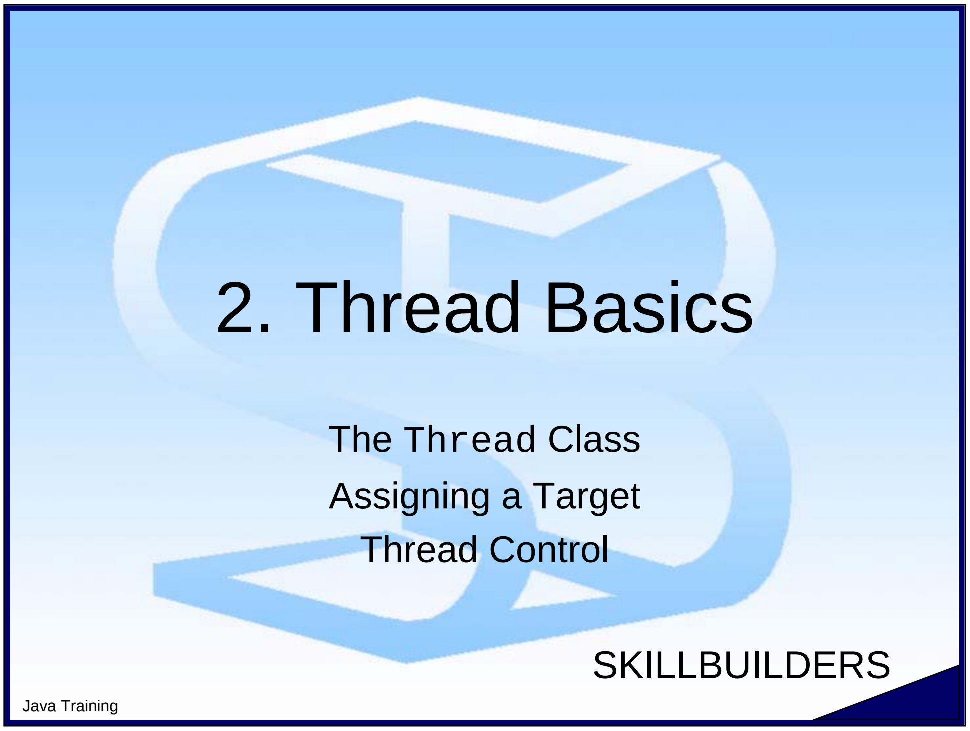


2. Thread Basics

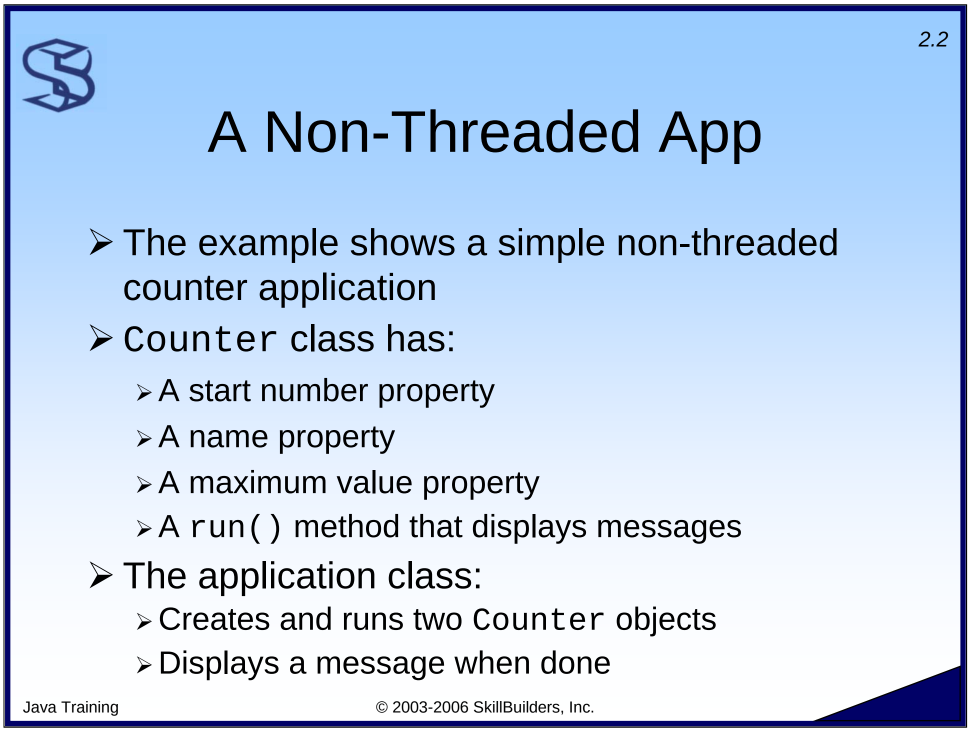
The `Thread` Class

Assigning a Target

Thread Control

SKILLBUILDERS

Java Training



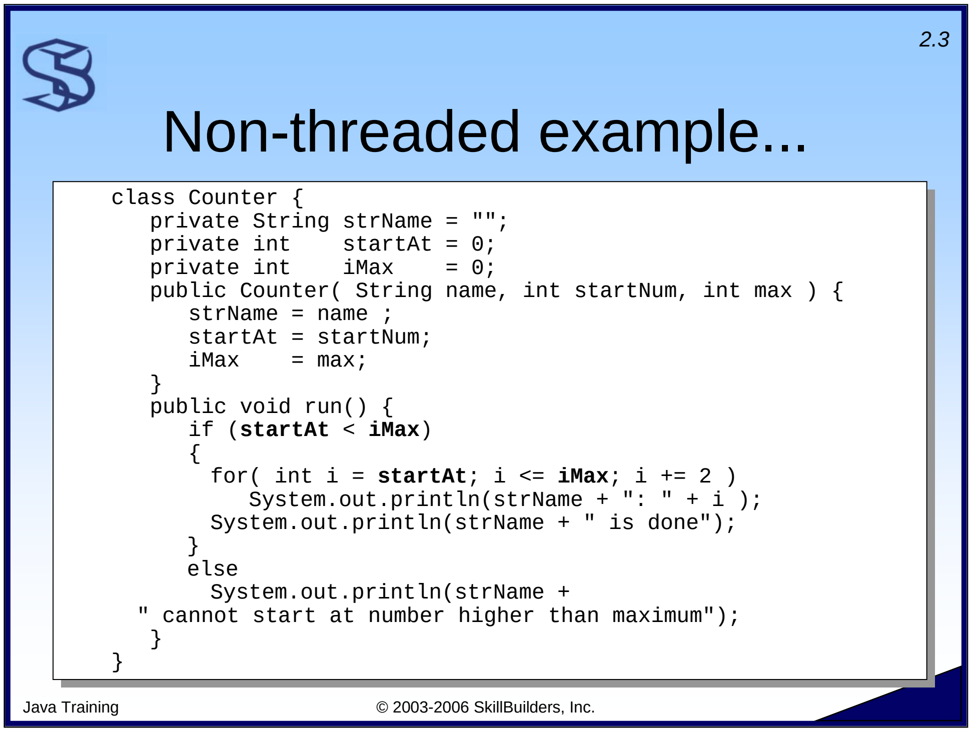
2.2

A Non-Threaded App

- The example shows a simple non-threaded counter application
- `Counter` class has:
  - A start number property
  - A name property
  - A maximum value property
  - A `run()` method that displays messages
- The application class:
  - Creates and runs two `Counter` objects
  - Displays a message when done

Java Training © 2003-2006 SkillBuilders, Inc.

To illustrate basic threading concepts we will start with a simple console application that displays counts to the screen without the use of threads.



2.3

Non-threaded example...

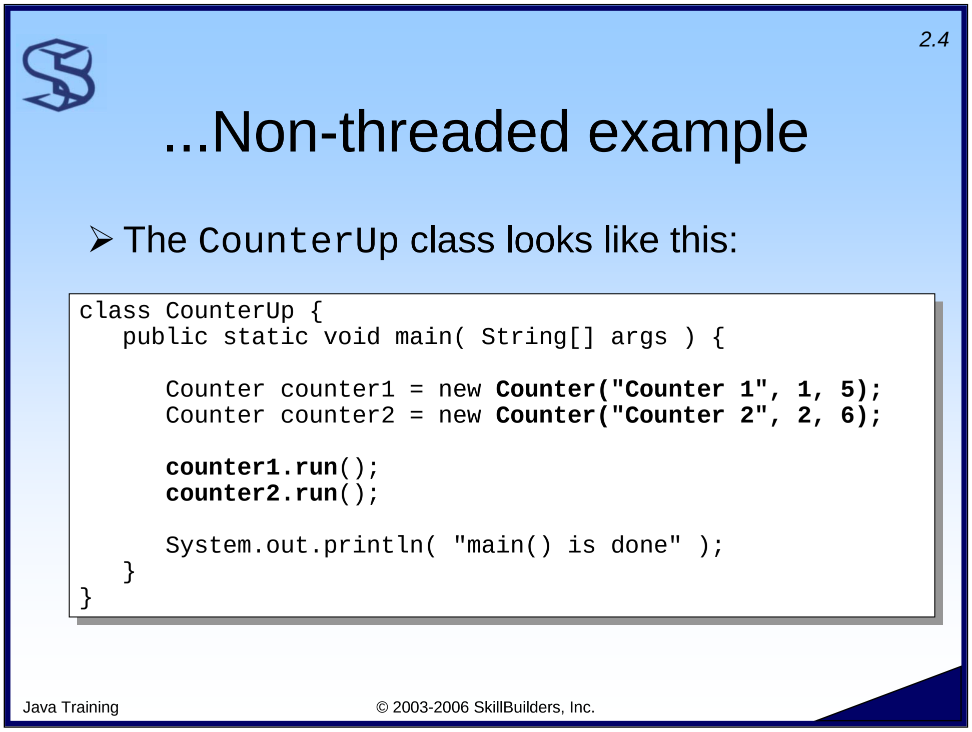
```
class Counter {
   private String strName = "";
   private int    startAt = 0;
   private int    iMax    = 0;
   public Counter( String name, int startNum, int max ) {
      strName = name ;
      startAt = startNum;
      iMax    = max;
   }
   public void run() {
      if (startAt < iMax)
      {
        for( int i = startAt; i <= iMax; i += 2 )
           System.out.println(strName + ": " + i );
        System.out.println(strName + " is done");
      }
      else
        System.out.println(strName +
  " cannot start at number higher than maximum");
   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

The `Counter` class has:

- A name property that is set via a constructor argument.
- A starting number to indicate where to begin counting set via a constructor argument.
- A maximum value property that is also set via a constructor argument.
- A `run()` method that uses a `for` loop to display a series of messages that includes the counter name, as assigned when the class was constructed, and the current count (**`i`**) number which starts at the input `startNum`, is incremented by 2 and continues until the `iMax` value (supplied in the constructor) is reached.

Note: the code includes a simple check with the `if (startAt < iMax)` to insure we do not try to begin with a number that is greater than our end point.



2.4

...Non-threaded example

- The `CounterUp` class looks like this:

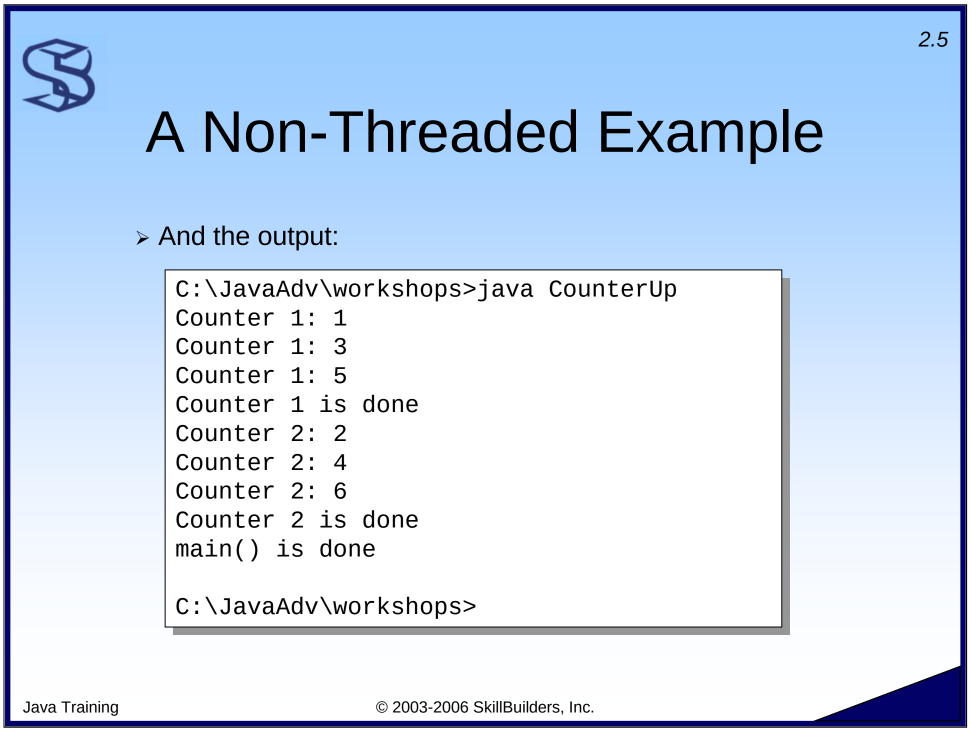
```
class CounterUp {
   public static void main( String[] args ) {

      Counter counter1 = new Counter("Counter 1", 1, 5);
      Counter counter2 = new Counter("Counter 2", 2, 6);

      counter1.run();
      counter2.run();

      System.out.println( "main() is done" );
   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

Counter Application

The `CounterUp` class is a console application. Its `main()` method:

1. Creates an instance of `Counter` with the name "`Counter 1`", that will start at 1 (to give us odd numbers) and a maximum of 5.
2. Creates an instance of `Counter` with the name "`Counter 2`", that will start at 2 (to give us even numbers) and a maximum of 6.
3. Calls the `run()` method of each `Counter` in turn.
4. Displays a message that `main()` is finished.



2.5

A Non-Threaded Example

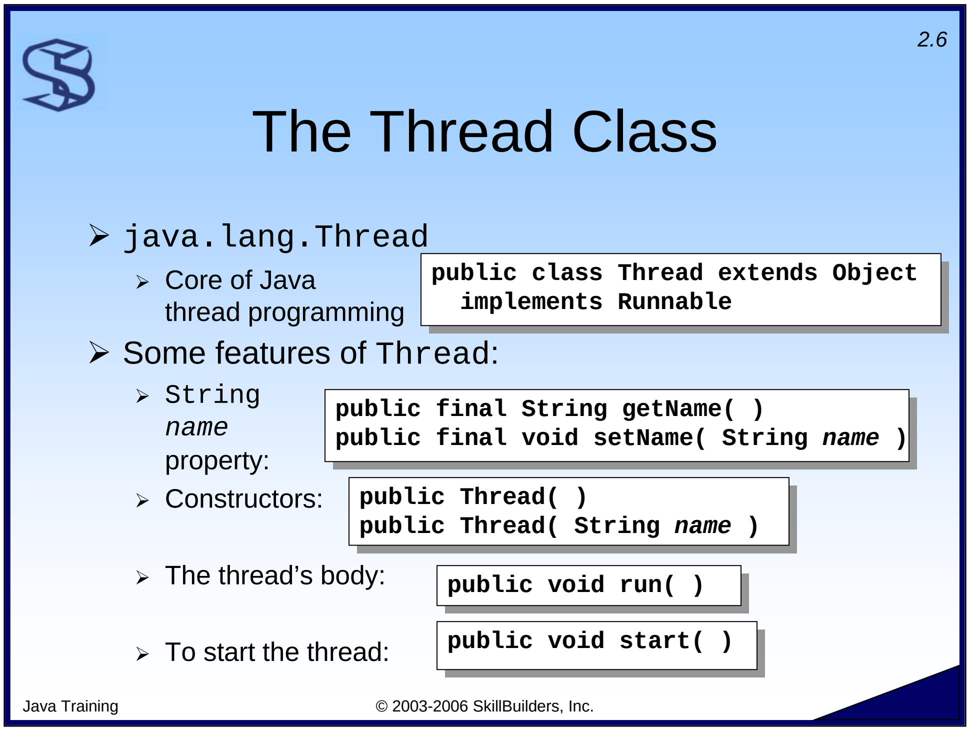
- And the output:

```
C:\JavaAdv\workshops>java CounterUp
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1 is done
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2 is done
main() is done

C:\JavaAdv\workshops>
```

Java Training © 2003-2006 SkillBuilders, Inc.

When we run this application, we see the output shown above. Because all code is executed in the application’s main thread, it is executed completely sequentially:

1. `Counter 1` begins and runs its course completely so we see the odd numbers
2. Then `Counter 2` begins and runs its course completely so we see the even numbers
3. Only after the two counters are finished does `main()` finish.



2.6

The Thread Class

- `java.lang.Thread`
  - Core of Java thread programming

```
public class Thread extends Object
  implements Runnable
```

- Some features of `Thread`:
  - `String` *name* property:

```
public final String getName( )
public final void setName( String name )
```

  - Constructors:

```
public Thread( )
public Thread( String name )
```

  - The thread's body:

```
public void run( )
```

  - To start the thread:

```
public void start( )
```

Java Training © 2003-2006 SkillBuilders, Inc.

Threading in Java centers on the `Thread` class. Because it is so central to Java programming, this class lives in the `java.lang` package. Let's look at some of its features:

The Class Declaration

```
public class Thread extends Object implements Runnable
```

- The class is a direct descendant of `java.lang.Object`.
- `Thread` implements an interface called `Runnable`; we will discuss it below.

`String` name Property

```
public final String getName( )
public final void setName( String name )
```

- Every thread has a name that need not be unique among active threads. If you do not assign a name when creating the `Thread` instance, the VM assigns one for you.
- These two methods are the standard get/set method pair for a `String` name property for the `Thread` object.
- Why name a thread? To better identify it in a debugger.

Continued...

Constructors

`Thread` has several constructors; here we see the default and the one that accepts a name argument.

The Thread's Body

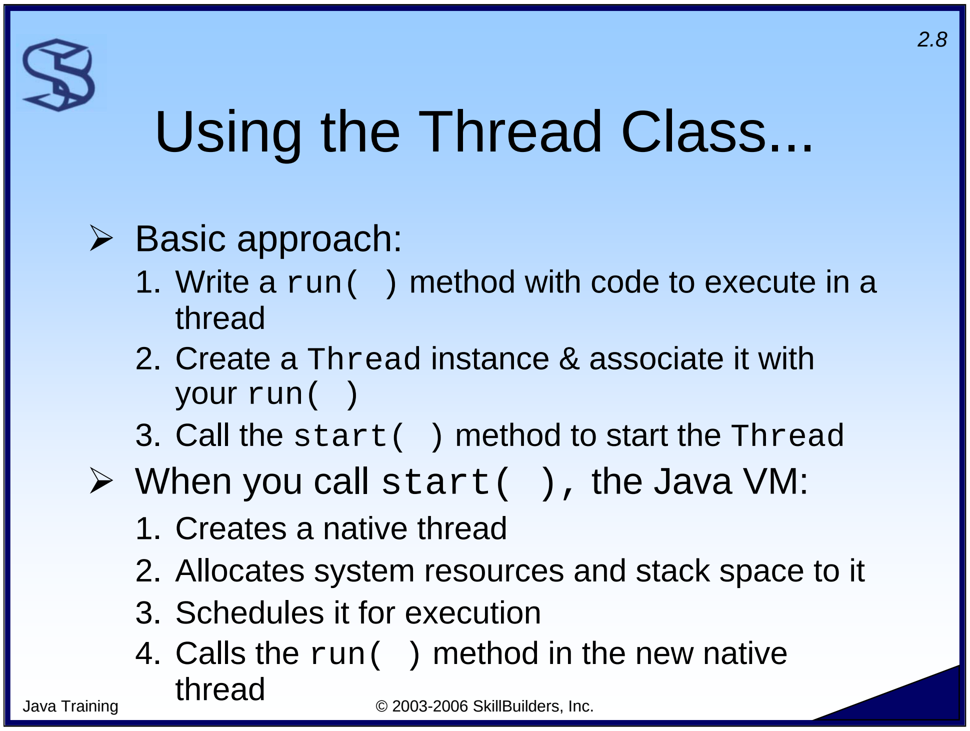
The body of the `Thread` object -- the code executed by the thread -- resides in the `run()` method:

```
public void run( )
```

Starting the Thread

To start execution of a `Thread` object, call its `start()` method:

```
public void start( )
```



2.8

Using the Thread Class...

- Basic approach:
  1. Write a `run( )` method with code to execute in a thread
  2. Create a `Thread` instance & associate it with your `run( )`
  3. Call the `start( )` method to start the `Thread`
- When you call `start( ),` the Java VM:
  1. Creates a native thread
  2. Allocates system resources and stack space to it
  3. Schedules it for execution
  4. Calls the `run( )` method in the new native thread

Java Training

We will study in detail how to use the `Thread` class. But here is the overall approach:

1. Write a `run()` method containing code to execute in a thread. The code in this method becomes the body of the thread.

2. Create a `Thread` instance.

3. Call the `start()` method to start that instance of `Thread`.

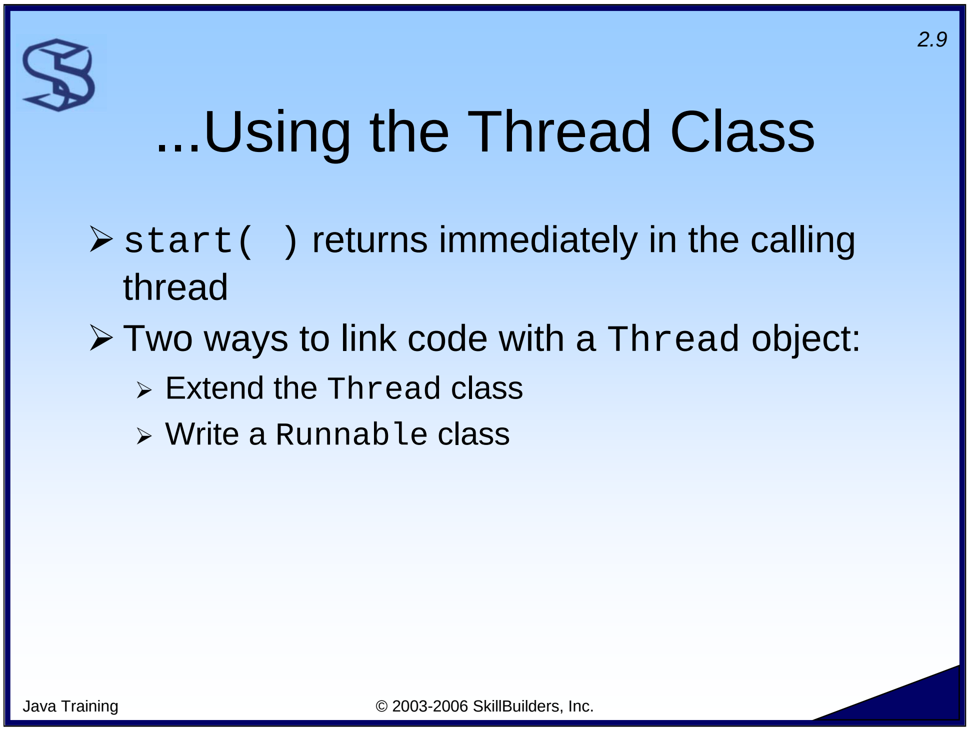
When you call `start(),` the following sequence of events takes place:

1. The Java VM makes a native call to the operating system to create a native thread. (This step is platform specific, but the details are hidden from you by the `Thread` class and the specific Java installation.)

2. The Java VM calls the `Thread` object's `run()` method.

3. The `run()` method you wrote executes in the new native thread.

4. The `start()` returns immediately in the calling thread. That thread continues execution.



2.9

...Using the Thread Class

- `start( )` returns immediately in the calling thread
- Two ways to link code with a `Thread` object:
  - Extend the `Thread` class
  - Write a `Runnable` class

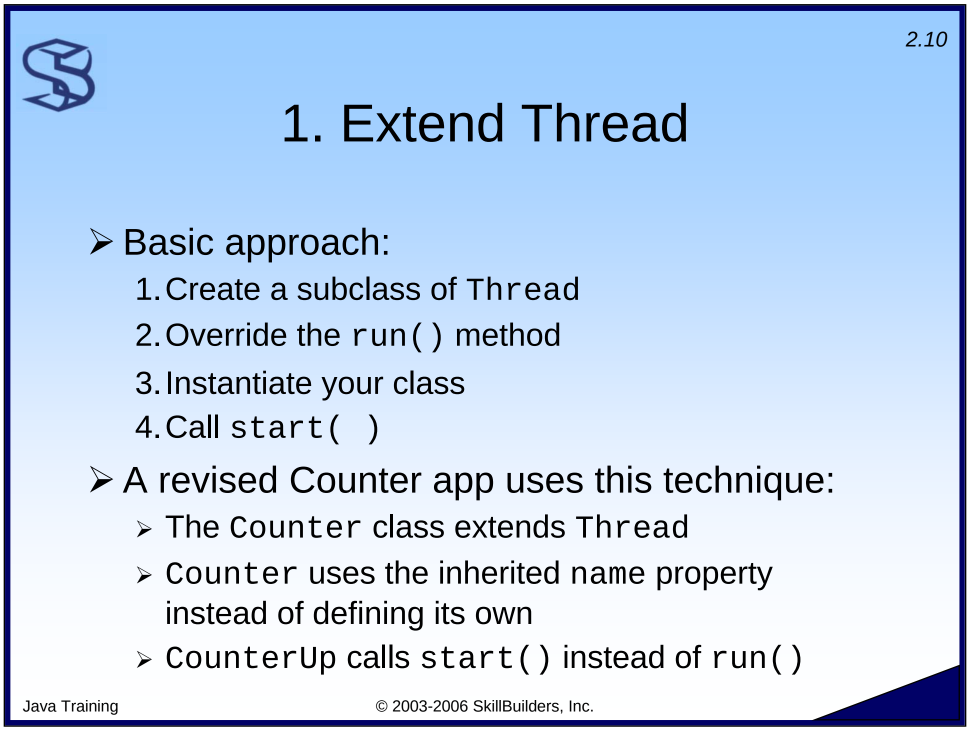
Java Training © 2003-2006 SkillBuilders, Inc.

How do you associate code with a *`Thread`* object? There are 2 main ways:

- Extend the `Thread` class – see page titled "1. Extend Thread".
- Write a `Runnable` class – there are 2 ways to do this

  See the page titled "2a. Write a Runnable class"

  See the page titled "2b. Thread in Runnable"



2.10

1. Extend Thread

- Basic approach:
  1. Create a subclass of `Thread`
  2. Override the `run()` method
  3. Instantiate your class
  4. Call `start( )`
- A revised Counter app uses this technique:
  - The `Counter` class extends `Thread`
  - `Counter` uses the inherited name property instead of defining its own
  - `CounterUp` calls `start()` instead of `run()`

Java Training © 2003-2006 SkillBuilders, Inc.

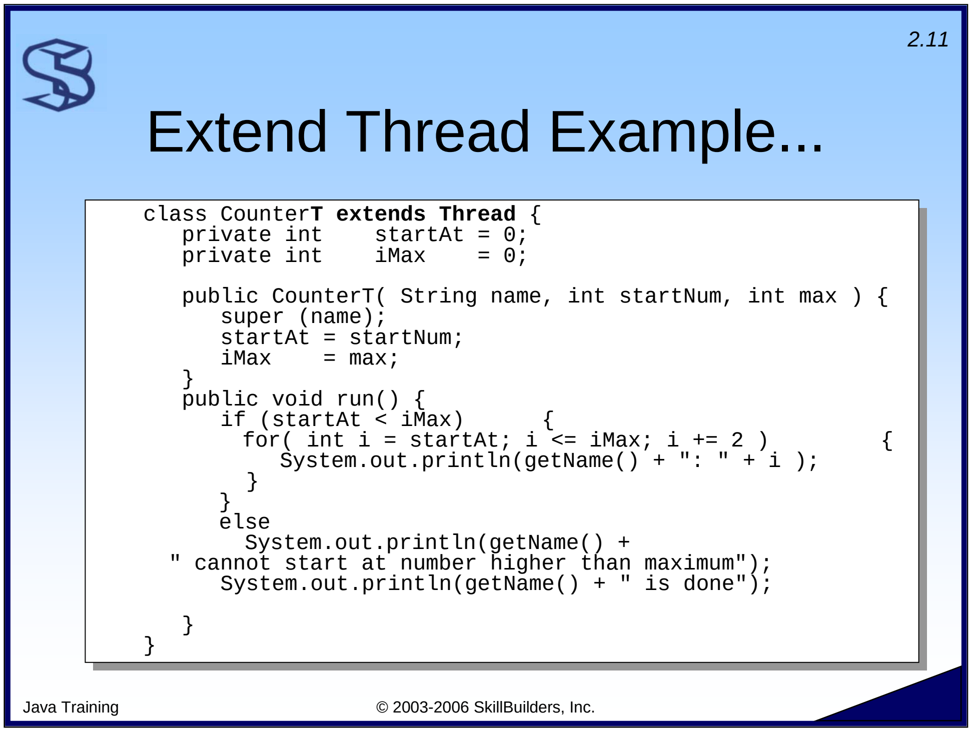
The first technique to tie a piece of code to a `Thread` is to write a class that extends `Thread`. Here is the basic approach:

1. Create a subclass of `Thread`.
2. Override the `run()` method.
3. Instantiate your class.
4. Call `start().`

Here's a revised version of our Counter application using this technique:

- The `Counter` class extends `Thread`.
- `Counter` uses the inherited `name` property instead of defining its own. Its constructor calls `super()` to assign the name to the thread.
- After instantiating two `Counter` objects, `CounterApp` calls their `start()` method instead of `run().`

See the code on the next page...



2.11

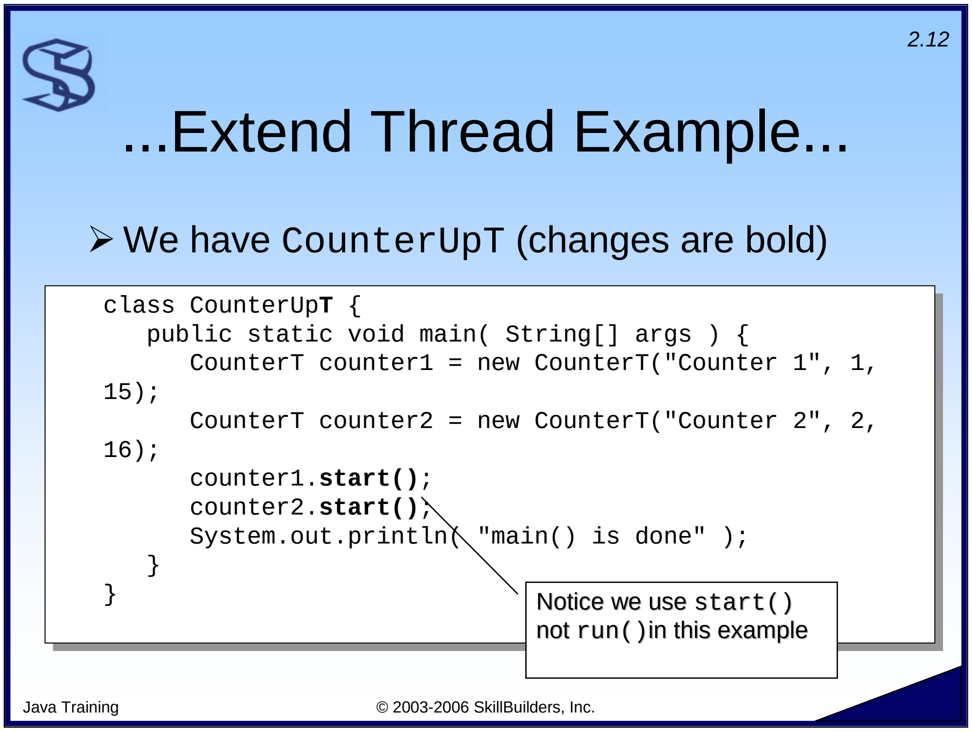
Extend Thread Example...

```
class CounterT extends Thread {
   private int    startAt = 0;
   private int    iMax    = 0;

   public CounterT( String name, int startNum, int max ) {
      super (name);
      startAt = startNum;
      iMax    = max;
   }
   public void run() {
      if (startAt < iMax)      {
        for( int i = startAt; i <= iMax; i += 2 )        {
           System.out.println(getName() + ": " + i );
        }
      }
      else
        System.out.println(getName() +
  " cannot start at number higher than maximum");
      System.out.println(getName() + " is done");

   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

The `CounterT` class has a `T` at the end of the class name just for us to distinguish from the earlier example of `Counter`:

- We have chosen to use `super` to get the name property. We no longer have a name property defined as an instance variable. The name is set in the `super` class `Thread` using the `String` passed into the constructor for this class.
- A maximum value property that is also set via a constructor argument.
- The `run()` method that uses a `for` loop to display a series of messages that includes the counter name, as assigned when the class was constructed, and the current count (**`i`**) number which starts at input `startAt`, is incremented by 2 (to get odd numbers) and continues until the `iMax` value (supplied in the constructor) is reached. We use the `getName()` method of `Thread` to display the thread name.



2.12

...Extend Thread Example...

- We have `CounterUpT` (changes are bold)

```
    class CounterUpT {
       public static void main( String[] args ) {
          CounterT counter1 = new CounterT("Counter 1", 1,
    15);
          CounterT counter2 = new CounterT("Counter 2", 2,
    16);
          counter1.start();
          counter2.start();
          System.out.println( "main() is done" );
       }
    }
```

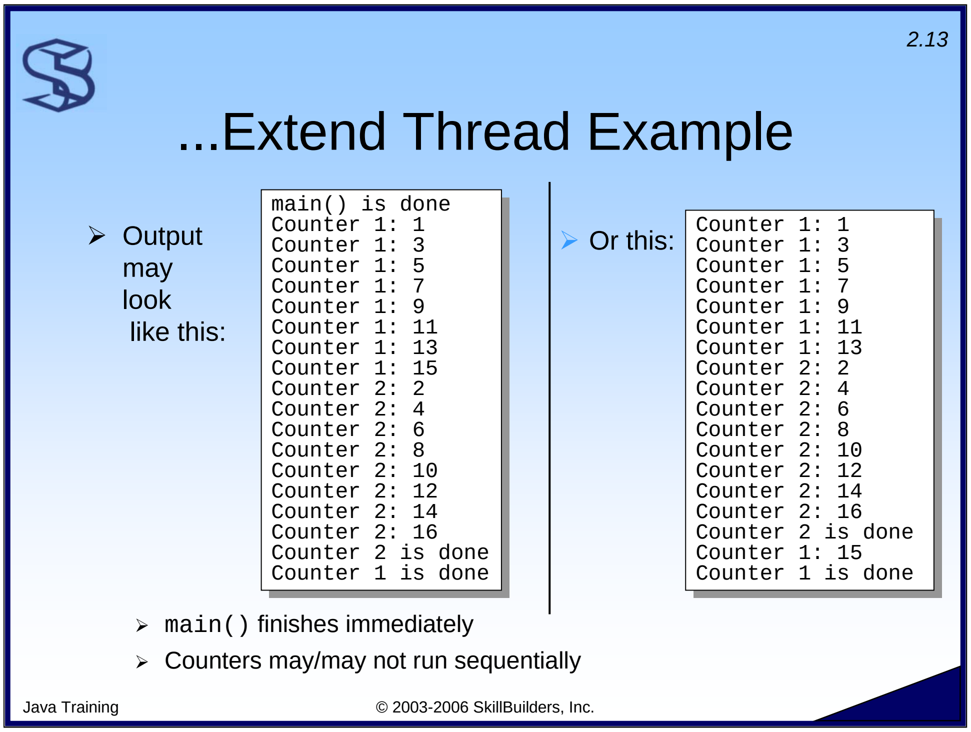
Notice we use `start()` not `run()`in this example

Java Training © 2003-2006 SkillBuilders, Inc.

We added a T to the class name here and made the modifications required to use the `CounterT` class.

Really the only change required in this code was to change the `run()` to `start()`.

Because the `CounterT` class was changed to `extend Thread`, the `start()` will schedule the thread and invoke its `run()` method.



2.13

...Extend Thread Example

- Output may look like this:

```
main() is done
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 1: 15
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1 is done
```

- Or this:

```
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1: 15
Counter 1 is done
```

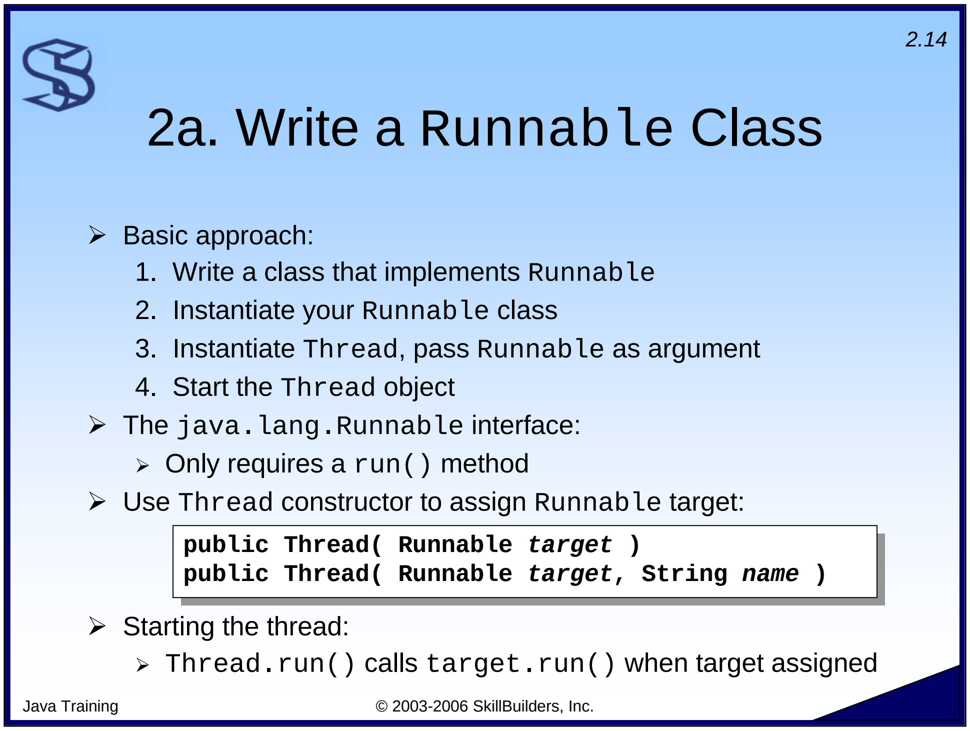
- `main()` finishes immediately
- Counters may/may not run sequentially

Java Training © 2003-2006 SkillBuilders, Inc.

Here is the output from the revised application. Notice that:

- `main()` finishes right away. This is because the `start()` method of the two `Counter` objects returned immediately.
- The two `Counter` objects take may turns displaying output. This is because the CPU schedules (allocates) time between them. (More on thread scheduling later...)

We will see later how to ensure they take turns



2.14

2a. Write a Runnable Class

- Basic approach:
  1. Write a class that implements `Runnable`
  2. Instantiate your `Runnable` class
  3. Instantiate `Thread`, pass `Runnable` as argument
  4. Start the `Thread` object
- The `java.lang.Runnable` interface:
  - Only requires a `run()` method
- Use `Thread` constructor to assign `Runnable` target:

```
public Thread( Runnable target )
public Thread( Runnable target, String name )
```

- Starting the thread:
  - `Thread.run()` calls `target.run()` when target assigned

Java Training © 2003-2006 SkillBuilders, Inc.

In the second technique, you write a `Runnable` class, instantiate it and link the `Runnable` object with a `Thread` object. Here is the basic approach:

1. Write a class that implements the `Runnable` interface.
2. Instantiate your `Runnable` class.
3. Instantiate `Thread`, passing your `Runnable` instance as a constructor argument representing the thread's target.
4. Start the `Thread` object. This triggers the `run()` method

The Runnable Interface

The `Runnable` interface is one of the simplest in the Java library. Here is its complete listing:

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

As you can see, the interface only requires a `run()` method identical to that of the `Thread` class itself.

Continued...

Assigning the Target

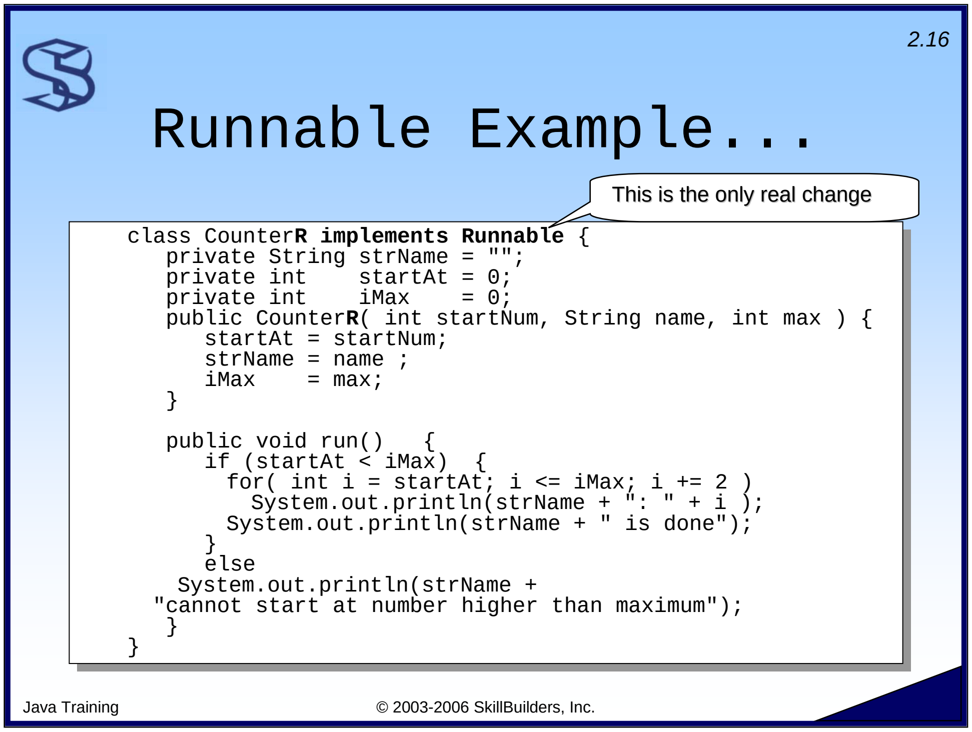
To make your `Runnable` object the target of a `Thread` object, use one of the `Thread` constructors that takes a `Runnable` argument:

```
public Thread( Runnable target )

public Thread( Runnable target, String name )
```

Starting the Thread

As noted, when you start a `Thread` object, the Java VM in turn calls its `run()` method. If you extend `Thread`, this means your redefined `run()` is called. But if a target is assigned, `Thread.run()` calls the target's `run()` instead. The logic for doing so is very simple. Here are the key parts of the `Thread` class:

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



2.16

Runnable Example...

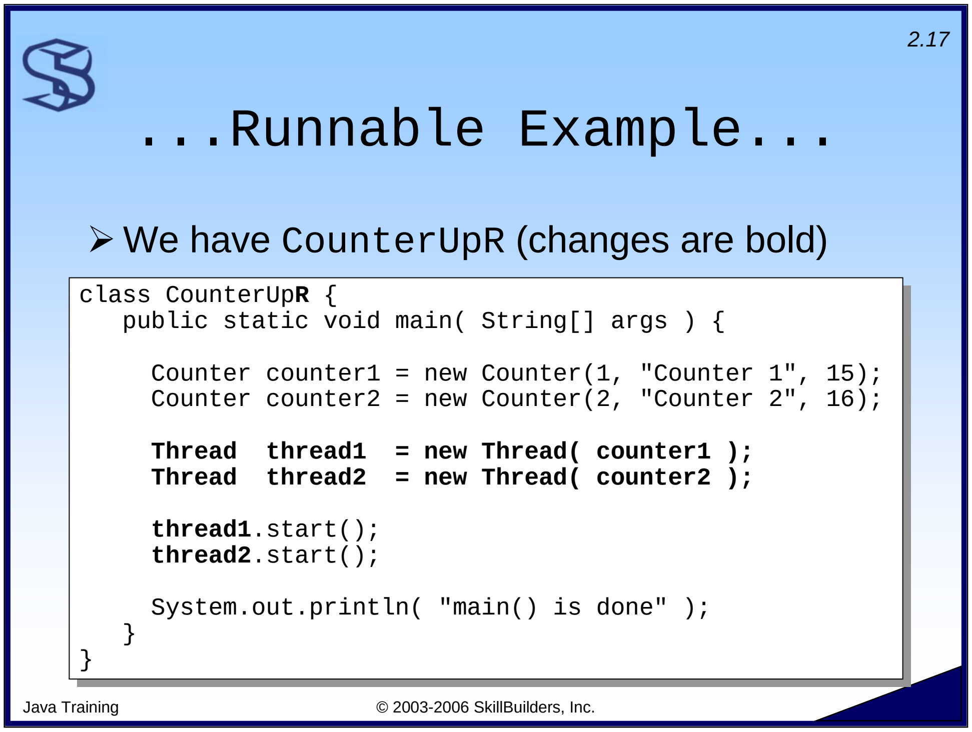
This is the only real change

```
class CounterR implements Runnable {
   private String strName = "";
   private int    startAt = 0;
   private int    iMax    = 0;
   public CounterR( int startNum, String name, int max ) {
      startAt = startNum;
      strName = name ;
      iMax    = max;
   }

   public void run()   {
      if (startAt < iMax)  {
        for( int i = startAt; i <= iMax; i += 2 )
          System.out.println(strName + ": " + i );
        System.out.println(strName + " is done");
      }
      else
    System.out.println(strName +
  "cannot start at number higher than maximum");
   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

The `CounterR` class has an R at the end of the class name just for us to distinguish from the earlier example of `Counter`.

FROM THE ORIGINAL `Counter` class the ONLY THING CHANGED IS THE **`implements Runnable`.**



2.17

...Runnable Example...

- We have `CounterUpR` (changes are bold)

```
class CounterUpR {
   public static void main( String[] args ) {

     Counter counter1 = new Counter(1, "Counter 1", 15);
     Counter counter2 = new Counter(2, "Counter 2", 16);

     Thread  thread1  = new Thread( counter1 );
     Thread  thread2  = new Thread( counter2 );

     thread1.start();
     thread2.start();

     System.out.println( "main() is done" );
   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

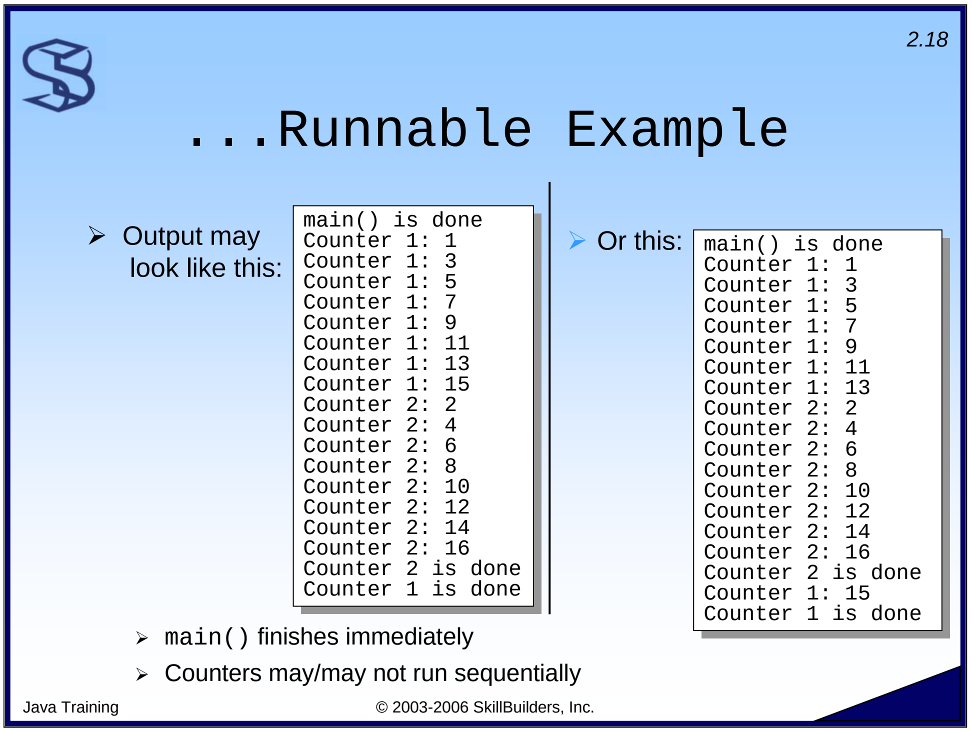
We added an **R** to the class name here and made the modifications required to use the new `CounterR` class.

Here the changes are more distinctive. While there was almost no impact in the counter class (just the `implements Runnable`) , here we see the instantiation of two thread classes and the `start()` of each of those threads.

The `Thread thread1 = new Thread(counter 1);` creates an instance of the `Thread` class called `thread1`. We pass in as a `runnable` target the first (Counter1) `CounterR` instance. The process is repeated for `thread2`, only its argument is `CounterR using even numbers`.

The `thread1.start()` will cause the thread instance to be scheduled and started. Since its target is `CounterR`, the `run()` method of that class is the one that will be executed. Repeated for `thread2`, it invokes `CounterR.run() with the even number start value.`

We do this because our classes may need to extend a more application specific class and cannot `extend Thread` as we saw in the previous example (`CounterUpT`). Remember Java does not permit multiple inheritance (extending more than one class) so being able to `implement Runnable` provides a way to have your application inheritance and use multi-threading.



2.18

...Runnable Example

- Output may look like this:

```
main() is done
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 1: 15
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1 is done
```

- Or this:

```
main() is done
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1: 15
Counter 1 is done
```

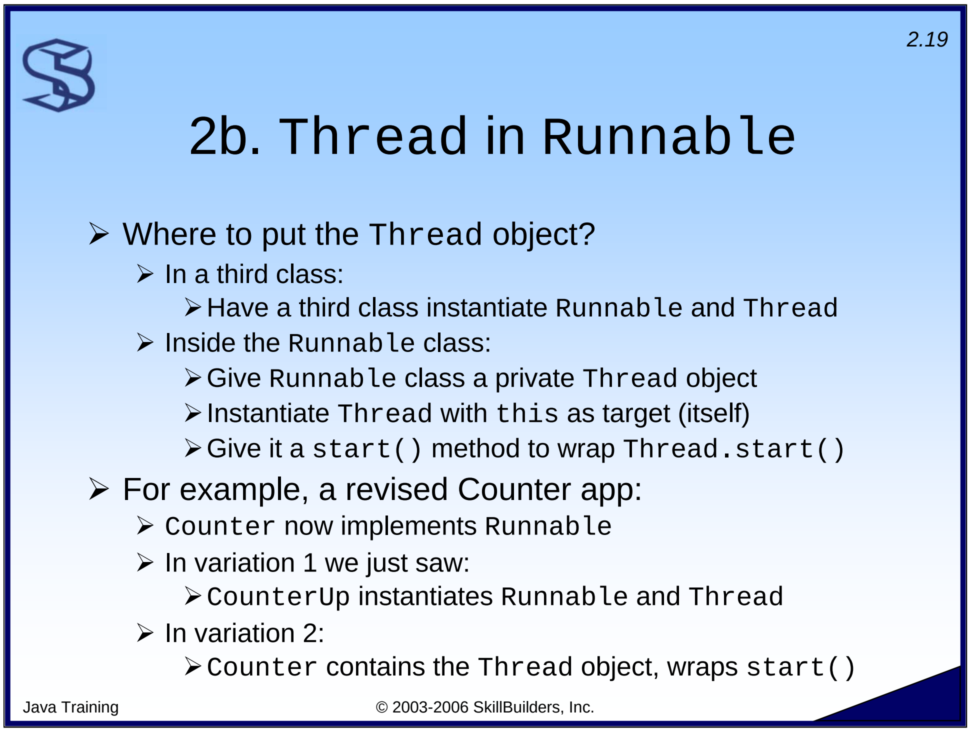
  - `main()` finishes immediately
  - Counters may/may not run sequentially

Java Training © 2003-2006 SkillBuilders, Inc.

Here is the output from the revised application. Notice that:

- `main()` finishes right away. This is because the `start()` method of the two `Counter` objects returned immediately.

- The two `Counter` objects take may turns displaying output. This is because the CPU schedules (allocates) time between them. (More on thread scheduling later...)



2.19

2b. `Thread` in `Runnable`

- Where to put the `Thread` object?
  - In a third class:
    - Have a third class instantiate `Runnable` and `Thread`
  - Inside the `Runnable` class:
    - Give `Runnable` class a private `Thread` object
    - Instantiate `Thread` with `this` as target (itself)
    - Give it a `start()` method to wrap `Thread.start()`
- For example, a revised Counter app:
  - `Counter` now implements `Runnable`
  - In variation 1 we just saw:
    - `CounterUp` instantiates `Runnable` and `Thread`
  - In variation 2:
    - `Counter` contains the `Thread` object, wraps `start()`

Java Training © 2003-2006 SkillBuilders, Inc.

When using the second technique, you face a choice as to where to put the `Thread` object and assign the target to it. You can do this in one of two places:

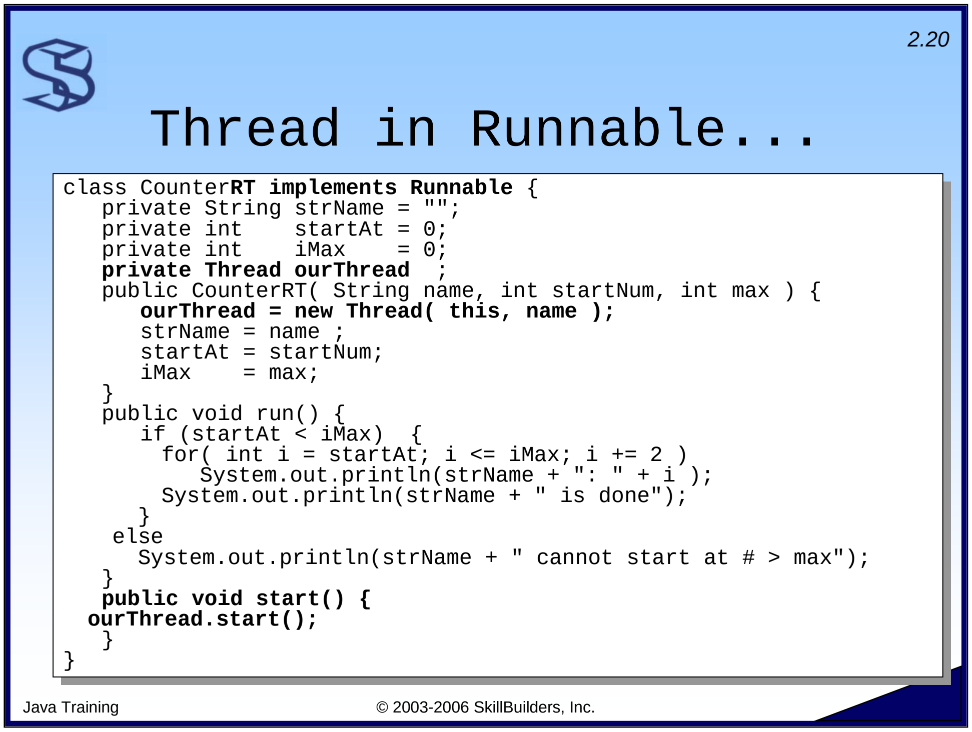
- Inside a third class
- Inside the `Runnable` class itself.

In the former case, simply instantiate your `Runnable` and a `Thread` wherever you wish.

In the latter case:

- Give your `Runnable` class a private `Thread` object
- Instantiate `Thread`, passing `this` as the target.
- You must also give your `Runnable` class a `start()` method to call (wrap) that of your `Thread` object; otherwise the user will have no way to start the thread.

We illustrated the first use of `runnable` before this, now we will show the more complex variation.



2.20

Thread in Runnable...

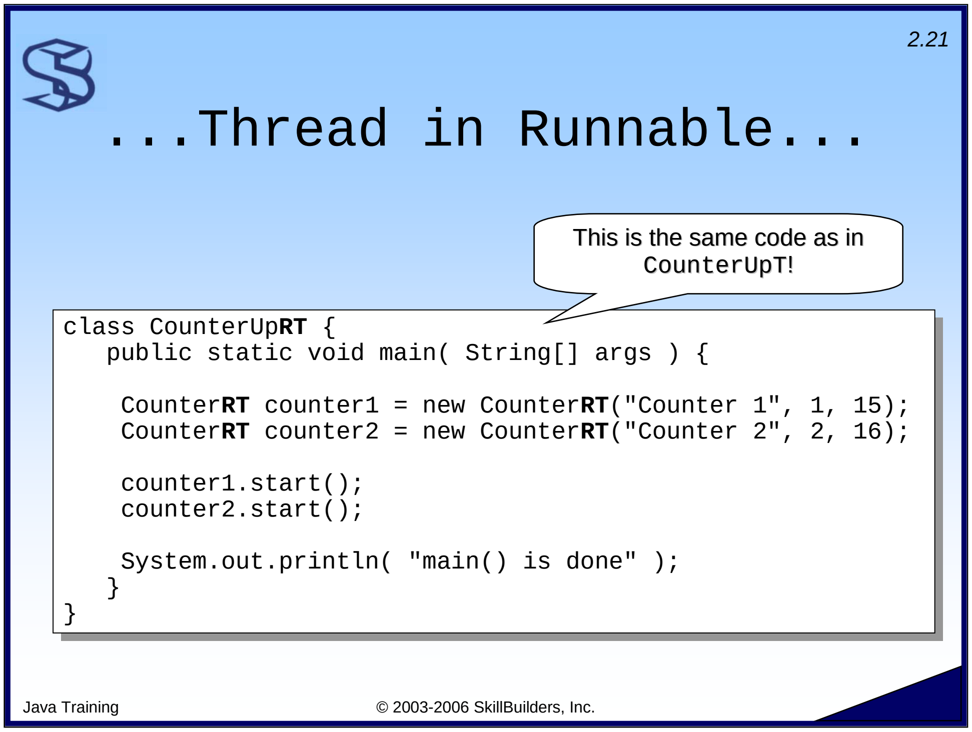
```
class CounterRT implements Runnable {
   private String strName = "";
   private int    startAt = 0;
   private int    iMax    = 0;
   private Thread ourThread  ;
   public CounterRT( String name, int startNum, int max ) {
      ourThread = new Thread( this, name );
      strName = name ;
      startAt = startNum;
      iMax    = max;
   }
   public void run() {
      if (startAt < iMax)  {
        for( int i = startAt; i <= iMax; i += 2 )
           System.out.println(strName + ": " + i );
        System.out.println(strName + " is done");
      }
    else
      System.out.println(strName + " cannot start at # > max");
   }
   public void start() {
  ourThread.start();
   }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

The `CounterRT` class has RT at the end of the class name just for us to distinguish from the earlier example of `Counter`;

We have defined a `private` instance variable that is a `Thread` called `thread`. In the constructor for this class, the `thread` variable is set to an instance of this class and given the name passed in to the constructor as an argument.

We have added a start method in this class to provide a means for users of the class to invoke `thread start().` Within our start we issue the actual `start().` This is called a wrapper class. It will invoke the `run()` method of this class which is the thread instance.



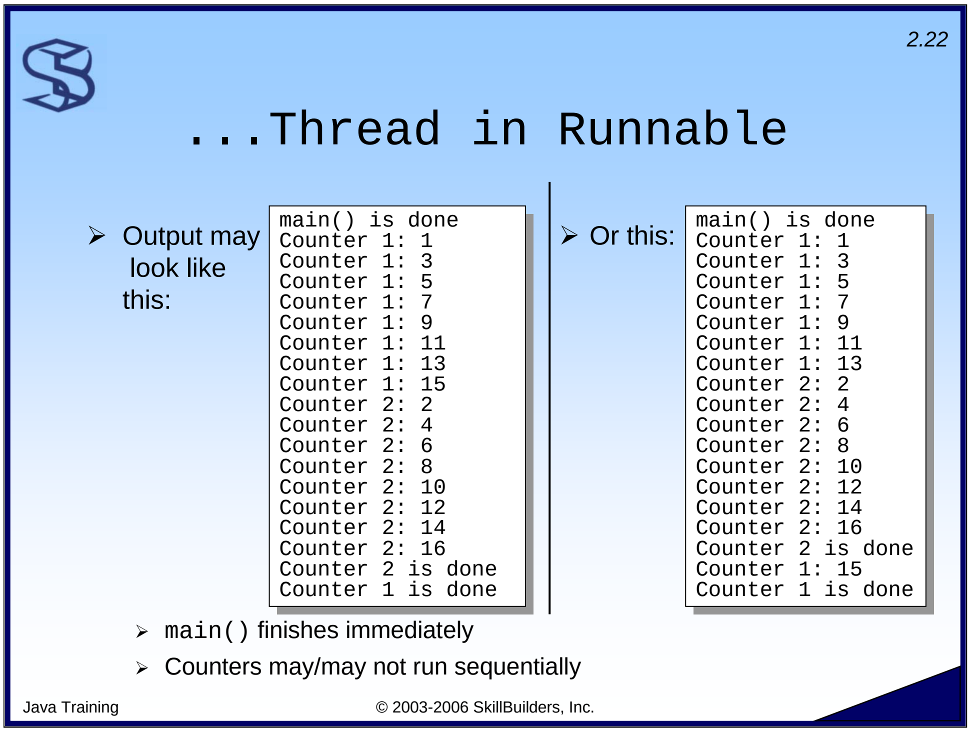
...Thread in Runnable...

This is the same code as in `CounterUpT`!

```
class CounterUpRT {
   public static void main( String[] args ) {

    CounterRT counter1 = new CounterRT("Counter 1", 1, 15);
    CounterRT counter2 = new CounterRT("Counter 2", 2, 16);

    counter1.start();
    counter2.start();

    System.out.println( "main() is done" );
   }
}
```

The `CounterUpRT` class has `RT` at the end of the class name just for us to distinguish from the earlier example of `CounterUp`.

This could have been the same exact code as the `CounterUpT` class we showed earlier. When the odd and even counter classes extended `Thread`, we just said `start().` Now we have removed the `extends Thread` but by implementing `Runnable` and creating a `Thread` instance with a `start()` method in the class – we can still invoke it as if it were a simple thread class.



2.22

...Thread in Runnable

- Output may look like this:

```
main() is done
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 1: 15
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1 is done
```

- Or this:

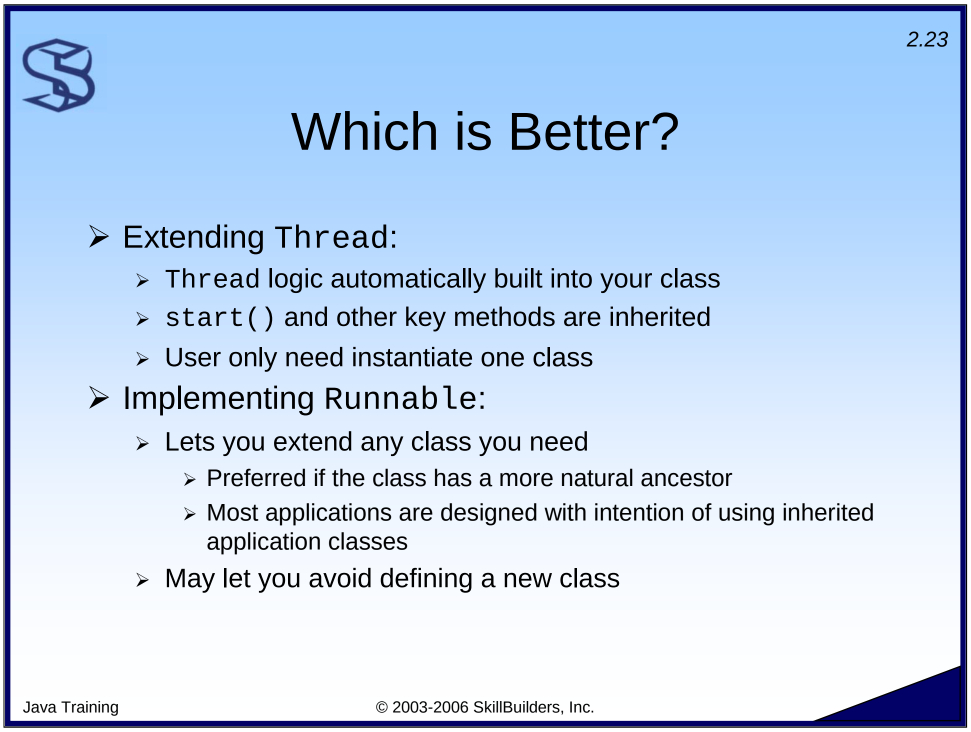
```
main() is done
Counter 1: 1
Counter 1: 3
Counter 1: 5
Counter 1: 7
Counter 1: 9
Counter 1: 11
Counter 1: 13
Counter 2: 2
Counter 2: 4
Counter 2: 6
Counter 2: 8
Counter 2: 10
Counter 2: 12
Counter 2: 14
Counter 2: 16
Counter 2 is done
Counter 1: 15
Counter 1 is done
```

  - `main()` finishes immediately
  - Counters may/may not run sequentially

Java Training © 2003-2006 SkillBuilders, Inc.

Here is the output from the revised application. Notice that:

- `main()` finishes right away. This is because the `start()` method of the two `Counter` objects returned immediately.
- The two `Counter` objects take may turns displaying output. This is because the CPU schedules (allocates) time between them. (More on thread scheduling later...).



2.23

Which is Better?

- Extending `Thread`:
  - `Thread` logic automatically built into your class
  - `start()` and other key methods are inherited
  - User only need instantiate one class
- Implementing `Runnable`:
  - Lets you extend any class you need
    - Preferred if the class has a more natural ancestor
    - Most applications are designed with intention of using inherited application classes
  - May let you avoid defining a new class

Java Training © 2003-2006 SkillBuilders, Inc.

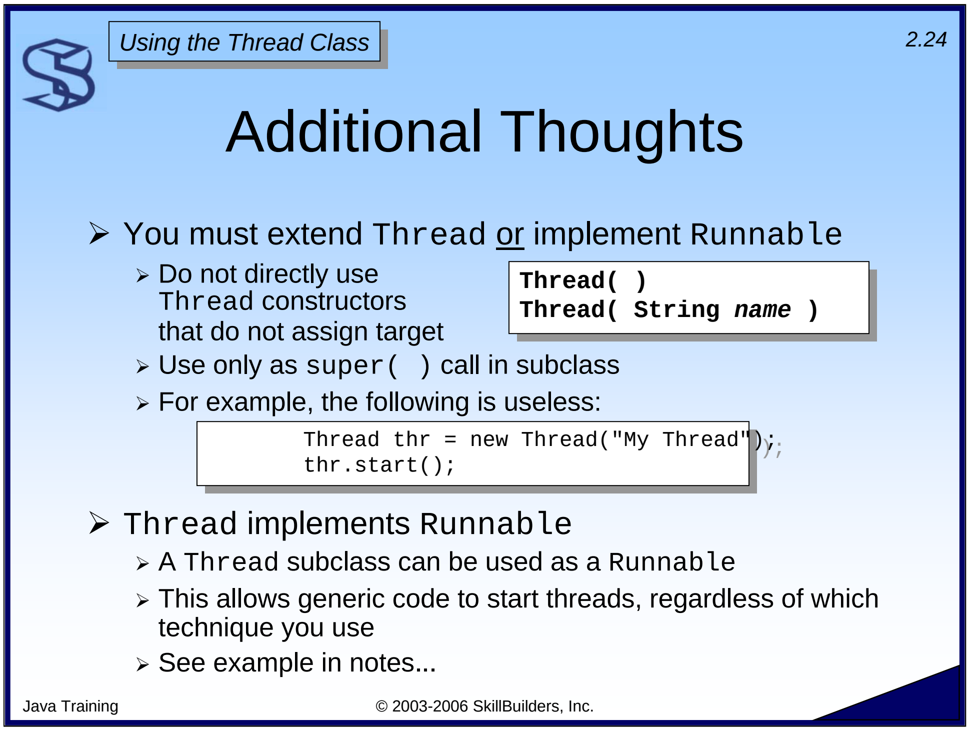
So, given the two techniques for using a `Thread` object, which is better? As always when you have a choice, there are advantages to each.

Extending Thread

When you extend the `Thread` class, the threading logic is automatically inherited by and encapsulated in your class. You inherit a `start()` method and all the other control methods we will discuss. What's more, the user of your class need instantiate only it, not `Thread`.

Implementing `Runnable`

Implementing `Runnable`, on the other hand, frees the extension slot to let you extend any class you need.



Using the Thread Class

2.24

Additional Thoughts

- You must extend `Thread` <u>or</u> implement `Runnable`
  - Do not directly use `Thread` constructors that do not assign target
  - Use only as `super( )` call in subclass
  - For example, the following is useless:

```
Thread( )
Thread( String name )
```

```
Thread thr = new Thread("My Thread");
thr.start();
```

- `Thread` implements `Runnable`
  - A `Thread` subclass can be used as a `Runnable`
  - This allows generic code to start threads, regardless of which technique you use
  - See example in notes...

Java Training © 2003-2006 SkillBuilders, Inc.

Use One or the Other Technique

A footnote to this discussion: You must use one of these techniques to link your code to a `Thread` object; otherwise the `run( )` method is empty and the thread does nothing. Even though the `Thread` class provides constructors that do not take target arguments (such as the no-args constructor), they are meant to be used not directly, but only as explicit or implied `super( )` calls in a subclass constructor.

For example, the following code will compile and run just fine. But it is useless because the `run( )` method invoked will do nothing:

```
Thread thr = new Thread(“My Thread”);
thr.start();
```

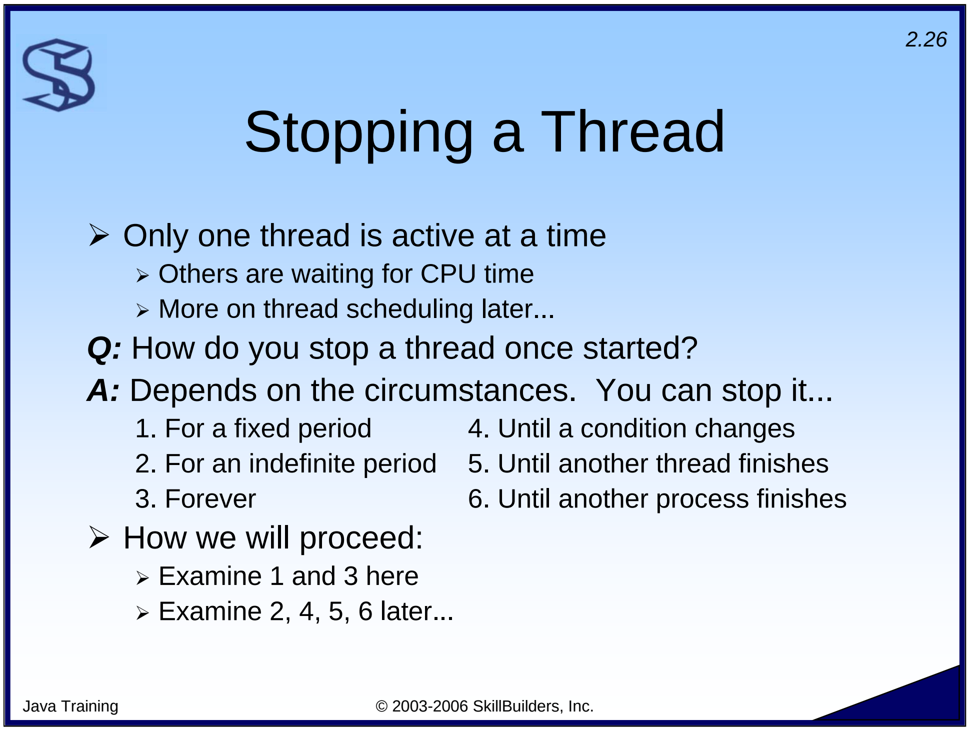
The `Thread` class `implements Runnable`

Since the `Thread` class implements the `Runnable` interface, an instance of a subclass of `Thread` is a `Runnable` and can be the target of another `Thread` object. This means you can write a generic routine to start threads; the routine can accept a `Runnable` argument, and your application can pass in an object whose class uses either technique.

See the example on the next page...

In this example we have a custom class called `ThreadMaker` that takes a *runnable* object, assigns it to a thread, then starts and returns the thread. The `CounterApp` class uses it to start `Counter` threads, even though `Counter extends Thread`.

```
// Custom class to create and start Threads.
public class ThreadMaker {
   public static Thread makeThread( Runnable target ) {
      Thread thr;
      if( target instanceof Thread )
         thr = (Thread) target;
      else
         thr = new Thread( target );
      thr.start();
      return thr;
   }
   // Rest of class...
}

class Counter extends Thread {
   // Body of Counter class as shown before...
}

class CounterApp {
   public static void main( String[] args ) {
      Counter counter1 = new Counter( "Counter 1", 6 );
      Counter counter2 = new Counter( "Counter 2", 6 );
      Thread t1 = ThreadMaker.makeThread( counter1 );
      Thread t2 = ThreadMaker.makeThread( counter2 );
      System.out.println( "main() is done" );
   }
}
```



2.26

Stopping a Thread

- Only one thread is active at a time
  - Others are waiting for CPU time
  - More on thread scheduling later...

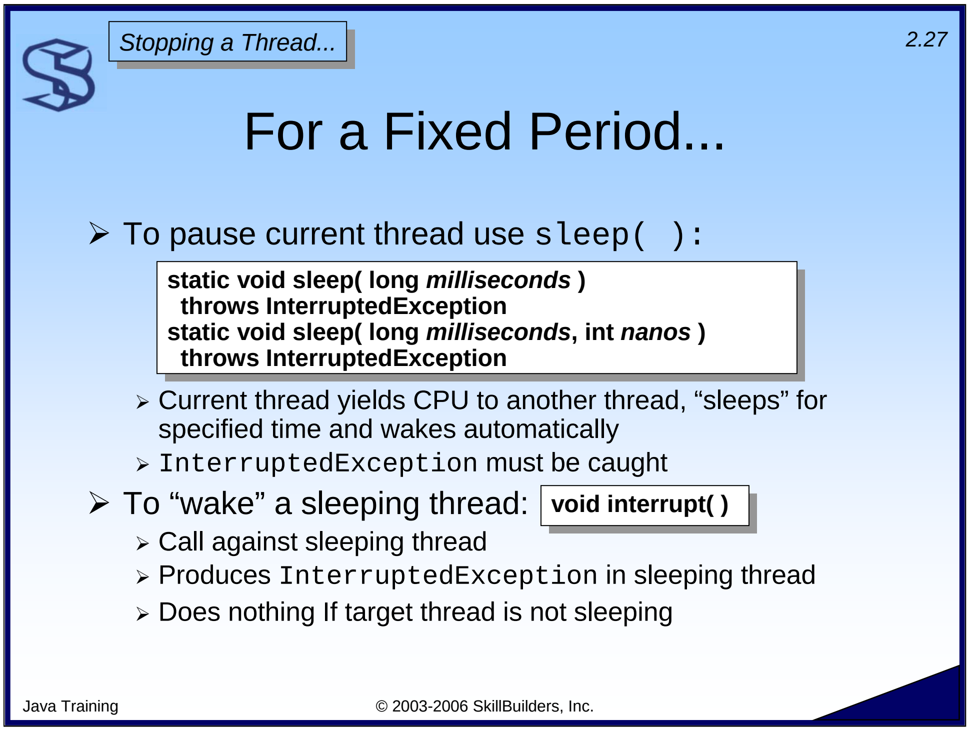
Q: How do you stop a thread once started?

A: Depends on the circumstances. You can stop it...

1. For a fixed period
2. For an indefinite period
3. Forever
4. Until a condition changes
5. Until another thread finishes
6. Until another process finishes

- How we will proceed:
  - Examine 1 and 3 here
  - Examine 2, 4, 5, 6 later...

Java Training © 2003-2006 SkillBuilders, Inc.



Stopping a Thread...

2.27

For a Fixed Period...

- To pause current thread use `sleep( )`:

```
static void sleep( long milliseconds )
  throws InterruptedException
static void sleep( long milliseconds, int nanos )
  throws InterruptedException
```

  - Current thread yields CPU to another thread, “sleeps” for specified time and wakes automatically
  - `InterruptedException` must be caught
- To “wake” a sleeping thread: **void interrupt( )**
  - Call against sleeping thread
  - Produces `InterruptedException` in sleeping thread
  - Does nothing If target thread is not sleeping

Java Training © 2003-2006 SkillBuilders, Inc.

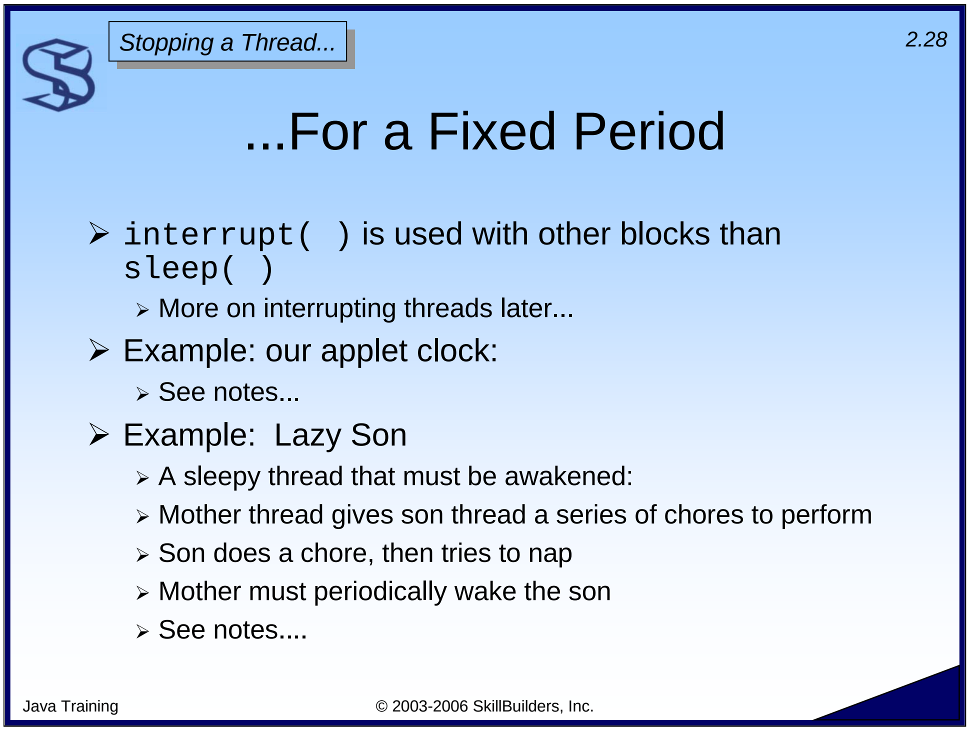
Often you need a thread to simply pause for a limited, fixed period of time. For example, our applet clock waits 1 second before updating the time display. In Java jargon, we say the current (active) thread is sleeping.

The `sleep( )` method of the `Thread` class puts the current thread to sleep for the specified number of milliseconds. Since at any moment there is only one current thread, this method is `static`. When the thread goes to sleep, it yields the CPU to another thread. If uninterrupted, the sleeping thread will awaken after the specified time.

But another thread can awaken the sleeping thread by interrupting it. One does this by calling the instance method `interrupt( )` of the sleeping thread.

The sleeping thread recognizes the interruption as an `InterruptedException` thrown at the point where `sleep( )` was called. This exception is checked; that is, it must be trapped, which is why you generally see `sleep( )` called inside a `try` block.

Note that `interrupt( )` is used with threads that are blocked for other reasons besides `sleep( ).` We’ll discuss these and more details of thread interruption later on...



Stopping a Thread...

2.28

...For a Fixed Period

- `interrupt( )` is used with other blocks than `sleep( )`
  - More on interrupting threads later...
- Example: our applet clock:
  - See notes...
- Example: Lazy Son
  - A sleepy thread that must be awakened:
  - Mother thread gives son thread a series of chores to perform
  - Son does a chore, then tries to nap
  - Mother must periodically wake the son
  - See notes....

Java Training

Here are a couple of examples of sleeping threads:

Applet Clock

In the first example we show the full code for the applet clock discussed earlier. The clock loop displays the current time, then sleeps for 1 second (1000 milliseconds). The `InterruptedException` must be caught, but we need not do anything about it.

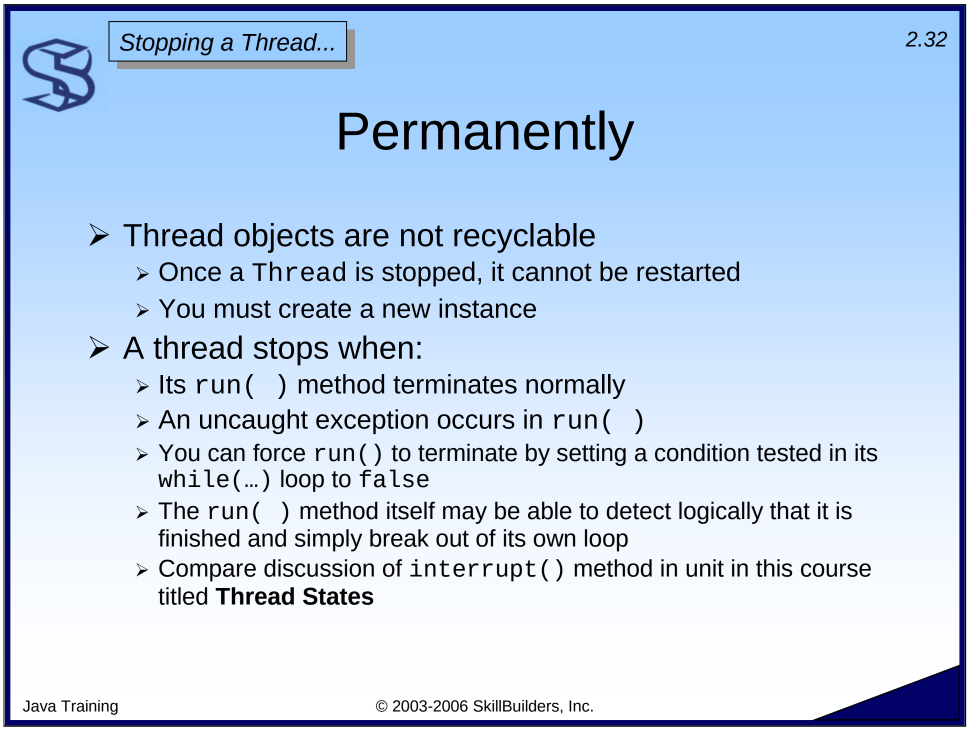
```
public void run() {
   // Continuously display the current time
   while(true) {
      showStatus((new Date()).toString());
      try{
         Thread.sleep(1000);  // sleep 1 second
      }
      catch( InterruptedException e ){// Do nothing}
   }
}
```

Another example can be found on the next page….

Lazy Son

In a second example a “mother” thread gives her “lazy son” thread a series of chores to do. The son does a chore, then tries to nap for 10 minutes. But in a separate thread created by the mother, she periodically must interrupt the son’s nap to remind him to continue his chores.

```java
// The lazy son class
public class LazySon extends Child implements Runnable {
   private static final long NAP_TIME = 10*60*1000; // 10 min
   private Chore[] _chores;

   public void assignChores( Chore[] chores ) {
      _chores = chores
   }

   private void doChore( Chore chore ) {
      // Son does one chore here...
   }
   public void run() {
      for( int i = 0; i < _chores.length; i++ ) {
         doChore( _chores[i] );   // Do 1 chore
         // Now try to take a nap...
         try {
            Thread.sleep(NAP_TIME);
         }
         catch( InterruptedException ie ) {
            System.out.println("Alright! I'm up!!!");
         }
      }
      System.out.println("Chores are done!!!");
   }
   // Rest of Son class...
}

// Continues...
// The nagging mother class
public class Mother {
   private Child _OnlySon = new LazySon();
   private Thread _thrSonChores;

   private void assignChoresToOnlySon(Chore[] chores) {
      // Get son going on chores:
      _OnlySon.assignChores(chores);
      _thrSonChores = new Thread(_OnlySon);
      _thrSonChores.start();

      // Create/start thread to nag son:
      (new NagLazySon()).start();
   }

   private class NagLazySon extends Thread {
      private int _iChoreCount;
      private final long NAP_TIME = 10*60*1000; // 10 min
      public void run()
         for( int i = 0; i < _iChoreCount; i ++ )
            System.out.println("Get back to work!");
            _thrSonChores.interrupt();
            try {
               Thread.sleep( NAP_TIME/2 );
            }
            catch( InterruptedException ie ) { // Do nothing }
         }
      }
   }
   // Rest of Mother class...
}
```



Stopping a Thread...

2.32

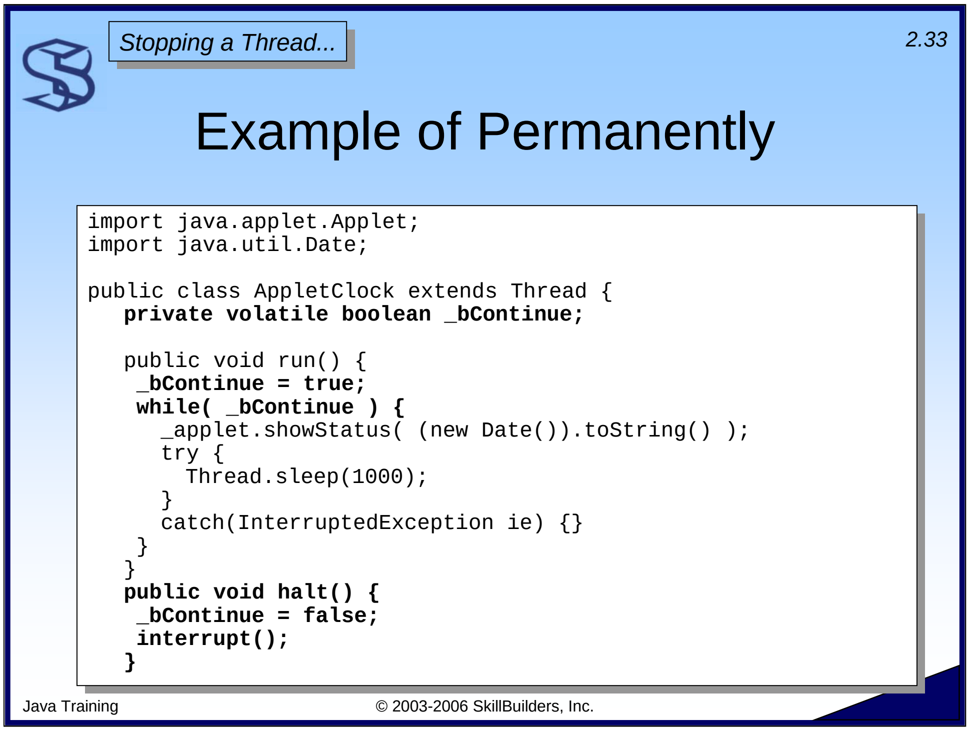
Permanently

- Thread objects are not recyclable
  - Once a `Thread` is stopped, it cannot be restarted
  - You must create a new instance
- A thread stops when:
  - Its `run( )` method terminates normally
  - An uncaught exception occurs in `run( )`
  - You can force `run()` to terminate by setting a condition tested in its `while(…)` loop to `false`
  - The `run( )` method itself may be able to detect logically that it is finished and simply break out of its own loop
  - Compare discussion of `interrupt()` method in unit in this course titled **Thread States**

Java Training © 2003-2006 SkillBuilders, Inc.

A thread will stop for good when its `run( )` method comes to an end, either by a normal termination or via an uncaught exception. You can force a thread to stop by setting the condition in its looping test to false.

Please turn to the next page for an example.



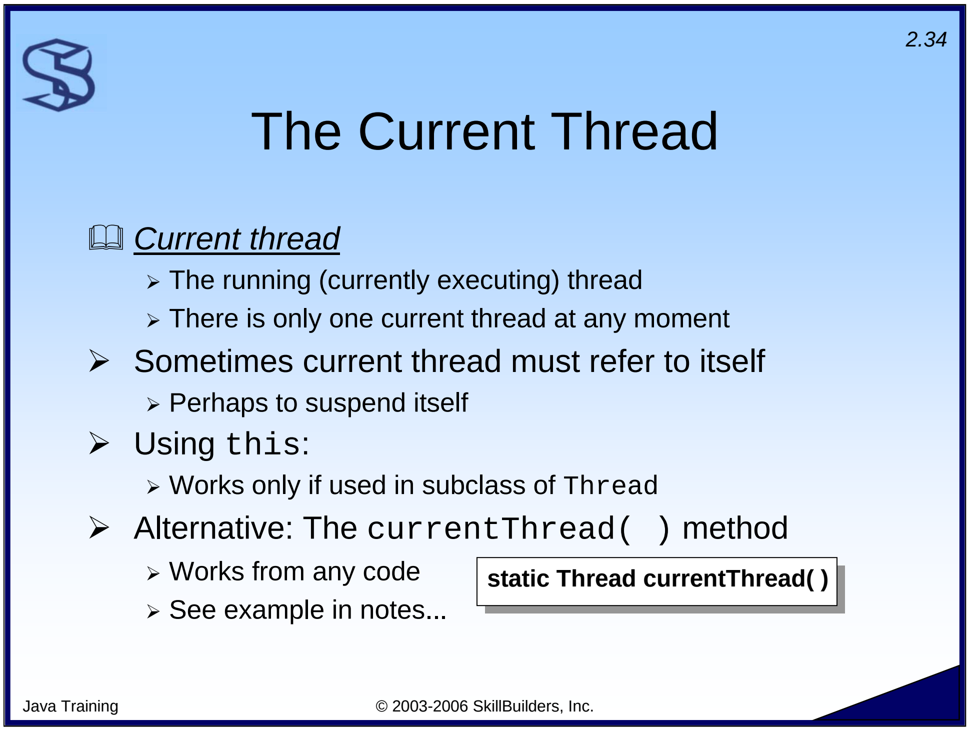
Stopping a Thread...

2.33

Example of Permanently

```
import java.applet.Applet;
import java.util.Date;

public class AppletClock extends Thread {
   private volatile boolean _bContinue;

   public void run() {
    _bContinue = true;
    while( _bContinue ) {
      _applet.showStatus( (new Date()).toString() );
      try {
        Thread.sleep(1000);
      }
      catch(InterruptedException ie) {}
    }
   }
   public void halt() {
    _bContinue = false;
    interrupt();
   }
```

Java Training © 2003-2006 SkillBuilders, Inc.



2.34

The Current Thread

- *<u>Current thread</u>*
  - The running (currently executing) thread
  - There is only one current thread at any moment
- Sometimes current thread must refer to itself
  - Perhaps to suspend itself
- Using `this`:
  - Works only if used in subclass of `Thread`
- Alternative: The `currentThread( )` method
  - Works from any code
  - See example in notes...

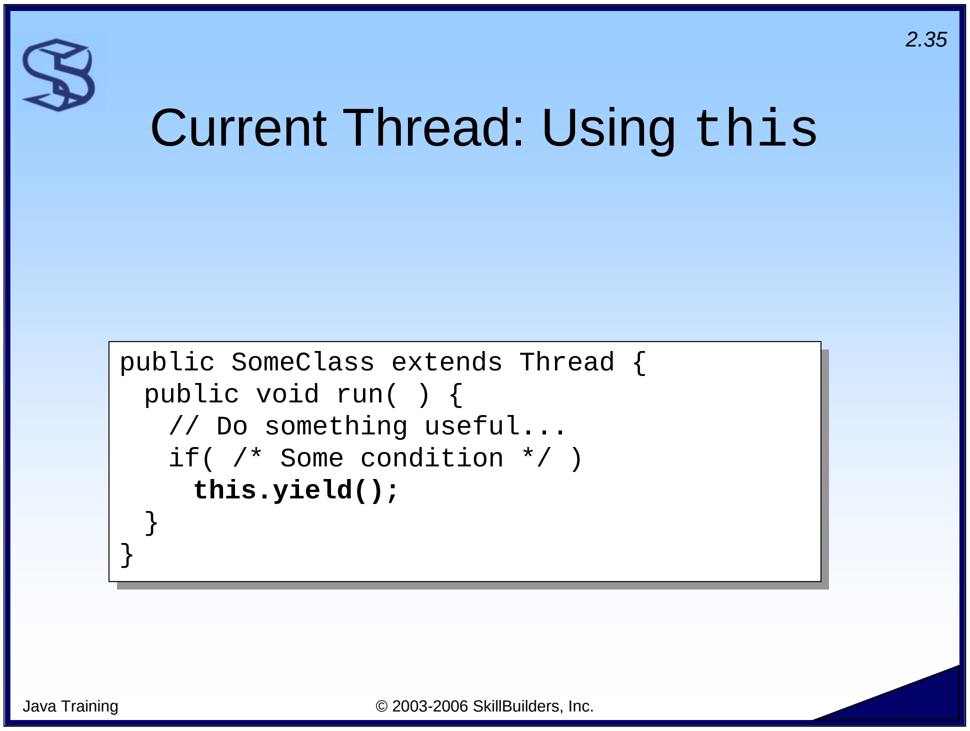
static Thread currentThread( )

Java Training © 2003-2006 SkillBuilders, Inc.

The current thread is the one currently executing. By definition, there is only one current thread at any moment, since we assume the CPU can only handle one at a time..

You will find cases where code executing in the current thread must refer to the thread itself. For example, perhaps the current thread wants to yield the CPU.

Please see the following pages for examples.



2.35

Current Thread: Using `this`

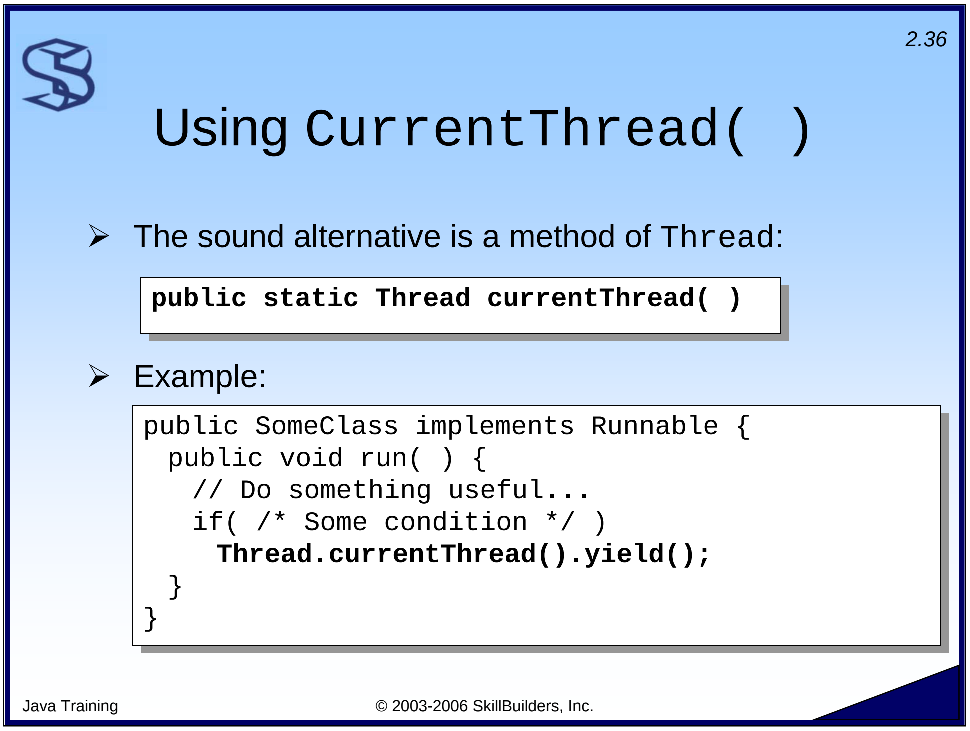
```
public SomeClass extends Thread {
  public void run( ) {
    // Do something useful...
    if( /* Some condition */ )
      this.yield();
  }
}
```

Java Training

© 2003-2006 SkillBuilders, Inc.

The generic self-reference `this` comes to mind, as in the example above.

But this only works if used in a subclass of `Thread`. What if `run( )` is contained in a custom *runnable* class that does not extend `Thread`? Or what if `run( )` calls code in a separate class that also does not extend `Thread`?



2.36

Using `CurrentThread( )`

- The sound alternative is a method of `Thread`:

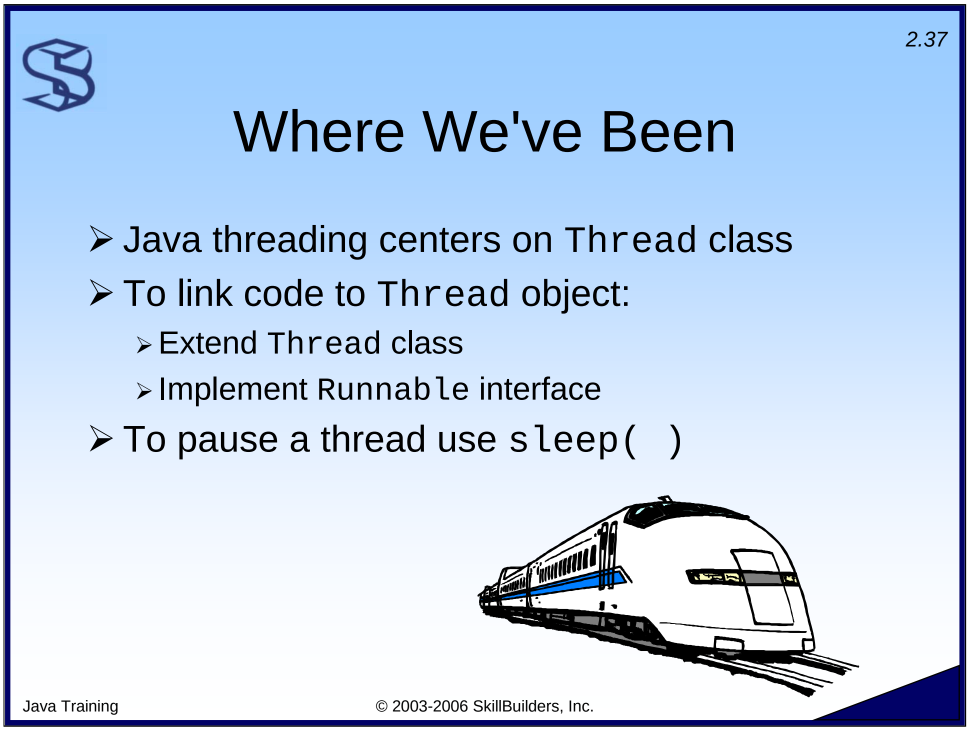
```
public static Thread currentThread( )
```

- Example:

```
public SomeClass implements Runnable {
  public void run( ) {
    // Do something useful...
    if( /* Some condition */ )
      Thread.currentThread().yield();
  }
}
```

Java Training © 2003-2006 SkillBuilders, Inc.

This call will return a reference to the current thread no matter where it is called from. An example is shown above.



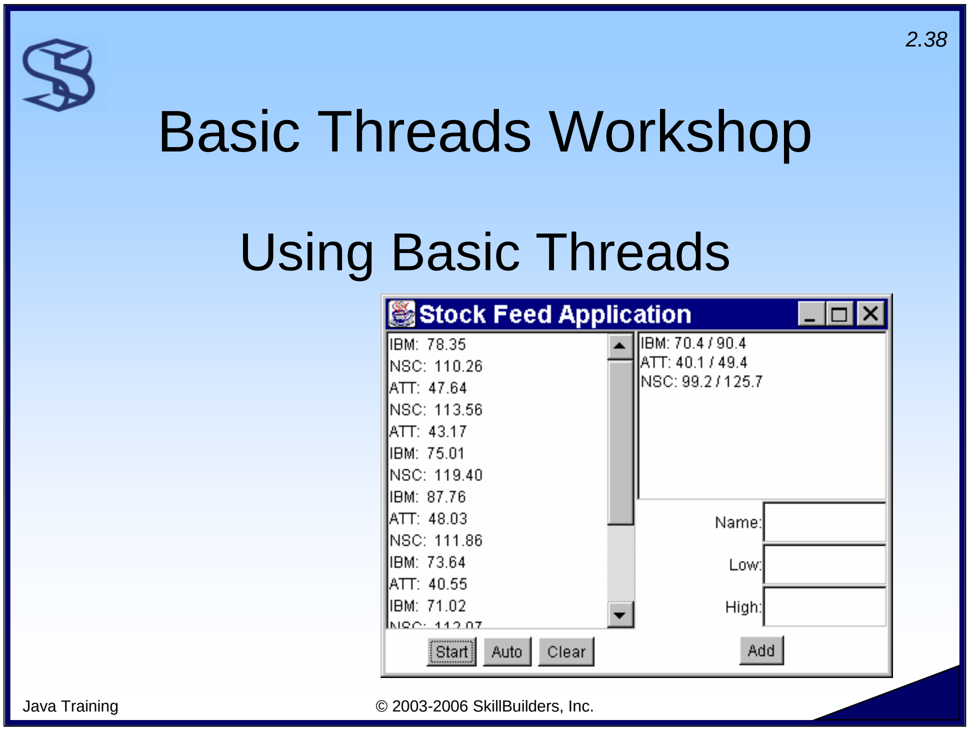
2.37

Where We've Been

- Java threading centers on `Thread` class
- To link code to `Thread` object:
  - Extend `Thread` class
  - Implement `Runnable` interface
- To pause a thread use `sleep( )`

Java Training

© 2003-2006 SkillBuilders, Inc.



2.38

Basic Threads Workshop

Using Basic Threads

```
IBM:  78.35
NSC:  110.26
ATT:  47.64
NSC:  113.56
ATT:  43.17
IBM:  75.01
NSC:  119.40
IBM:  87.76
ATT:  48.03
NSC:  111.86
IBM:  73.64
ATT:  40.55
IBM:  71.02
NSC:  112.07
```

```
IBM: 70.4 / 90.4
ATT: 40.1 / 49.4
NSC: 99.2 / 125.7
```

Java Training

© 2003-2006 SkillBuilders, Inc.

Your Objective

In this workshop you will create some basic threads using the two techniques discussed: extending `Thread` and implementing `Runnable`. Note the illustration above.

You will use a GUI console application that displays a stream of stock prices.

- The stocks, with their low and high prices, are listed on the right. The fields and button below allow you to add to the list.
- When the user clicks `Start`, a thread is created for each stock that feeds a stream of quotations for that stock to the list. Clicking the button again stops the threads.
- Clicking `Auto` runs the streams automatically for a fixed period of time.

A. Examine the Application

The application has been started for you and is found in `StockApp.java` in the directory indicated by the instructor. *The source code is reprinted at the end of the instructions for this exercise.* Copy it to your working directory and compile it.

Run it as follows: `java StockApp`

Continues…

You should be able to add stocks to the list. Now open the file. It has these classes:

- `Stock` encapsulates a single stock with its name, low and high.
- `StockAppGui` is a frame containing the interface and the listeners for the buttons.
- `StockApp` has a `main( )` method that simply instantiates `StockAppGui`.

The `Stock` Class

The `Stock` class has this listing:

```
class Stock {
    public Stock( String name, double low, double high );
    public String getName( );
    public double getLow( );
    public double getHigh( );
    public double getPrice( );
    public String getPriceAsString( );
    public String toString( );
   }
```

The `StockAppGui` Class

`StockAppGui` has this listing:

```
class StockAppGui extends Frame {
   // Fields
   private List lstPrices_          // For stock quotes
   private Hashtable htStocks_   // For stock objects

   // Constructor
   public StockAppGui( String[ ] args );  // For command-line args

   // Application methods
   private void addStock(String sName, String sLow, String sHigh);
   private void startStream( );     // Start stream of stock quotes
   private void stopStream( );      // Stop stream
   private void startAutoRun( );    // Run stream automatically
   private void stopAutoRun( );     // Stop automatic stream
   private void toggleAutoRun( );   // Toggle automatic stream
}
```

Continues…

B. Write a **`StockStream`** Class

Write a `StockStream` class that extends `Thread` and encapsulates a `Stock` and a `java.awt.List`. At random intervals the thread gets a price from the `Stock` object and adds it (along with the stock name) to the list.

1. In the same Java file, write a non-public `StockStream` class that extends `Thread`. Give it fields for the `Stock` and the `List`.

2. Give the class a constructor that takes an argument for each and assigns them to the fields. Use the stock name as the `Thread` name.

3. Override the `run( )` method to run an infinite loop. In the loop, get a price, add a string with the stock name and price to the list, then sleep for a random amount of time. For sleeping, use an algorithm like this:
   ```
   Thread.sleep((int) (Math.random() * 500) + 500);
   ```
   Tip: To make the newly added quote appear in the visible portion of the list, do the following (where `listName` represents your `List` variable):
   ```
   listName.add( stringOfData );
   listName.makeVisible( listName.getItemCount() – 1 );
   ```

4. Save and compile your work.

C. Use the **`StockStream`** Class

1. In the `StockAppGui` class declare and instantiate a private `Vector` to hold threads.

2. Code the `startStream( )` method to go through the hashtable of stocks; for each stock, create a `StockStream` object, add it to the vector and start it.

3. Code the `stopStream( )` method to go through the vector, stop each thread, then clear the vector (use the `clear( )` method).

4. Compile and test your work. When you click the `Start` button, you should see a stream of stock quotes in the list. Clicking `Stop` should turn it off.

Continues…

D. Run the Stream Automatically

Create another thread to start the stream, pause 5 seconds or so, then stop the stream. In this case, make `StockAppGui` a `Runnable` and give it to a private thread.

1. Make `StockAppGui` a `Runnable`.
2. In its `run( )` method, start the stream, sleep for 5 seconds, then stop the stream.
3. Declare an instance level `Thread` variable called `thrAuto_` (but do not initialize it in the declaration).
4. In the `startAutoRun( )` method, create a thread with the `StockAppGui` instance as the target, assign it to `thrAuto_`, then start the thread.
5. Compile and test your work. Clicking the `Auto` button calls `toggleAutoRun( )`, which calls `startAutoRun( ).` As a result you should see the quote stream run for a finite period of time.

The Initial Application

```
// StockApp.java

// Threads Lab 1.

import java.awt.*;
import java.awt.event.*;
import java.util.Hashtable;



// Stock class encapsulates a single stock
class Stock {
   private String name_;
   private double low_, high_;

   // Basic c'tor for setting attributes
   public Stock( String name, double low, double high ) {
      name_ = name;
      low_  = low;
      high_ = high;
   }

   // Basic getters for attributes
   public String  getName() { return name_; }
   public double  getLow()  { return low_; }
   public double  getHigh() { return high_; }

   public double getPrice() {
      // Get a random price between low and high
      return low_ + (Math.random() * (high_ - low_));
   }
```

```
      public String getPriceAsString() {
         String s = "" + getPrice();
         return s.substring(0, s.indexOf('.') + 3);
      }

      public String toString() {
         return getName() + ": " + getLow() + " / " + getHigh();
      }
   }

   class StockAppGui extends Frame {
      private String[]  args_;
      private List      lstPrices_ = new List();
      private List      lstStocks_ = new List();
      private TextField tfName_    = new TextField();
      private TextField tfLow_     = new TextField();
      private TextField tfHigh_    = new TextField();
      private ButtonHandler btnh_  = new ButtonHandler();
      private Hashtable htStocks_  = new Hashtable();

      public StockAppGui( String[] args ) {
         super( "Stock Feed Application" );
         args_ = args;
         initLayout();

         // Add a few stocks just for convenience:
         addStock( "IBM", "70.4", "90.4" );
         addStock( "ATT", "40.1", "49.4" );
         addStock( "NSC", "99.2", "125.7" );

         show();
      }
```

```
// APP METHODS
private void addStock(String sName, String sLow, String sHigh) {
   // Instantiate Stock with values from TextFields
   double low  = Double.valueOf( sLow  ).doubleValue();
   double high = Double.valueOf( sHigh ).doubleValue();
   Stock  stk  = new Stock( sName, low, high );

   htStocks_.put( sName, stk );          // Add to hashtable
   lstStocks_.add( stk.toString() );     // Add to list

   // Clear text fields
   tfName_.setText("");
   tfLow_. setText("");
   tfHigh_.setText("");
   tfName_.requestFocus();
}

private void startStream() {
}

private void stopStream() {
}

private void startAutoRun() {
}

private void stopAutoRun() {
}
```

```
private void toggleAutoRun() {
   startAutoRun();
}



// SETUP METHODS


private void initLayout() {
   Panel pnlMain = new Panel();
   pnlMain.setLayout( new GridLayout(1,2) );
   pnlMain.add( getStockFeedPanel() );
   pnlMain.add( getStockPanel() );


   add( BorderLayout.CENTER, pnlMain );  // Add main panel to
frame
   setBounds( 100, 100, 400, 300 );


   // Add WindowListener to close window
   addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent we ) {
         System.exit(0);
      }
   } );


}


private Panel getStockFeedPanel() {
   Panel pnl = new Panel();
   pnl.setLayout( new BorderLayout() );
   pnl.add( BorderLayout.CENTER, lstPrices_ );
   pnl.add( BorderLayout.SOUTH, getStockFeedButtonPanel() );
   return pnl;
}
```

```
   private Panel getStockFeedButtonPanel() {
      Panel pnl    = new Panel();
      Button btn[] = new Button[3];
      btn[0] = new Button("Start");
      btn[1] = new Button("Auto");
      btn[2] = new Button("Clear");
      for( int i = 0; i < btn.length; i++ ) {
         btn[i].setActionCommand(btn[i].getLabel().toLowerCase());
         btn[i].addActionListener( btnh_ );
         pnl.add(btn[i]);
      }
      return pnl;
   }

   private Panel getStockPanel() {
      Panel pnl = new Panel();
      pnl.setLayout( new GridLayout(2,1) );
      pnl.add( lstStocks_ );
      pnl.add( getStockEntryPanel() );
      return pnl;
   }

   private Panel getStockEntryPanel() {
      // Set up panel
      Panel pnlMain = new Panel();
      pnlMain.setLayout( new BorderLayout() );

      Panel pnlCenter = new Panel();
      pnlCenter.setLayout( new GridLayout(3,2) );
      pnlCenter.add(new Label("Name:", Label.RIGHT));
      pnlCenter.add(tfName_);
      pnlCenter.add(new Label("Low:",  Label.RIGHT));
      pnlCenter.add(tfLow_);
      pnlCenter.add(new Label("High:", Label.RIGHT));
      pnlCenter.add(tfHigh_);
      pnlMain.add( BorderLayout.CENTER, pnlCenter );

      Panel pnlSouth = new Panel();
      Button btnAdd = new Button("Add");
      btnAdd.setActionCommand(btnAdd.getLabel().toLowerCase());
      btnAdd.addActionListener( btnh_ );
      pnlSouth.add(btnAdd);
      pnlMain.add( BorderLayout.SOUTH, pnlSouth );

      return pnlMain;
   }

   // INNER CLASSES
   private class ButtonHandler implements ActionListener {
      public void actionPerformed( ActionEvent ae ) {
         Button btn = (Button) ae.getSource();
         if( ae.getActionCommand().equals("start")) {
            startStream();
            btn.setLabel("Stop");
            btn.setActionCommand("stop");
         }
         else if( ae.getActionCommand().equals("stop")) {
            stopStream();
            btn.setLabel("Start");
            btn.setActionCommand("start");
         }
         else if( ae.getActionCommand().equals("clear")) {
```

```
                  lstPrices_.removeAll();
         }
         else if( ae.getActionCommand().equals("add")) {
            addStock( tfName_.getText(), tfLow_.getText(),
             tfHigh_.getText() );
         }
         else if( ae.getActionCommand().equals("auto")) {
            toggleAutoRun();
         }
      }
   }  // End of inner class ButtonHandler
}  // End of StockAppGui class


class StockApp {
   public static void main( String[] args ) {
      new StockAppGui(args);
   }
}
```