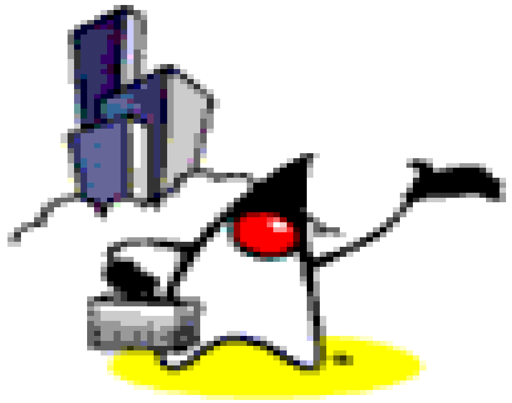
Java Threads

Sang Shin
Michèle Garoche
www.javapassion.com
"Learn with Passion!"

Topics

- What is a thread?
- Thread states
- Thread priorities
- Thread class
- Two ways of creating Java threads
  - Extending Thread class
  - Implementing Runnable interface
- ThreadGroup
- Synchronization
- Inter-thread communication
- Scheduling a task via Timer and TimerTask

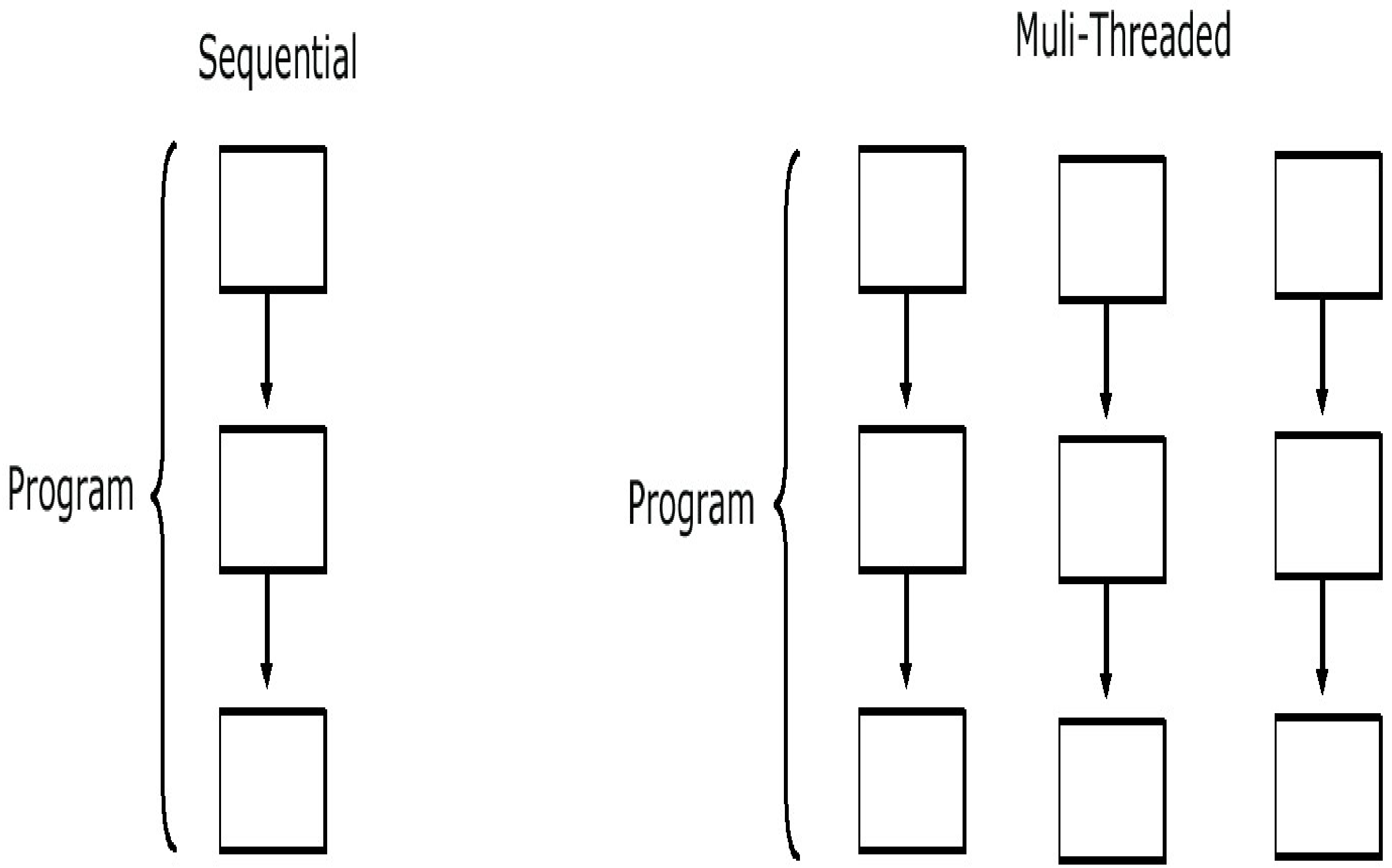


What is a Thread?

Threads

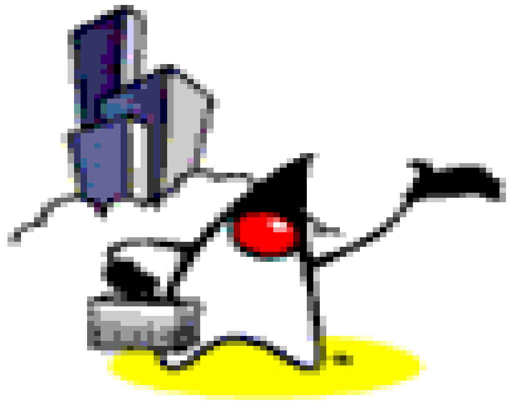
- Definition
  - Single sequential flow of control within a program to perform a task
- Why threads?
  - Need to handle concurrent operations

Single-threading vs. Multi-threading



Multi-threading in Java Platform

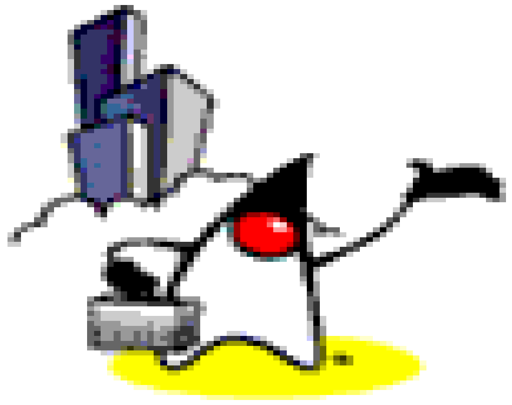
- Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling
- But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads



Thread States

Thread States

- A thread can in one of several possible states:
  1. Running
     - Currently running
     - In control of CPU
  2. Ready to run
     - Can run but is not yet given the chance
  3. Resumed
     - Ready to run after being suspended or blocked
  4. Suspended
     - Voluntarily allowed other threads to run
  5. Blocked
     - Waiting for some resource or event to occur



Thread Priorities

Thread Priorities

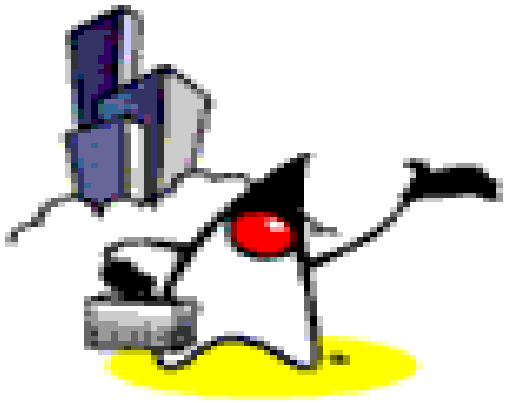
- Why priorities?
  - Determine which thread receives CPU control and gets to be executed first
- Definition:
  - Integer value ranging from 1 to 10
  - Higher the thread priority → larger chance of being executed first
  - Example:
    - Two threads are ready to run
    - First thread: priority of 5, already running
    - Second thread = priority of 10, comes in while first thread is running

Thread Priorities

- Context switch
  - Occurs when a thread snatches the control of CPU from another
  - When does it occur?
    - Running thread voluntarily relinquishes CPU control
    - Running thread is preempted by a higher priority thread
- More than one highest priority thread that is ready to run
  - Deciding which receives CPU control depends on the operating system
  - Windows: Uses time-sliced round-robin
  - Solaris: Executing thread should voluntarily relinquish CPU control

Demo: View Threads of a Java GUI Application

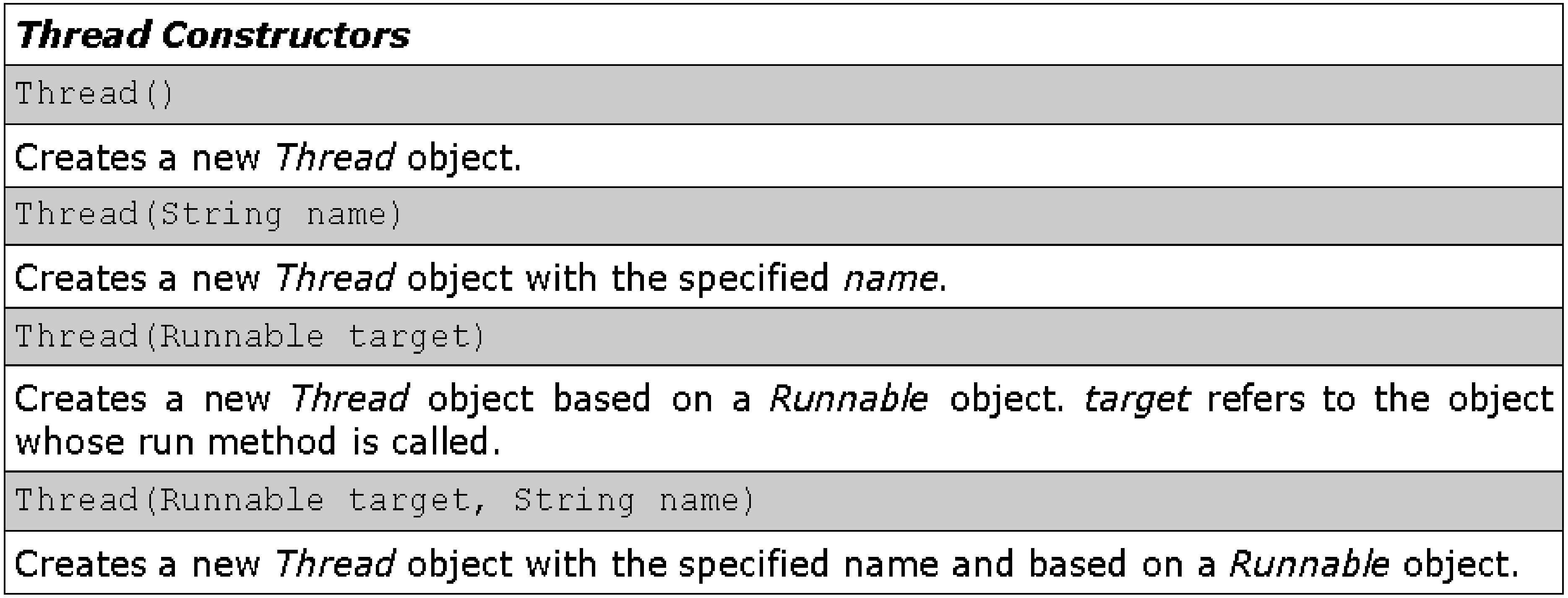
Run “ViewAllThreads” programs of 1021_javase_threads.zip



Thread Class

The *Thread* Class: Constructor

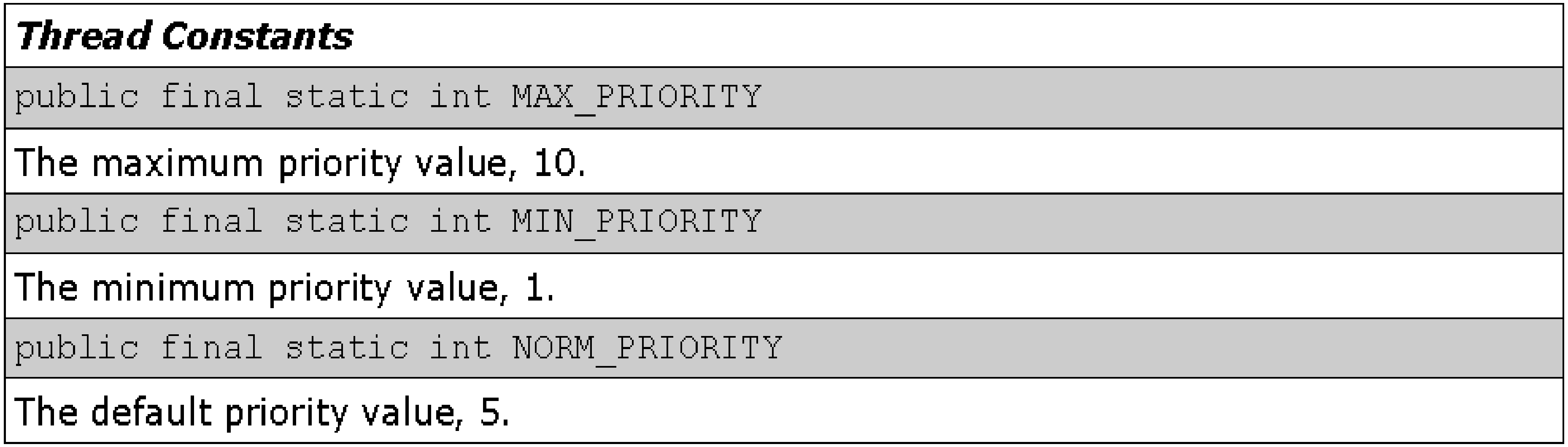
- Has eight constructors



Thread Constructors
`Thread()`
Creates a new *Thread* object.
`Thread(String name)`
Creates a new *Thread* object with the specified *name*.
`Thread(Runnable target)`
Creates a new *Thread* object based on a *Runnable* object. *target* refers to the object whose run method is called.
`Thread(Runnable target, String name)`
Creates a new *Thread* object with the specified name and based on a *Runnable* object.

The *Thread* Class: Constants

- Contains fields for priority values

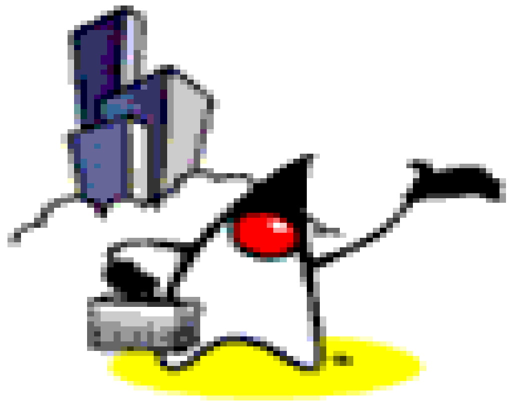


Thread Constants
`public final static int MAX_PRIORITY`
The maximum priority value, 10.
`public final static int MIN_PRIORITY`
The minimum priority value, 1.
`public final static int NORM_PRIORITY`
The default priority value, 5.

The *Thread* Class: Methods

- Some *Thread* methods

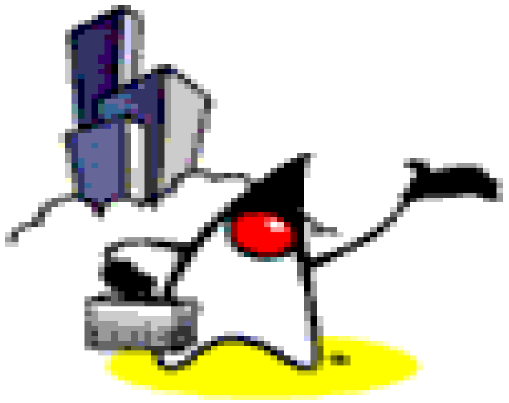
Thread Methods
`public static Thread currentThread()`
Returns a reference to the thread that is currently running.
`public final String getName()`
Returns the name of this thread.
`public final void setName(String name)`
Renames the thread to the specified argument *name*. May throw *SecurityException*.
`public final int getPriority()`
Returns the priority assigned to this thread.
`public final boolean isAlive()`
Indicates whether this thread is running or not.



Two Ways of Programming for Creating Java Threads

Two Ways of Programming for Creating and Starting a Thread

1. Extending the *Thread* class
2. Implementing the *Runnable* interface



Extending Thread Class

Extending Thread Class

- The subclass extends *Thread* class
  - The subclass overrides the *run()* method of *Thread* class
- An object instance of the subclass can then be created
- Calling the *start()* method of the object instance of the subclass starts the execution of the thread
  - Java runtime starts the execution of the thread by calling *run()* method of object instance

Two Schemes of starting a thread from a subclass

- Scheme #1: The *start()* method is not in the constructor of the subclass
  - The start() method needs to be explicitly invoked after object instance of the subclass is created in order to start the thread
- Scheme #2: The *start()* method is in the constructor of the subclass
  - Creating an object instance of the subclass will start the thread automatically

Scheme #1: start() method is Not in the constructor of subclass

```
class PrintNameThread extends Thread {
    PrintNameThread(String name) {
        super(name);
    }
    public void run() {
        String name = getName();
        for (int i = 0; i < 100; i++) {
            System.out.print(name);
        }
    }
}
//continued
```

Scheme #1: start() method needs to be called explicitly

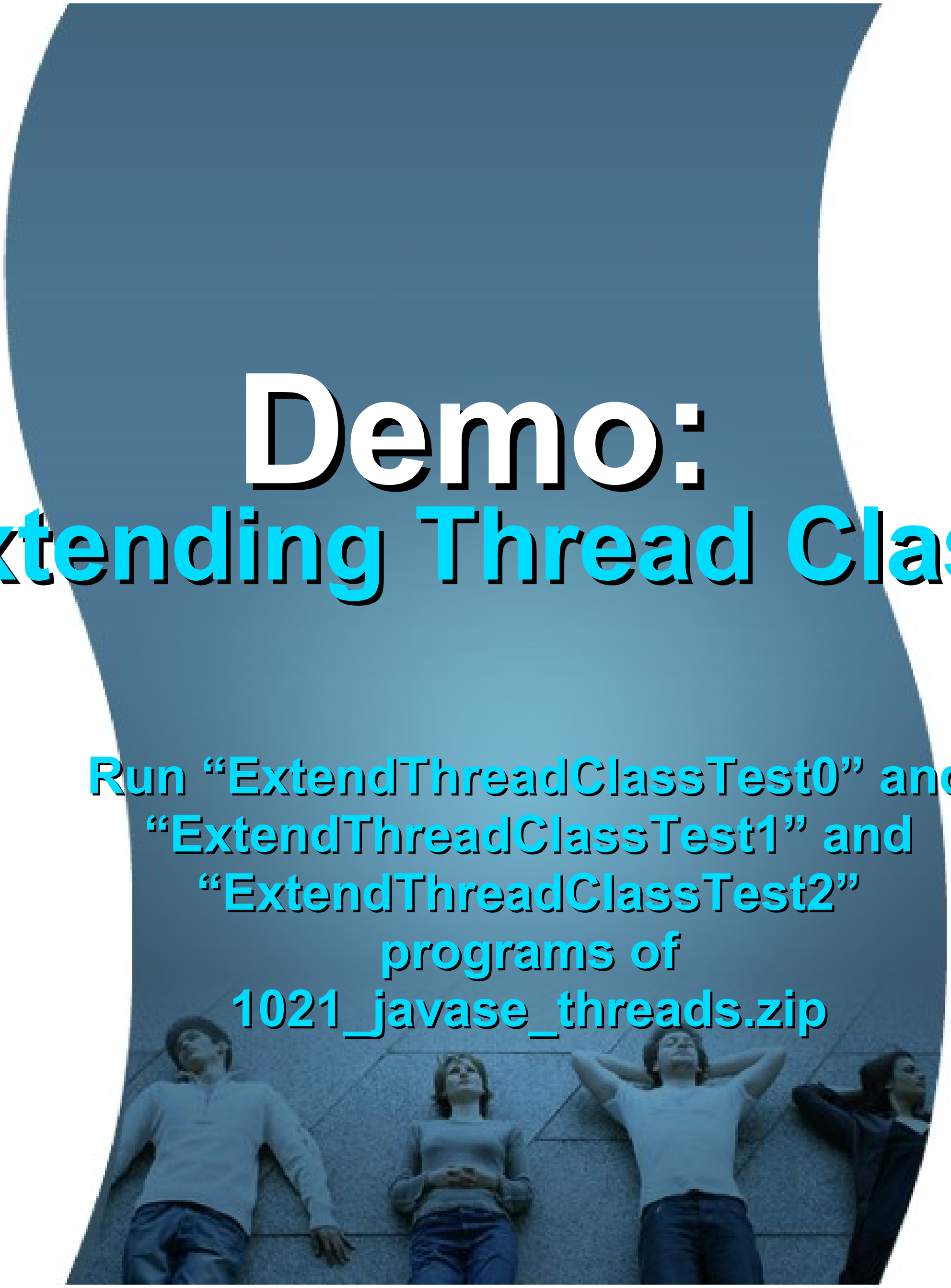
```
class ExtendThreadClassTest1 {
    public static void main(String args[]) {
        PrintNameThread pnt1 =
                        new PrintNameThread("A");
        pnt1.start(); // Start the first thread
        PrintNameThread pnt2 =
                        new PrintNameThread("B");
        pnt2.start(); // Start the second thread

    }
}
```

Scheme #2: start() method is in a constructor of the subclass

```
class PrintNameThread extends Thread {
    PrintNameThread(String name) {
        super(name);
        start(); //runs the thread once instantiated
    }
    public void run() {
        String name = getName();
        for (int i = 0; i < 100; i++) {
            System.out.print(name);
        }
    }
}
//continued
```

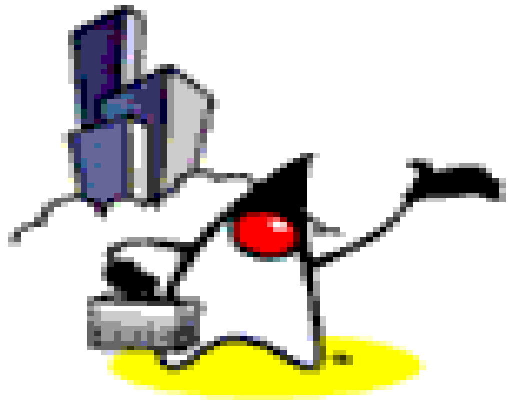
Scheme #2: Just creating an object instance starts a thread

```
class ExtendThreadClassTest3 {
   public static void main(String args[]) {
      new PrintNameThread("A");
      new PrintNameThread("B");
   }
}
```



Demo: Extending Thread Class

Run "ExtendThreadClassTest0" and "ExtendThreadClassTest1" and "ExtendThreadClassTest2" programs of 1021_javase_threads.zip



Implementing Runnable Interface

Runnable Interface

- The *Runnable* interface should be implemented by any class whose instances are intended to be executed as a thread
- The class must define *run()* method of no arguments
  - The *run()* method is like *main()* for the new thread

Why Use Runnable Interface?

- A class that implements *Runnable* can run without subclassing *Thread* by instantiating a *Thread* instance and passing itself in as the target

Two Ways of Starting a Thread For a class that implements Runnable

- Scheme #1: Caller thread creates a Thread object and starts it explicitly after an object instance of the class that implements *Runnable* interface is created
  - The *start()* method of the Thread object needs to be explicitly invoked after object instance is created
- Scheme #2: The *Thread* object is created and started within the constructor method of the class that implements *Runnable* interface
  - The caller thread just needs to create object instances of the *Runnable* class

Scheme #1: Caller thread creates a Thread object and starts it explicitly

```
// PrintNameRunnable implements Runnable interface
class PrintNameRunnable extends SomeClass
                            implements Runnable  {

    String name;

    PrintNameRunnable(String name) {
        this.name = name;
    }

    // Implementation of the run() defined in the
    // Runnable interface.
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(name);
        }
    }
}
```

Scheme #1: Caller thread creates a Thread object and starts it explicitly

```
public class RunnableThreadTest1 {

    public static void main(String args[]) {

        PrintNameRunnable pnt1 = new PrintNameRunnable("A");
        Thread t1 = new Thread(pnt1);
        t1.start();

    }
}
```

Scheme #2: Thread object is created and started within a constructor

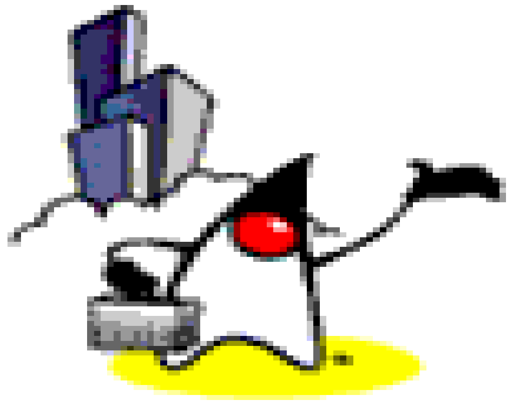
```
// PrintNameRunnable implements Runnable interface
class PrintNameRunnable extends SomeClass
                        implements Runnable {

    Thread thread;

    PrintNameRunnable(String name) {
        thread = new Thread(this, name);
        thread.start();
    }

    // Implementation of the run() defined in the
    // Runnable interface.
    public void run() {
        String name = thread.getName();
        for (int i = 0; i < 10; i++) {
            System.out.print(name);
        }
    }
}
```

Scheme #2: Thread object is created and started within a constructor

```
public class RunnableThreadTest2 {

    public static void main(String args[]) {

        // Since the constructor of the PrintNameRunnable
        // object creates a Thread object and starts it,
        // there is no need to do it here.
        new PrintNameRunnable("A");

        new PrintNameRunnable("B");
        new PrintNameRunnable("C");
    }
}
```

Demo:
Implementing Runnable Interface

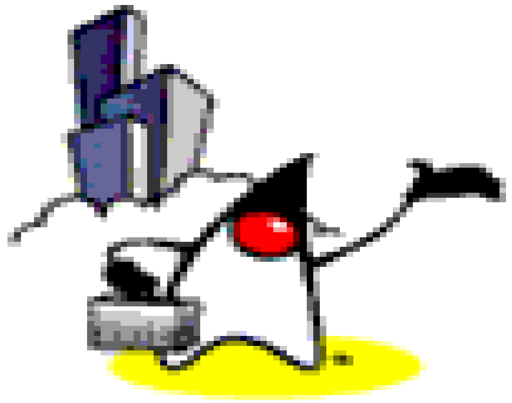
Run “RunnableThreadTest1” and “RunnableThreadTest2” programs of 1021_javase_threads.zip



Extending Thread Class vs. Implementing Runnable Interface

Extending Thread vs. Implementing Runnable Interface

- Choosing between these two is a matter of taste
- Implementing the *Runnable* interface
  - May take more work since we still
    - Declare a *Thread* object
    - Call the *Thread* methods on this object
  - Your class can still extend other class
- Extending the *Thread* class
  - Easier to implement
  - Your class can no longer extend any other class, however



ThreadGroup

ThreadGroup Class

- A thread group represents a set of threads
- In addition, a thread group can also include other thread groups
  - The thread groups form a tree in which every thread group except the initial thread group has a parent
- A thread is allowed to access information about its own thread group, but not to access information about its thread group's parent thread group or any other thread group

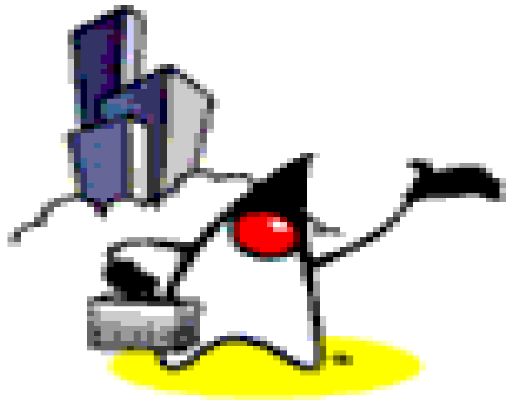
Example: ThreadGroup

```
// Start three threads first.  They should
// belong to a same ThreadsGroup.
new SimpleThread("Jamaica").start();
new SimpleThread("Fiji").start();
new SimpleThread("Bora Bora").start();

// Get ThreadGroup of the current thread
ThreadGroup group
 = Thread.currentThread().getThreadGroup();

// Display the names of the threads in the
// current ThreadGroup.
Thread[] tarray = new Thread[10];
int actualSize = group.enumerate(tarray);
for (int i=0; i<actualSize;i++){
    System.out.println("Thread " +
    tarray[i].getName() + " in thread group "
     + group.getName());
}
```

Demo:

ThreadGroup

Run “ThreadGroupTest” program of 1021_javase_threads.zip



Synchronization

Race condition & How to Solve it

- Race conditions occur when multiple simultaneously executing threads access the same object (called a shared resource) returning unexpected (wrong) results
- Example:
  - Threads often need to share a common resource ie a file, with one thread reading from the file while another thread writes to the file
- They can be avoided by synchronizing the threads which access the shared resource

An Unsynchronized Example

```
class PrintStringsThread implements Runnable {
    Thread thread;
    String str1, str2;
    PrintStringsThread(String str1, String str2) {
        this.str1 = str1;
        this.str2 = str2;
        thread = new Thread(this);
        thread.start();
    }
    public void run() {
        TwoStrings.print(str1, str2);
    }
}
//continued...
```

An Unsynchronized Example

```
class TestThread {
    public static void main(String args[]) {
        new PrintStringsThread("Hello ", "there.");
        new PrintStringsThread("How are ", "you?");
        new PrintStringsThread("Thank you ",
                                               "very much!");
    }
}
```

An Unsynchronized Example

- Sample output:

```
Hello How are Thank you there.
you?
very much!
```

Demo:

Not Synchronized (Race Condition)

Run
“SynchronizedExample0_Unsynchronized”
program of
1021_javase_threads.zip

Synchronization Lock in Java

- Java offers a synchronization monitor on each instance of the Object class, so it can be used as a synchronization lock
  - In other words, every object in Java can be used as a synchronization lock

Synchronization: Locking an Object

- Threads are synchronized through an object's monitor (lock)
- Only the thread who owns the object's monitor (owner thread of the lock) can execute the block of code that needs to be synchronized (synchronized block)
- When synchronized block finishes, the object monitor (lock) is then released – any other thread waiting for the object monitor will be then chosen to be the owner of the object monitor

Synchronization: Locking an Object

- A thread becomes the owner of the object's monitor in one of three ways
  - Option 1: Use *synchronized* static method
  - Option 2: Use *synchronized* instance method
  - Option 3: Use *synchronized* statement on a common object

Option 1: Use synchronized Static Method

```
class TwoStrings {
  // All the statements in the method become the
  // synchronized block, and the class object is the
  // lock.
  synchronized static void print(String str1,
                                 String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

Option 1: Use synchronized Static method

```
class PrintStringsThread implements Runnable {
    Thread thread;
    String str1, str2;
    PrintStringsThread(String str1, String str2) {
        this.str1 = str1;
        this.str2 = str2;
        thread = new Thread(this);
        thread.start();
    }
    public void run() {
        TwoStrings.print(str1, str2);
    }
}
//continued...
```

Option 1: Use synchronized Static method

```
class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ",
                                  "very much!");
   }
}
```

Option 1: Executing Synchronized Static Method

- Sample output:

```
Hello there.
How are you?
Thank you very much!
```

Option 2: Use synchronized Static Method

```
class TwoStrings {
  // All the statements in the method become the
  // synchronized block, and and the instance object
  // is the lock.
  synchronized void print(String str1,
                                String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

Demo:
Use “synchronized static method”

Run “SynchronizedExample1” program of 1021_javase_threads.zip

Option 3: Use synchronized statement on a common object

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
//continued...
```

Demo:

Use “synchronized instance method”

Run “SynchronizedExample2” & “SynchronizedExample2a” & “SynchronizedExample2b” program of 1021_javase_threads.zip

Option 3: Use synchronized statement on a common object

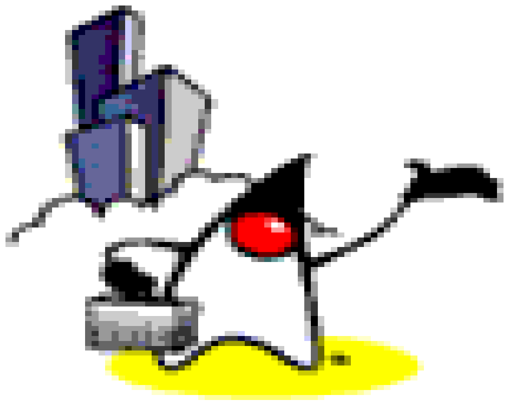
```
class PrintStringsThread implements Runnable {
    Thread thread;
    String str1, str2;
    TwoStrings ts;
    PrintStringsThread(String str1, String str2,
                              TwoStrings ts) {
        this.str1 = str1;
        this.str2 = str2;
        this.ts = ts;
        thread = new Thread(this);
        thread.start();
    }
//continued...
```

Option 3: Use synchronized statement on a common object

```
   public void run() {
      // All the statements specified in the
      // parentheses of the synchronized statement
      // become the synchronized block, and the
      // object specified in the statement is the
      // lock.
      synchronized (ts) {
         ts.print(str1, str2);
      }
   }
}
class TestThread {
   public static void main(String args[]) {
      TwoStrings ts = new TwoStrings();
      new PrintStringsThread("Hello ", "there.", ts);
      new PrintStringsThread("How are ", "you?", ts);
      new PrintStringsThread("Thank you ",
                                     "very much!", ts);
}}
```

Demo:
Use “synchronized object”

Run “SynchronizedExample3” program of 1021_javase_threads.zip



Inter-thread Communication: Producer-Consumer Example

Producer-Consumer

- Imagine a scenario in which there exists two distinct threads both operating on a single shared data area
- One thread, the Producer inserts data into the data area while the other thread, the Consumer, removes data from that same area
- In order for the Producer to insert data into the data area, there must be enough space
  - The Producer's sole function is to insert data into the data-area, it is not allowed to remove any data from the area.

Producer-Consumer (Continued)

- For the Consumer to be able to remove data from the data area, there must be data there in the first place
  - The sole function of the Consumer is to remove data from the data area
- The solution of the Producer-Consumer problem lies with devising a suitable communication protocol through which the two threads may exchange information.
- The definition of such a protocol is the main factor that makes the Producer-Consumer problem interesting in terms of concurrent systems

Unsynchronized Producer-Consumer Example: CubbyHole.java

```
public class CubbyHole {
    private int contents;

    // Get the data
    public int get() {
        return contents;
    }

    // Place the data
    public synchronized void put(int value) {
        contents = value;
    }
}
```

Unsynchronized Producer-Consumer Example: Producer.java

```java
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                                   + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Unsynchronized Producer-Consumer Example: Consumer.java

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                                 + " got: " + value);
        }
    }
}

```

Unsynchronized Producer-Consumer Example: Main program

```
public class ProducerConsumerUnsynchronized {

    public static void main(String[] args) {

        CubbyHole c = new CubbyHole();

        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        // There is no inter-thread communication
        p1.start();
        c1.start();
    }
}
```

Result of Unsynchronized Producer-Consumer

- CPU resource is being wasted
  - The consumer has to do polling wasting CPU cycles while it waited for the producer to produce.
  - Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- Results are unpredictable
  - A number (as data) may be read before a number (as data) has been produced
  - Multiple numbers (data items) may be produced with only one or two being read

Result of Unsynchronized Producer-Consumer

```
Consumer #1 got: 0
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```

Demo:
Unsynchronized Producer-Consumer

Run “ProducerConsumerUnsynchronized” program of 1021_javase_threads.zip

Inter-thread Communication: Methods from Object Class

Methods for Interthread Communication
`public final void wait()`
Causes this thread to wait until some other thread calls the *notify* or *notifyAll* method on this object. May throw *InterruptedException*.
`public final void notify()`
Wakes up a thread that called the *wait* method on the same object.
`public final void notifyAll()`
Wakes up all threads that called the *wait* method on the same object.

wait() method of Object Class

- *wait()* method is defined in the Object class
  - Every Java class inherits the *wait()* method
- *wait()* method causes a thread to release the lock it is holding on an object; allowing another thread to run
- *wait()* can only be invoked from within synchronized code
- it should always be wrapped in a try block as it throws *IOException*
- *wait()* can only invoked by the thread that owns the lock on the object

wait() method of Object Class

- When *wait()* is called, the thread becomes dormant until one of four things occur:
  - another thread invokes the *notify()* method for this object and the scheduler arbitrarily chooses to run the thread
  - another thread invokes the *notifyAll()* method for this object
  - another thread interrupts this thread
  - the specified *wait()* time elapses
- When one of the above occurs, the thread becomes re-available to the Thread scheduler and competes for a lock on the object
- Once it regains the lock on the object, it resumes where it became dormant

notify() method

- Wakes up a single thread that is waiting on this object's monitor
  - If any threads are waiting on this object, one of them is chosen to be awakened
  - The choice is arbitrary and occurs at the discretion of the implementation
- Can only be used within synchronized code
- The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object

notifyAll() method

- Wakes up all threads that are waiting on this object's monitor
- The highest priority thread will run first.

Synchronized Producer-Consumer Example: CubbyHole.java

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        // When data is not available, wait
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        // Now data is available, wake up all threads
        // waiting for the lock to be released
        available = false;
        notifyAll();
        return contents;
    }
    // continued
```

Synchronized Producer-Consumer Example: CubbyHole.java

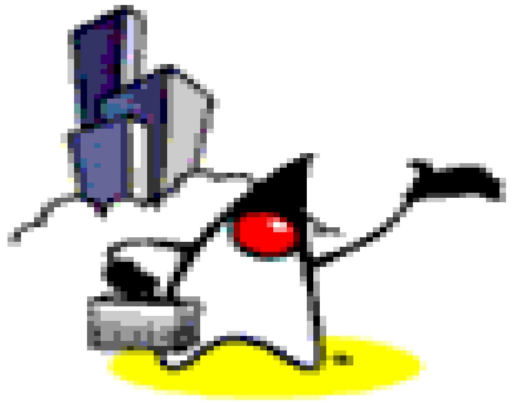
```

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}
```

Result of Synchronized Producer-Consumer

```
Producer 1 put: 0
Consumer 1 got: 0
Producer 1 put: 1
Consumer 1 got: 1
Producer 1 put: 2
Consumer 1 got: 2
Producer 1 put: 3
Consumer 1 got: 3
Producer 1 put: 4
Consumer 1 got: 4
Producer 1 put: 5
Consumer 1 got: 5
Producer 1 put: 6
Consumer 1 got: 6
Producer 1 put: 7
Consumer 1 got: 7
Producer 1 put: 8
Consumer 1 got: 8
Producer 1 put: 9
Consumer 1 got: 9
```

Demo:
Synchronized Producer-Consumer

Run “ProducerConsumerSynchronized” program of 1021_javase_threads.zip



Scheduling a task via Timer & TimerTask Classes

Timer Class

- Provides a facility for threads to schedule tasks for future execution in a background thread
- Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.
- Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially
- Timer tasks should complete quickly
  - If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

TimerTask Class

- Abstract class with an abstract method called run()
- Concrete class must implement the run() abstract method

Example: Timer Class

```
public class TimerReminder {

    Timer timer;

    public TimerReminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.format("Time's up!%n");
            timer.cancel(); //Terminate the timer thread
        }
    }

    public static void main(String args[]) {
        System.out.format("About to schedule task.%n");
        new TimerReminder(5);
        System.out.format("Task scheduled.%n");
    }
}
```

Thank you!

Sang Shin
Michèle Garoche
http://www.javapassion.com
“Learn with Passion!”