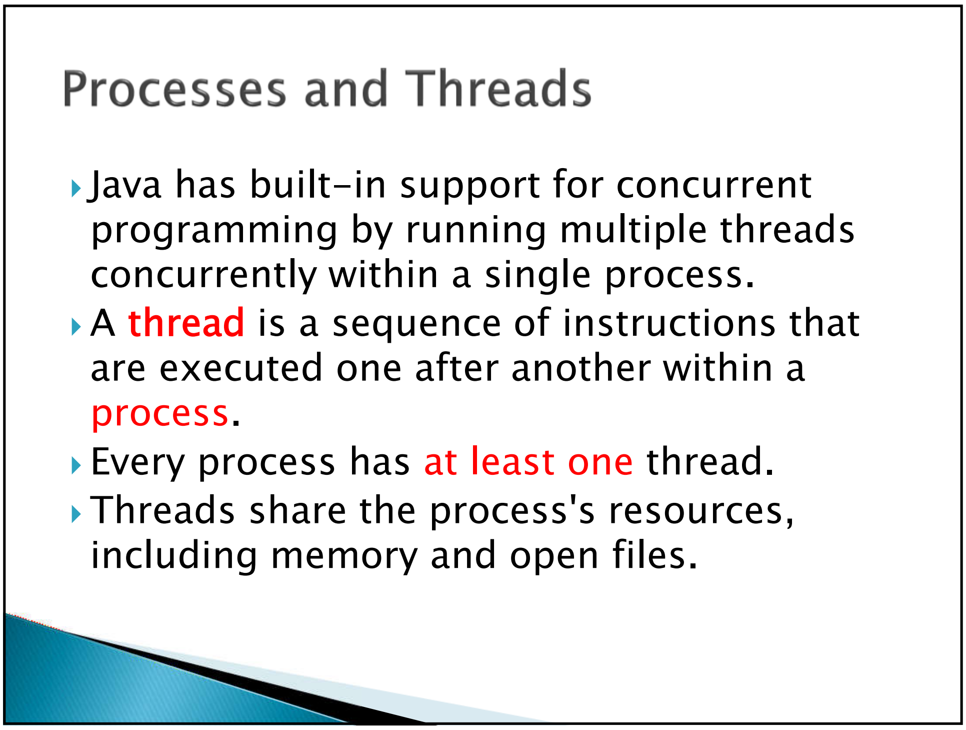


Chapter 9 Multithreading



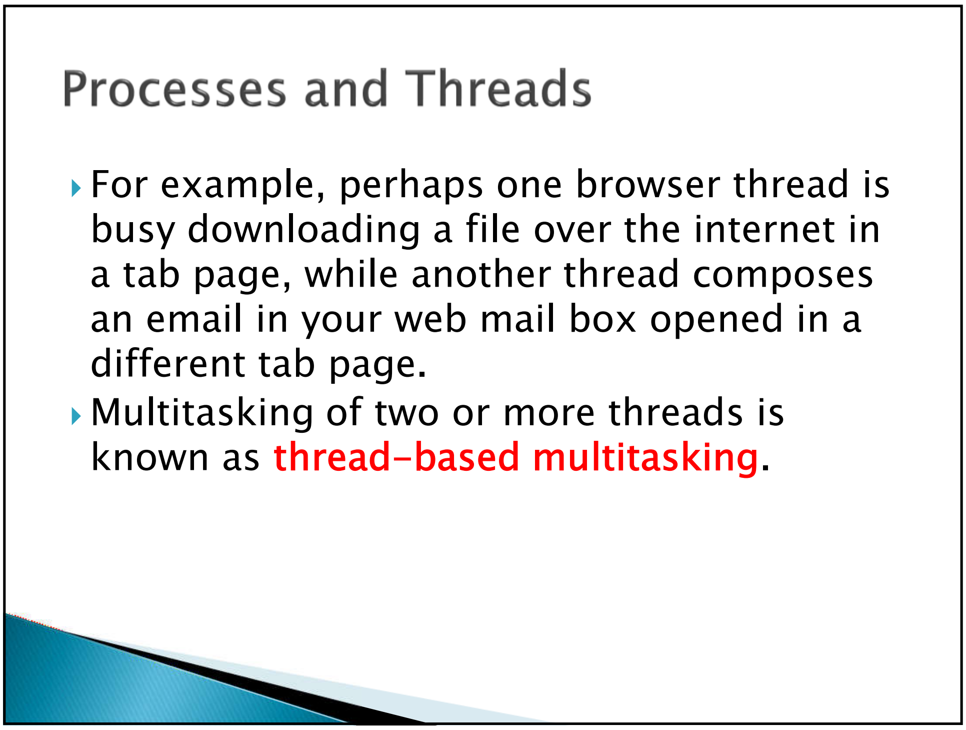
Processes and Threads

- A **process进程**is an instance of a program程序 running in a computer
- When you start your web browser, the operating system creates a browser process for you.
- You can start more than one web browser to run simultaneously.
- Your operation system identifies each of them by process ID.



Processes and Threads

- Java has built-in support for concurrent programming by running multiple threads concurrently within a single process.
- A **thread** is a sequence of instructions that are executed one after another within a process.
- Every process has at least one thread.
- Threads share the process's resources, including memory and open files.



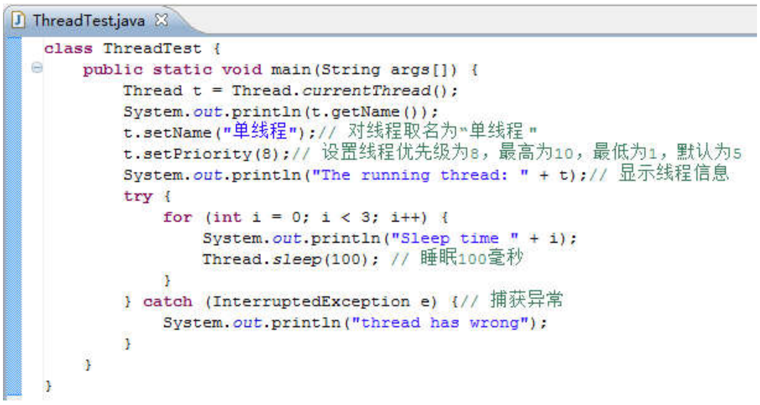
Processes and Threads

- For example, perhaps one browser thread is busy downloading a file over the internet in a tab page, while another thread composes an email in your web mail box opened in a different tab page.
- Multitasking of two or more threads is known as **thread-based multitasking**.

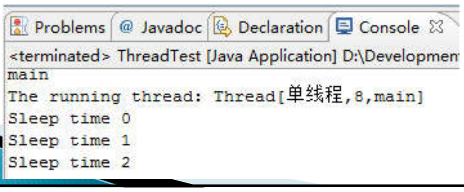
Processes and Threads

- Every application has at least one thread called the main thread. 主线程
- A thread has the ability to create new threads.
- The creating thread is the parent thread, and the created thread is called a child thread.
- Threads can help to take advantage of multi-core processor programming.

Main Thread in Java



```
class ThreadTest {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());
        t.setName("单线程");// 对线程取名为"单线程"
        t.setPriority(8);// 设置线程优先级为8，最高为10，最低为1，默认为5
        System.out.println("The running thread: " + t);// 显示线程信息
        try {
            for (int i = 0; i < 3; i++) {
                System.out.println("Sleep time " + i);
                Thread.sleep(100); // 睡眠100毫秒
            }
        } catch (InterruptedException e) {// 捕获异常
            System.out.println("thread has wrong");
        }
    }
}
```

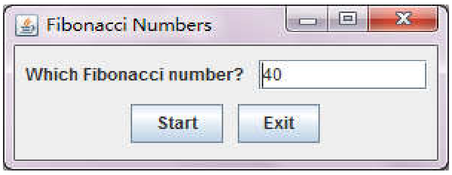


```
<terminated> ThreadTest [Java Application] D:\Developmen
main
The running thread: Thread[单线程,8,main]
Sleep time 0
Sleep time 1
Sleep time 2
```

Processes and Threads

- The compute-intensive Fibonacci problem in a Swing program shows the motivation of thread-based multitasking.

```
private long fib(long n) {
    if (n == 0) {
        return 0L;
    }
    if (n == 1) {
        return 1L;
    }
    return fib(n - 1) + fib(n - 2);
}
```



Which Fibonacci number? 40

```
btnStart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        int n = Integer.parseInt(textFieldNumber.getText());
        long result = fib(n);
        JOptionPane.showMessageDialog(null, result + "");
    }
});
```

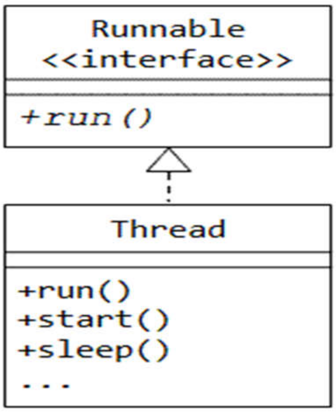
Processes and Threads

- If you type in a number larger than 40 and click the "Start" button, the "Start" button will remain pressed until the calculation result is shown in a message dialog.
- At the same time, the window is frozen, meaning that there is no response when you resize or move the window.
- If you click the "Exit" button before the result dialog pops up, there is also no response.

Threads in Java

- In Java, a thread is represented by an object belonging to the subclass of java.lang.Thread and overrides its run() method,
- or a class that implements the Runnable interface, which has one method, run().
- This run() method defines the operations that will be performed by the thread.

Threads in Java

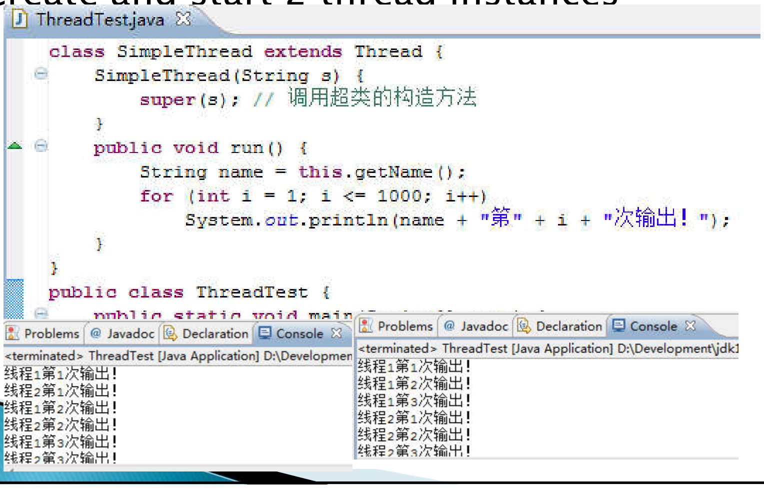


Threads in Java

- run()		Specifies what the new thread does. When run() completes, the thread terminates.
- start()		Makes the new thread runnable. The current thread continues as parent.
- sleep()		Suspends the thread and yield control to other threads for the specified milliseconds.
- interrupt()	Interrupts the sleep() method.
- isAlive()		Returns true if the thread is not terminated.
- setPriority()		Sets the priority of the thread, which is dependent on implementation.
- yield()		Suspends the current thread to allow other threads to run.
- The stop(), suspend(), and resume() methods have been deprecated since JDK 1.4, because they are not thread-safe, due to the release of monitors.

Example Thread Class

- Define a thread class, output for 1000 times
- Create and start 2 thread instances



ThreadTest.java

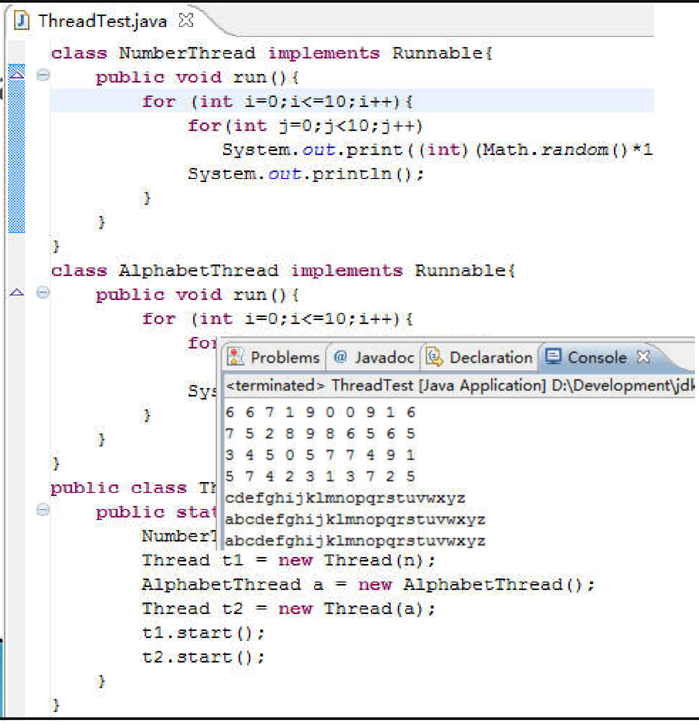
```java
class SimpleThread extends Thread {
    SimpleThread(String s) {
        super(s); // 调用超类的构造方法
    }
    public void run() {
        String name = this.getName();
        for (int i = 1; i <= 1000; i++)
            System.out.println(name + "第" + i + "次输出！ ");
    }
}
public class ThreadTest {
    public static void main
```

```
<terminated> ThreadTest [Java Application] D:\Developmen
线程1第1次输出!
线程2第1次输出!
线程1第2次输出!
线程2第2次输出!
线程1第3次输出!
线程2第3次输出!
```

```
<terminated> ThreadTest [Java Application] D:\Development\jdk1
线程1第1次输出!
线程1第2次输出!
线程1第3次输出!
线程2第1次输出!
线程2第2次输出!
线程2第3次输出!
```

12

Ex



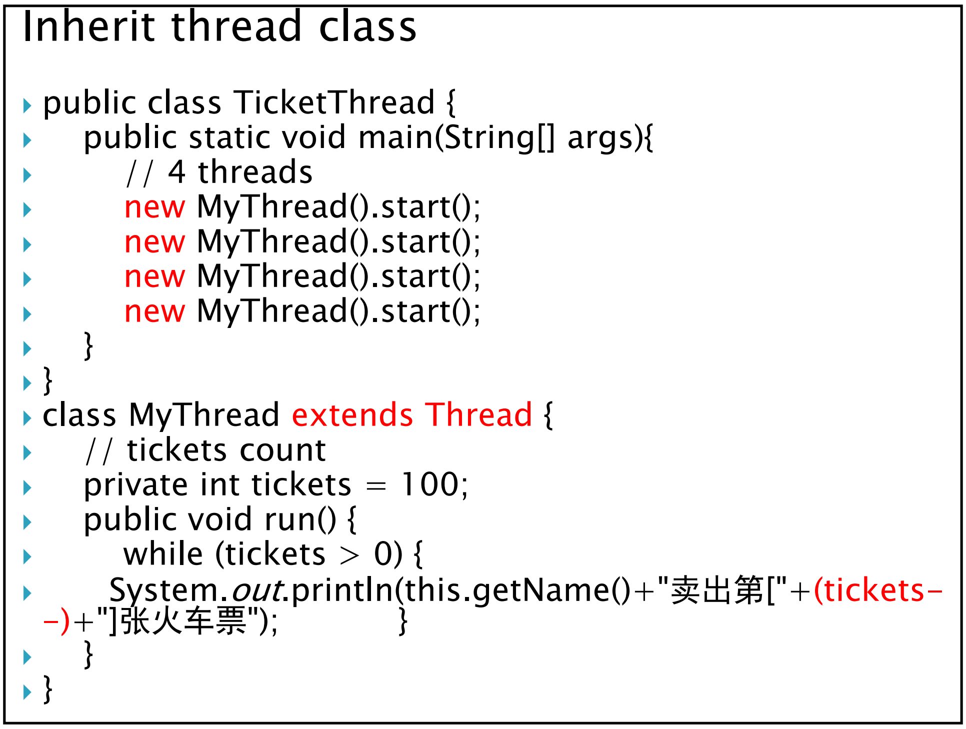
```
class NumberThread implements Runnable{
    public void run(){
        for (int i=0;i<=10;i++){
            for(int j=0;j<10;j++)
               System.out.print((int)(Math.random()*1
            System.out.println();
        }
    }
}
class AlphabetThread implements Runnable{
    public void run(){
        for (int i=0;i<=10;i++){
            fo

            Sys
        }
    }
}
public class Th
    public stat
        NumberT
        Thread t1 = new Thread(n);
        AlphabetThread a = new AlphabetThread();
        Thread t2 = new Thread(a);
        t1.start();
        t2.start();
    }
}
```

```
<terminated> ThreadTest [Java Application] D:\Development\jdk
6 6 7 1 9 0 0 9 1 6
7 5 2 8 9 8 6 5 6 5
3 4 5 0 5 7 7 4 9 1
5 7 4 2 3 1 3 7 2 5
cdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
```

13

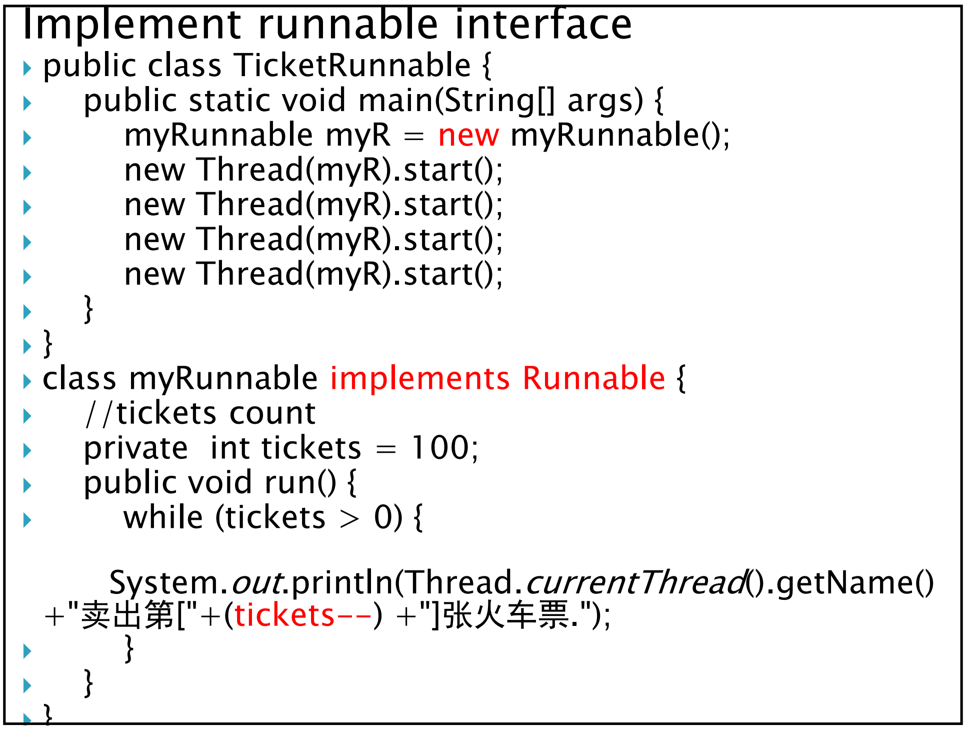
Thread Class and Runnable Interface

- Simulating 4 ticket windows to sell 100 train tickets at the same time
  - Inherit thread class
  - Implement runnable interface



Inherit thread class

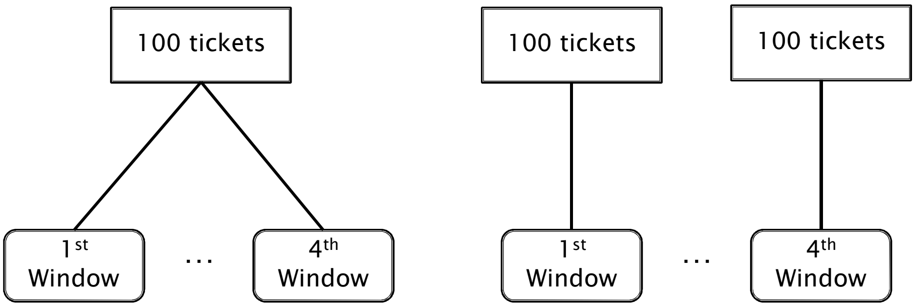
```
public class TicketThread {
    public static void main(String[] args){
        // 4 threads
        new MyThread().start();
        new MyThread().start();
        new MyThread().start();
        new MyThread().start();
    }
}
class MyThread extends Thread {
    // tickets count
    private int tickets = 100;
    public void run() {
        while (tickets > 0) {
      System.out.println(this.getName()+"卖出第["+(tickets--)+"]张火车票");          }
    }
}
```



Implement runnable interface

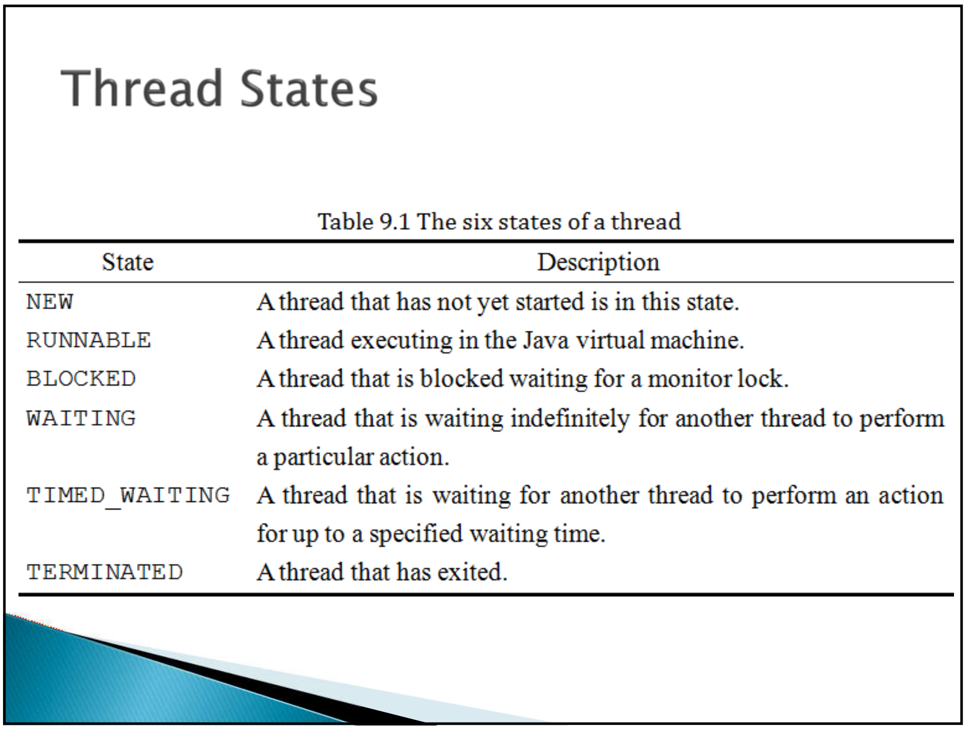
```
public class TicketRunnable {
    public static void main(String[] args) {
        myRunnable myR = new myRunnable();
        new Thread(myR).start();
        new Thread(myR).start();
        new Thread(myR).start();
        new Thread(myR).start();
    }
}
class myRunnable implements Runnable {
    //tickets count
    private  int tickets = 100;
    public void run() {
        while (tickets > 0) {

      System.out.println(Thread.currentThread().getName()
+"卖出第["+(tickets--) +"]张火车票.");
        }
    }
}
```

Runnable vs Thread



Runnable vs Thread

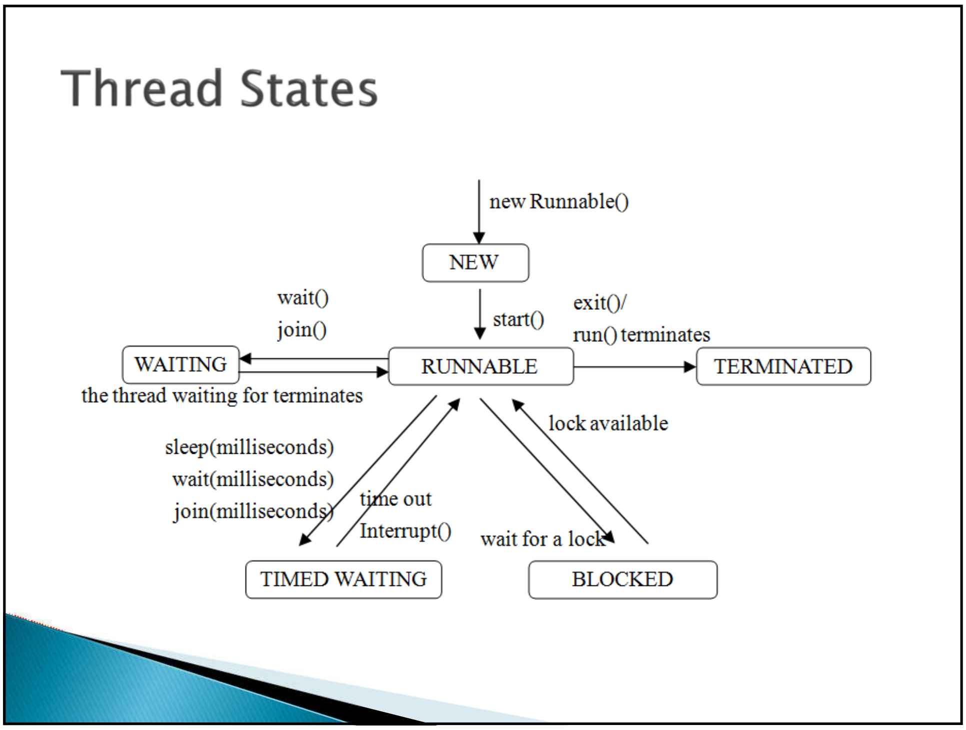
- Runnable is an interface in Java to create a thread that allows many threads to share the same thread object.
- The thread is a class in Java to create a thread where each thread has a unique object associated with it.
  - Memory
    - In Runnable, multiple threads share the same object, so require less memory
    - In Thread class, each thread creates a unique object, therefore requires more memory
  - Extending Ability
    - After implementing Runnable interface, it can extend a class.
    - After extending Thread class, it cannot extend any other class.
  - Code Maintainability
    - Runnable interface makes code more maintainable.
    - In Thread class, maintaining is time-consuming.



Thread States

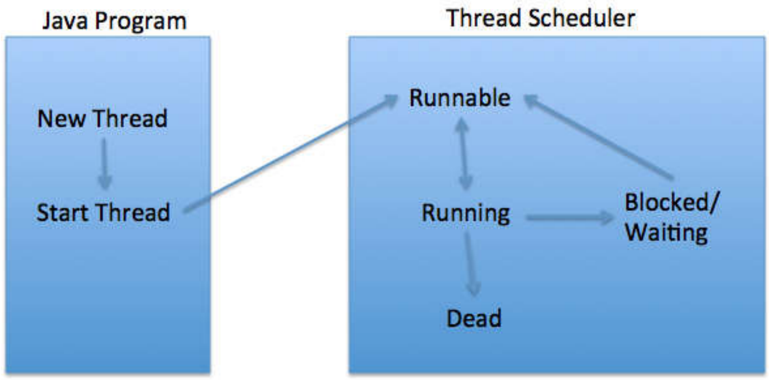
Table 9.1 The six states of a thread

State	Description
`NEW`	A thread that has not yet started is in this state.
`RUNNABLE`	A thread executing in the Java virtual machine.
`BLOCKED`	A thread that is blocked waiting for a monitor lock.
`WAITING`	A thread that is waiting indefinitely for another thread to perform a particular action.
`TIMED_WAITING`	A thread that is waiting for another thread to perform an action for up to a specified waiting time.
`TERMINATED`	A thread that has exited.



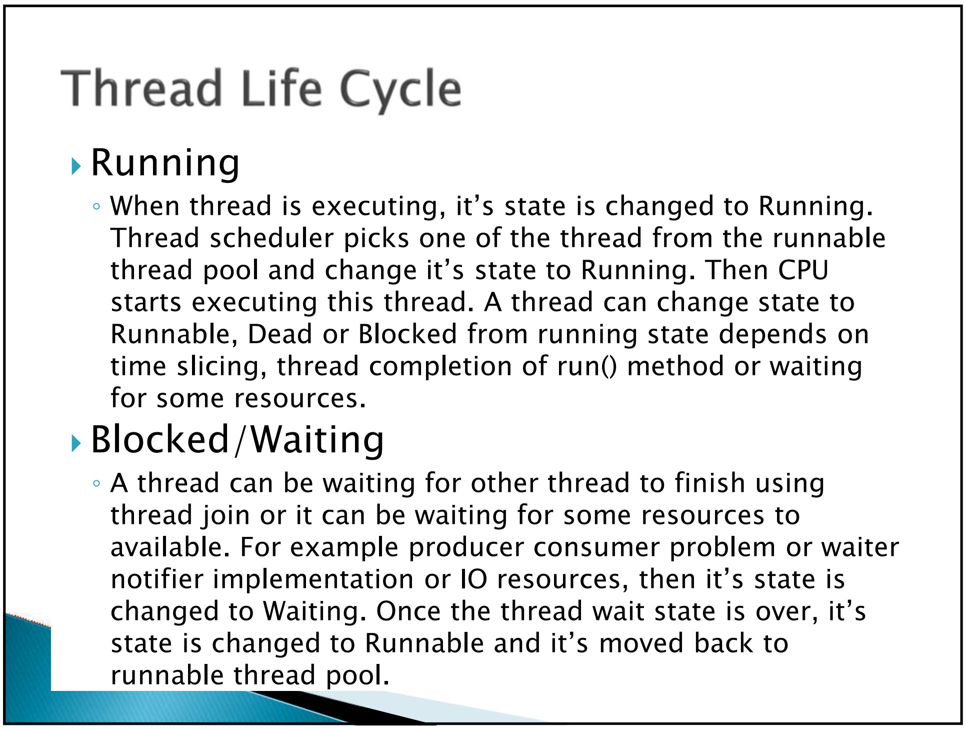
Thread States

Thread Life Cycle



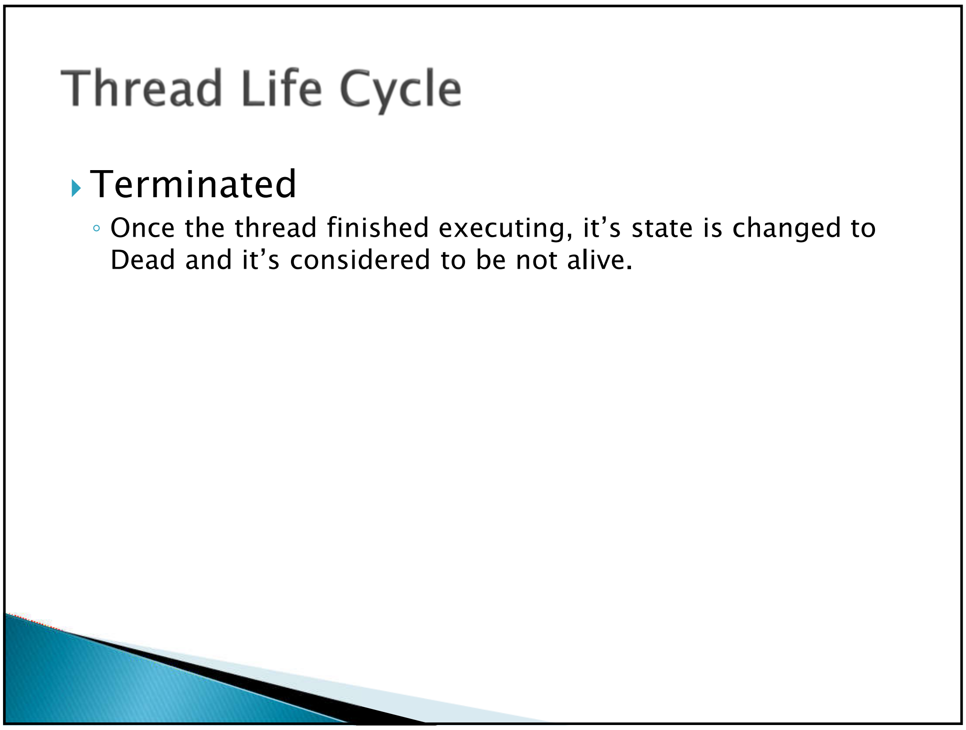
Thread Life Cycle

- New
  - When we create a new Thread object using new operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.
- Runnable
  - When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.



Thread Life Cycle

- Running
  - When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.
- Blocked/Waiting
  - A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available. For example producer consumer problem or waiter notifier implementation or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.



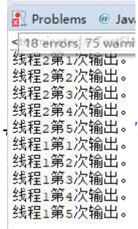
Thread Life Cycle

- Terminated
  - Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

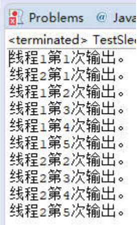
sleep() method

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

```
class SleepThread extends Thread {
    SleepThread(String s) {
        super(s);
    }
    public void run() {
        for (int i = 1; i <= 5; i++)
            System.out.println(this.getName() + "第"
    }
}
public class TestSleepMethod {
    public static void main(String[] args) {
        SleepThread s1 = new SleepThread("线程1");
        SleepThread s2 = new SleepThread("线程2");
        s1.start();
        s2.start();
    }
}
```



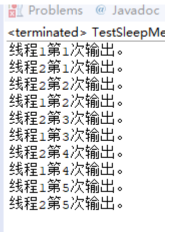
```
线程2第1次输出。
线程2第2次输出。
线程2第3次输出。
线程2第4次输出。
线程2第5次输出。
线程1第1次输出。
线程1第2次输出。
线程1第3次输出。
线程1第4次输出。
线程1第5次输出。
```



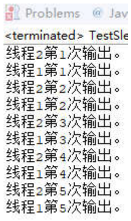
```
<terminated> TestSle
线程1第1次输出。
线程2第1次输出。
线程1第2次输出。
线程1第3次输出。
线程1第4次输出。
线程1第5次输出。
线程2第2次输出。
线程2第3次输出。
线程2第4次输出。
线程2第5次输出。
```

sleep() method

```
class SleepThread extends Thread {
    SleepThread(String s) {
        super(s);
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(this.getName() + "第" + i + "次输出。");
            try {
                sleep(500);
            } catch (InterruptedException e) {
            }
        }
    }
}
public class TestSleepMethod {
    public static void main(String[] args) {
        SleepThread s1 = new SleepThread("线程1");
        SleepThread s2 = new SleepThread("线程2");
        s1.start();
        s2.start();
    }
}
```



```
<terminated> TestSleepMe
线程1第1次输出。
线程2第1次输出。
线程2第2次输出。
线程1第2次输出。
线程2第3次输出。
线程1第3次输出。
线程2第4次输出。
线程1第4次输出。
线程1第5次输出。
线程2第5次输出。
```

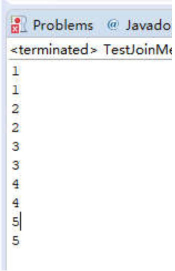


```
<terminated> TestSle
线程2第1次输出。
线程1第1次输出。
线程2第2次输出。
线程1第2次输出。
线程2第3次输出。
线程1第3次输出。
线程2第4次输出。
线程1第4次输出。
线程2第5次输出。
线程1第5次输出。
```

join () method

- The join() method causes the currently running threads to stop executing until the thread it joins with completes its task.

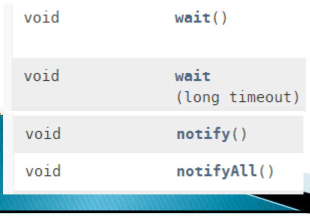
```
public class TestJoinMethod extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                System.out.println(e);
            }
                System.out.println(i);
        }
    }
    public static void main(String args[]) {
        TestJoinMethod t1 = new TestJoinMethod();
        TestJoinMethod t2 = new TestJoinMethod();
        TestJoinMethod t3 = new TestJoinMethod();
        t1.start();
        try {
            t1.join();
        } catch (Exception e) {
            System.out.println(e);
        }
        t2.start();
        t3.start();
    }
```



```
<terminated> TestJoinMe
1
1
2
2
3
3
4
4
5
5
```

wait () and notify() method

- Wait method defined in class Object, causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.



void	wait()
void	wait (long timeout)
void	notify()
void	notifyAll()

```
class WaitThread extends Thread {
    WaitThread(String s) {
    super(s);
}
public void run() {
     for (int i = 1; i <= 5; i++) {
     System.out.println(this.getName() + "第"
+ i + "次输出。");
     try {
        synchronized(this){
            this.wait(500);}
     } catch (InterruptedException e) {
     }
   }
  }
}
public class TestWaitMethod {
public static void main(String[] args) {
    WaitThread s1 = new WaitThread("线程1");
    WaitThread s2 = new WaitThread("线程2");
    s1.start();
    s2.start();
 }
}
```

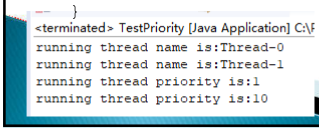
Other methods

- boolean isAlive()
  - Tests if this thread is alive. A thread is alive if it has been started and has not yet died.
- String getName()
  - Returns this thread's name.
- void setName(String name)
  - Changes the name of this thread.
- long getId()
  - Returns the identifier of this Thread.
- Deprecated Methods
  - suspend()
  - void resume()

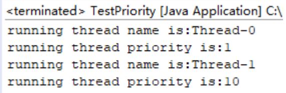
Thread Priority

- Each thread have a priority. Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority.

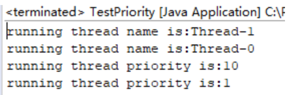
```
public class TestPriority extends Thread {
    public void run() {
        System.out.println("running thread name is:" +
Thread.currentThread().getName());
        System.out.println("running thread priority is:" +
Thread.currentThread().getPriority());
    }
    public static void main(String args[]) {
        TestPriority m1 = new TestPriority();
        TestPriority m2 = new TestPriority();
        m1.setPriority(1);
        m2.setPriority(10);
        m1.start();
        m2.start();
    }
}
```



```
<terminated> TestPriority [Java Application] C:\F
running thread name is:Thread-0
running thread name is:Thread-1
running thread priority is:1
running thread priority is:10
```



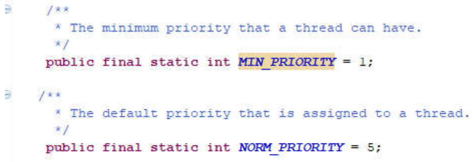
```
<terminated> TestPriority [Java Application] C:\
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10
```



```
<terminated> TestPriority [Java Application] C:\F
running thread name is:Thread-1
running thread name is:Thread-0
running thread priority is:10
running thread priority is:1
```

Thread Priority

- 3 constants defined in Thread class



```
/**
 * The minimum priority that a thread can have.
 */
public final static int MIN_PRIORITY = 1;

/**
 * The default priority that is assigned to a thread.
 */
public final static int NORM_PRIORITY = 5;
```

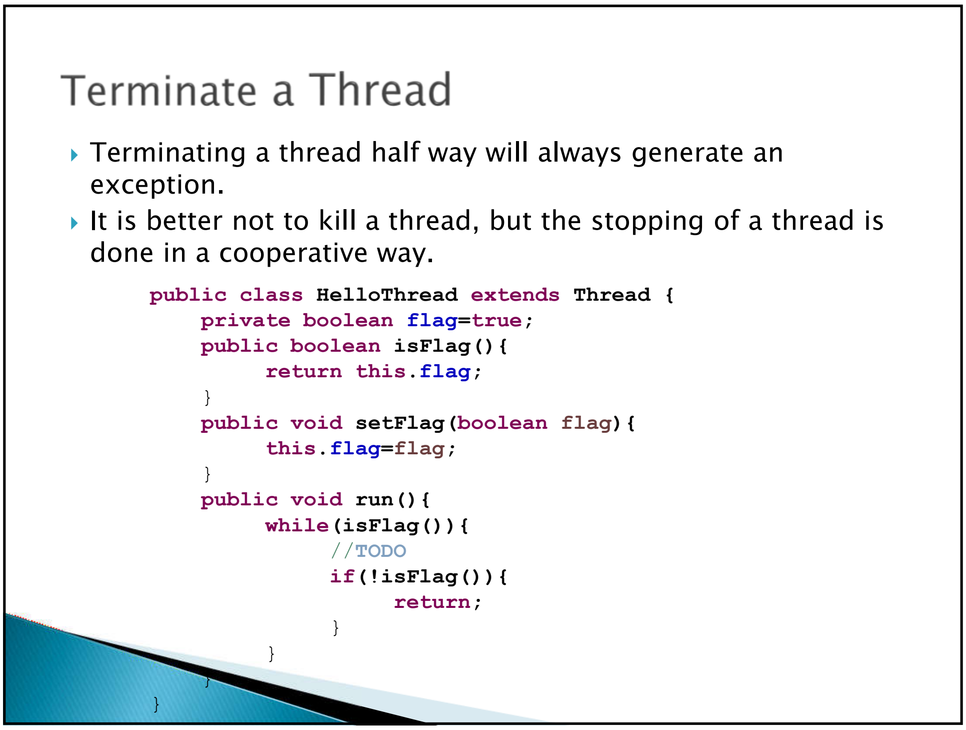
```
public class TestPriority extends Thread {
    public void run() {
        System.out.println("running thread name is:" +
Thread.currentThread().getName());
        System.out.println("running thread priority is:" +
Thread.currentThread().getPriority());
    }
    public static void main(String args[]) {
        TestPriority m1 = new TestPriority();
        TestPriority m2 = new TestPriority();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

Daemon Thread

- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

```
class Daemon extends Thread {
    public Daemon() {
        setDaemon(true);
    }
}
```

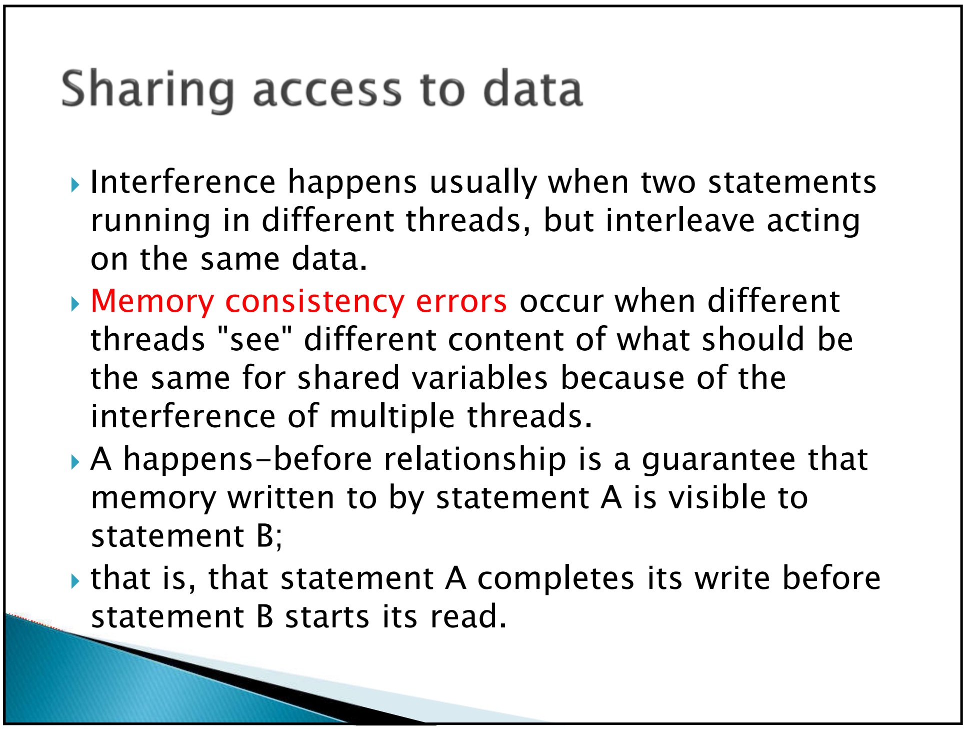
```
thread.isDaemon();
```



Terminate a Thread

- Terminating a thread half way will always generate an exception.
- It is better not to kill a thread, but the stopping of a thread is done in a cooperative way.

```
public class HelloThread extends Thread {
    private boolean flag=true;
    public boolean isFlag(){
         return this.flag;
    }
    public void setFlag(boolean flag){
         this.flag=flag;
    }
    public void run(){
         while(isFlag()){
              //TODO
              if(!isFlag()){
                   return;
              }
         }
    }
}
```



Sharing access to data

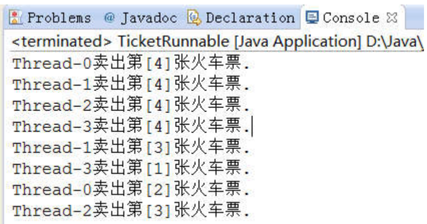
- Interference happens usually when two statements running in different threads, but interleave acting on the same data.
- Memory consistency errors occur when different threads "see" different content of what should be the same for shared variables because of the interference of multiple threads.
- A happens-before relationship is a guarantee that memory written to by statement A is visible to statement B;
- that is, that statement A completes its write before statement B starts its read.

Thread safety

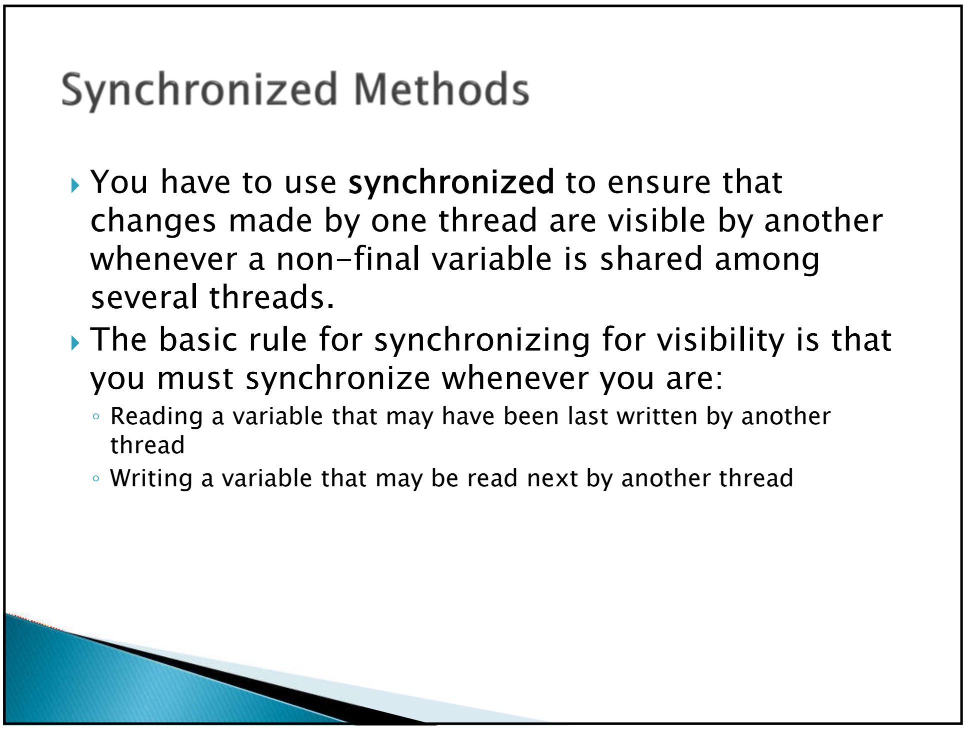
- Thread safety in java is the process to make our program safe to use in multithreaded environment, there are different ways through which we can make our program thread safe.
- Synchronization is the easiest and most widely used tool for thread safety in java.

Thread safety

```
class myRunnable implements Runnable {
private  int tickets = 4;
    public void run() {
        while (tickets > 0) {
        System.out.println(Thread.currentThread().getName()+"卖出第["+(tickets) +"]张火车票.");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            tickets--;
        }
    }
}
```

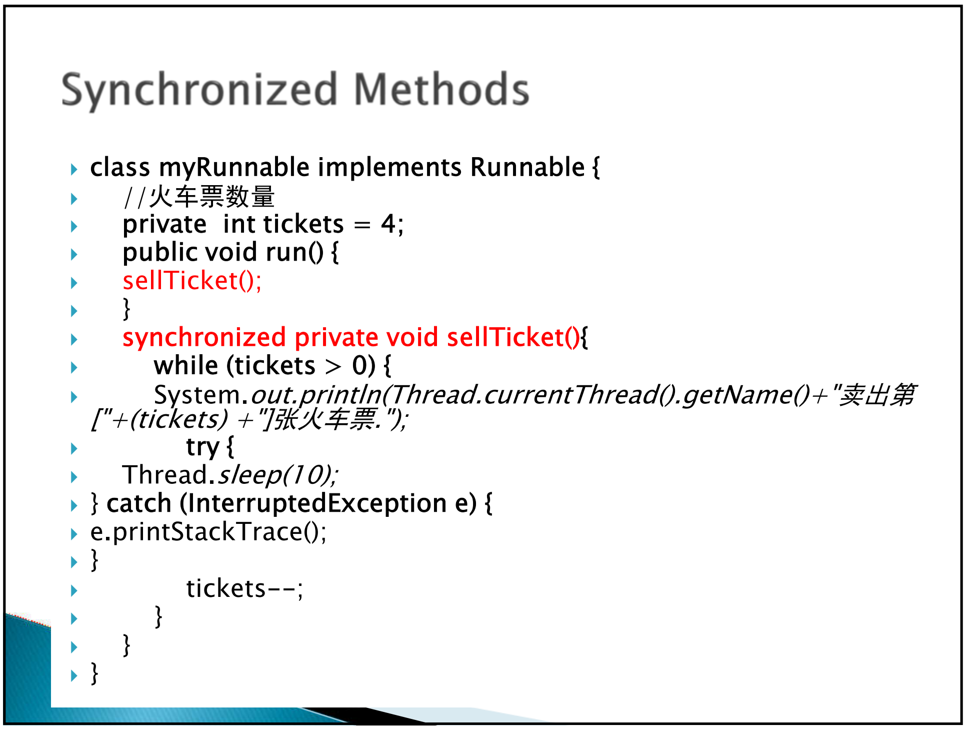


```
<terminated> TicketRunnable [Java Application] D:\Java\
Thread-0卖出第[4]张火车票.
Thread-1卖出第[4]张火车票.
Thread-2卖出第[4]张火车票.
Thread-3卖出第[4]张火车票.
Thread-1卖出第[3]张火车票.
Thread-3卖出第[1]张火车票.
Thread-0卖出第[2]张火车票.
Thread-2卖出第[3]张火车票.
```



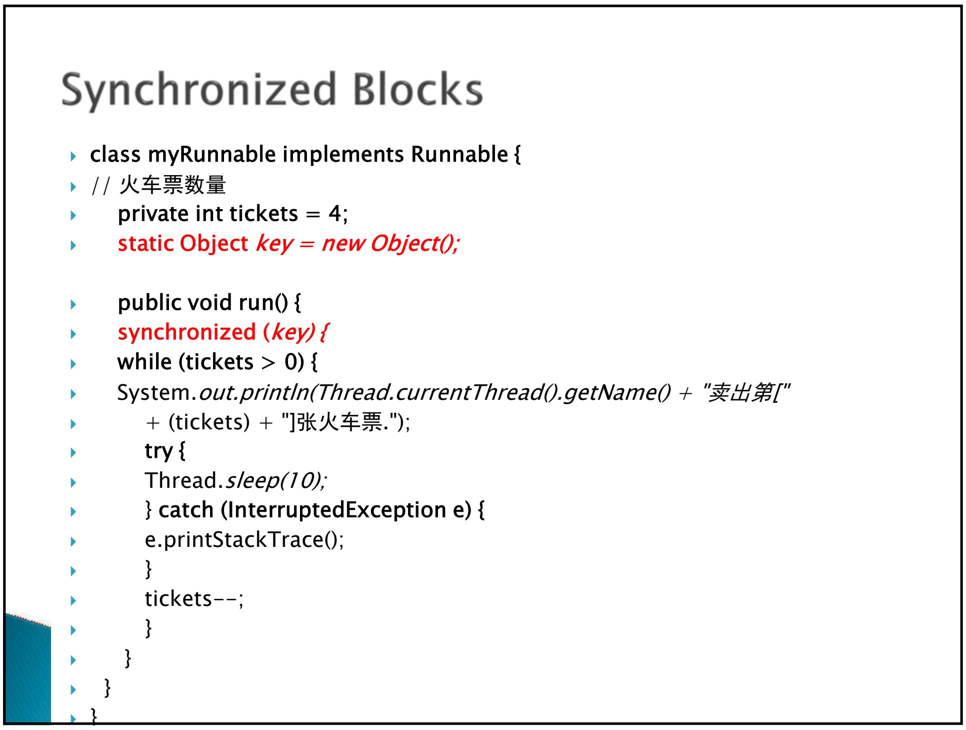
Synchronized Methods

- You have to use **synchronized** to ensure that changes made by one thread are visible by another whenever a non-final variable is shared among several threads.
- The basic rule for synchronizing for visibility is that you must synchronize whenever you are:
  - Reading a variable that may have been last written by another thread
  - Writing a variable that may be read next by another thread



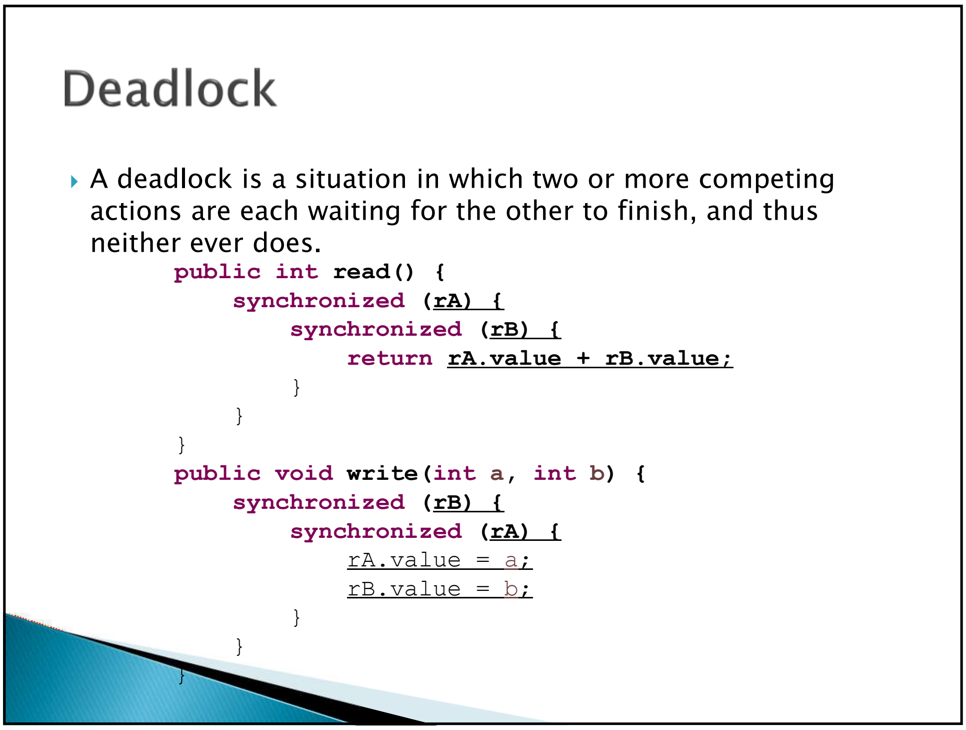
Synchronized Methods

```
class myRunnable implements Runnable {
    //火车票数量
    private  int tickets = 4;
    public void run() {
    sellTicket();
    }
    synchronized private void sellTicket(){
        while (tickets > 0) {
        System.out.println(Thread.currentThread().getName()+"卖出第
["+(tickets) +"]张火车票.");
            try {
    Thread.sleep(10);
} catch (InterruptedException e) {
e.printStackTrace();
}
            tickets--;
        }
    }
}
```



Synchronized Blocks

```
class myRunnable implements Runnable {
// 火车票数量
    private int tickets = 4;
    static Object key = new Object();

    public void run() {
    synchronized (key) {
    while (tickets > 0) {
    System.out.println(Thread.currentThread().getName() + "卖出第["
        + (tickets) + "]张火车票.");
        try {
        Thread.sleep(10);
        } catch (InterruptedException e) {
        e.printStackTrace();
        }
        tickets--;
        }
      }
   }
}
```



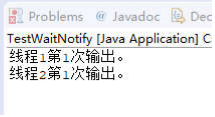
Deadlock

- A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

```
    public int read() {
        synchronized (rA) {
            synchronized (rB) {
                return rA.value + rB.value;
            }
        }
    }
    public void write(int a, int b) {
        synchronized (rB) {
            synchronized (rA) {
                rA.value = a;
                rB.value = b;
            }
        }
    }
```

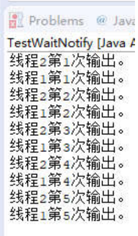
wait () and notify() method

```
class WaitThread extends Thread {
    final static Object monitor = new Object();
    WaitThread(String s) {
        super(s);
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(this.getName() + "第" + i + "次输出。");
            try {
                synchronized (monitor) {
                    monitor.notify();
                    monitor.wait();
                }
            } catch (InterruptedException e) {
            }
        }
    }
}
public class TestWaitNotify {
    public static void main(String[] args) {
        WaitThread s1 = new WaitThread("线程1");
        WaitThread s2 = new WaitThread("线程2");
        s1.start();
        s2.start();
    }
}
```



TestWaitNotify [Java Application] C

```
线程1第1次输出。
线程2第1次输出。
```



TestWaitNotify [Java A

```
线程2第1次输出。
线程1第1次输出。
线程2第2次输出。
线程1第2次输出。
线程2第3次输出。
线程1第3次输出。
线程2第4次输出。
线程1第4次输出。
线程2第5次输出。
线程1第5次输出。
```