CSE 686 Internet Programming

Week 5: Threads

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

February 12, 14, 2018

Based on **TCP/IP Sockets in Java**, Kenneth L. Calvert & Michael J. Donahoo, 2008, Chapter 4

How to get current date/time?

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class CurrentTime {
    public static void main(String[] args) {
        long startTime, endTime;

        startTime= System.currentTimeMillis();
        for (int i = 0; i < 1000; i++) System.out.println("Time flies …." );
        endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + " milli-seconds have passed!" );

        Date d = new Date(endTime);
        SimpleDateFormat df = new SimpleDateFormat();
        System.out.println("More readable format: " + df.format(d));
    }
}
```

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

Unix time

From Wikipedia, the free encyclopedia

Unix time (also known as **POSIX time** or **epoch time**)[*citation needed*] is a system for describing a point in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970,[1] minus the number of leap seconds that have taken place since then.[1][2] It is used widely in Unix-like and many other operating systems and file formats. Because the same timestamp can refer to two distinct instants of time around a leap second, it is neither a linear measure of time nor a true representation of UTC. Unix time may be checked on most Unix systems by typing `date +%s` on the command line.

Unix time passed 1,000,000,000 seconds in 2001-09-09T01:46:40Z. It was celebrated in Copenhagen, Denmark at a party held by DKUUG (at 03:46:40 local time).

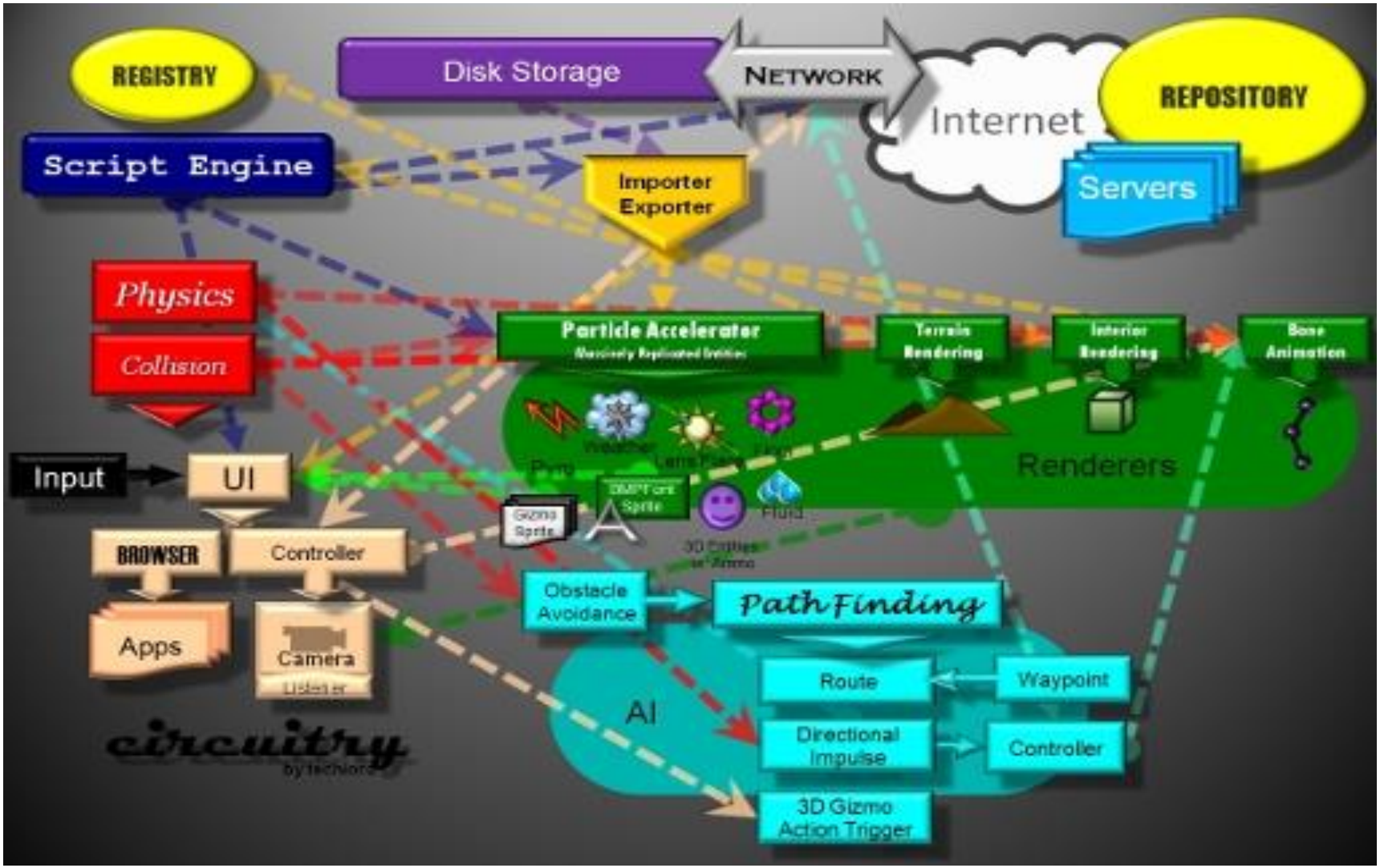
On systems where the representation of Unix time is as a signed 32-bit number, the representation will end after the completion of 2,147,483,647 (2^{31} - 1) seconds from 00:00:00 1 January 1970 minus the number of leap seconds that have taken place since then, which will happen on 19 January, 2038 03:14:08 UTC. This is referred to as the "Year 2038 problem" where the 32-bit signed Unix time will overflow and will take the actual count to negative.

Threads

Threads: Common Examples

- You're probably already familiar with programs that use threads to perform several tasks at once. Here are some common examples:
  - **Web browsers -** can download files while still letting you view Web pages.
  - **E-mail programs** – can download messages before you can read the first message.
  - **Word processors -** can print long documents while you continue to work.
  - **Game programs -** use several threads to handle different parts of the game (graphics, user interaction, network, etc.) to improve the overall responsiveness of the game. (next slide)
  - **GUI-based programs -** use at least two threads **-** one thread to run the application's main logic, and another thread to monitor mouse and keyboard.
  - The **Java Virtual Machine** itself uses threading for its housekeeping chores, such as the garbage collection.

Game Engine Architecture



https://forum.thegamecreators.com/thread/165850

Iterative Servers

- Our basic TCP echo server (TCPEchoServer.java) handles one client at a time. (Use **TCPEchoClientGUI.java** for this.)
- If a client connects while another is already being serviced, the server will **not** echo the new client's data until it has finished with the current client.
- This type of server is known as an **iterative server**.
  - They handle clients sequentially, finishing with one client before servicing the next.
  - They work best for applications where each client requires a small, bounded amount of server connection time
  - If the time to handle a client can be long, the wait experienced by subsequent clients may be unacceptable.

Multi-Threaded Server

- What we need is some way for each connection to proceed independently, without interfering with other connections.
  - Java **threads** provide exactly that
    - A convenient mechanism allowing servers to handle many clients simultaneously. (next slide)
  - In our echo server, we can give each client to an independently executing thread.

java.lang

Class Thread

java.lang.Object
    java.lang.Thread

All Implemented Interfaces:

Runnable

Direct Known Subclasses:

ForkJoinWorkerThread

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

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

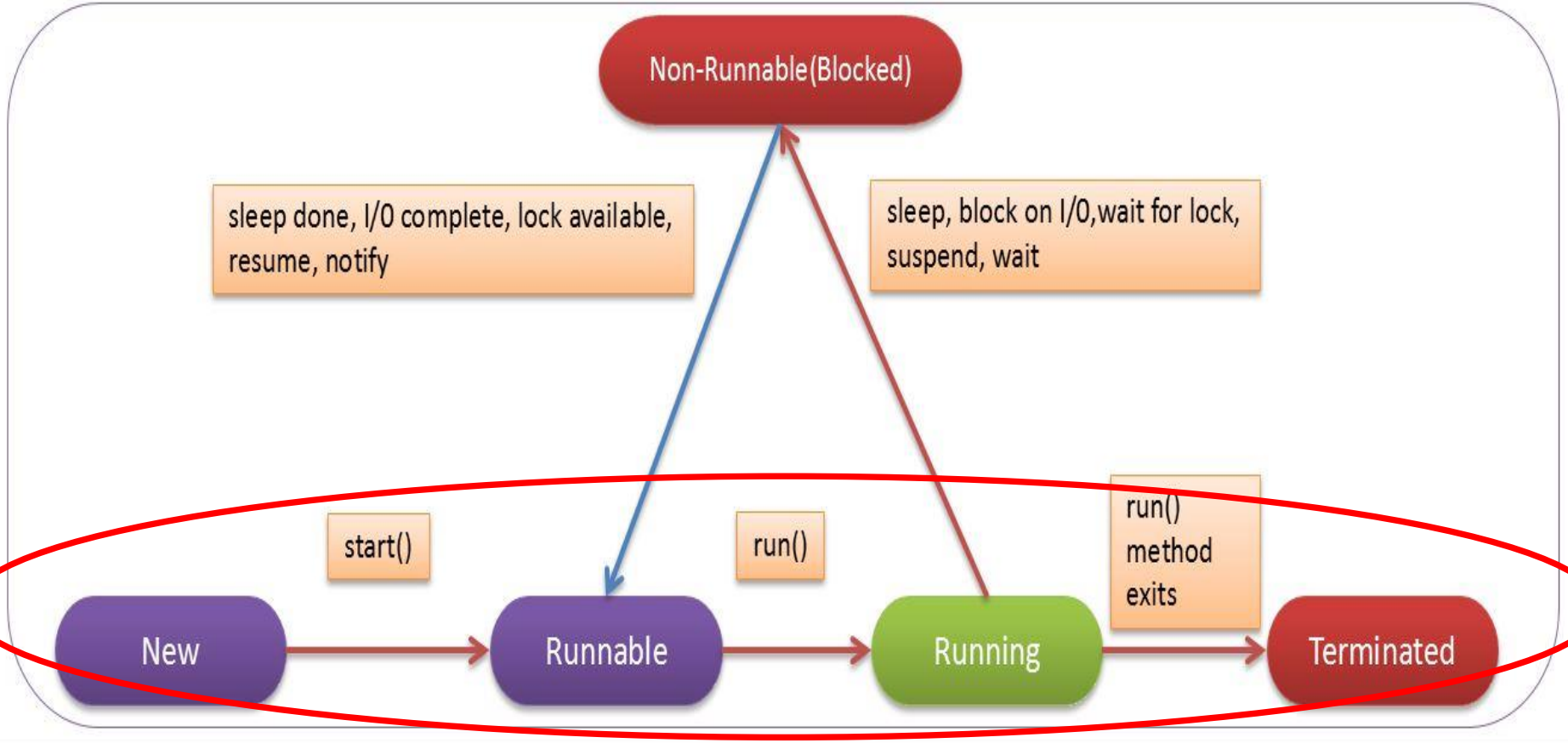
Java Threads

- In Java, **threads** let you divide the work of an application up into separate pieces, which then all run simultaneously. (Divide and Conquer)
  - In Java, a **thread** is a single sequence of executable code within a larger program.
  - All the programs you have seen so far have used just one thread - the **main** thread that starts automatically when you run the program.
  - Java lets you create programs that start additional threads to perform specific tasks.
  - The result is a faster and more efficient program, but along with the increased speed comes more difficult programming and debugging.

Life cycle of a Thread (Thread States)

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows

1. **New** - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. **Runnable** - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. **Running** - The thread is in running state if the thread scheduler has selected it.
4. **Non-Runnable (Blocked) -** This is the state when the thread is still alive, but is currently not eligible to run.
5. **Terminated** - A thread is in terminated or dead state when its run() method exits.



How to Create Threads in Java

- An application that creates an instance of **Thread** must provide the code (for the **run**() method) that will run in that thread.

- There are two ways to do it:
  1. Provide a **Runnable** object.
  2. **Subclass** Thread.

1. Provide a Runnable Object

- The **Runnable** interface defines a single method, **run()**, meant to contain the code executed in the thread. The Runnable object is passed to the **Thread constructor**, as in the following example:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread aThread = new Thread(new HelloRunnable());
        aThread.start();
        // or new Thread(new HelloRunnable()).start();
    }
}
```

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

compact1, compact2, compact3

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Runnable
```

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Since:

See Also:

`Thread`, `Callable`

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
`void`	`run()` When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

Method Detail

run

`void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also:

`Thread.run()`

Constructor Summary

Constructors

Constructor and Description
`Thread()` Allocates a new `Thread` object.
`Thread(Runnable target)` Allocates a new `Thread` object.
`Thread(Runnable target, String name)` Allocates a new `Thread` object.
`Thread(String name)` Allocates a new `Thread` object.
`Thread(ThreadGroup group, Runnable target)` Allocates a new `Thread` object.
`Thread(ThreadGroup group, Runnable target, String name)` Allocates a new `Thread` object so that it has `target` as its run object, has the specified `name` as its name, and belongs to the thread group referred to by `group`.
`Thread(ThreadGroup group, Runnable target, String name, long stackSize)` Allocates a new `Thread` object so that it has `target` as its run object, has the specified `name` as its name, and belongs to the thread group referred to by `group`, and has the specified *stack size*.
`Thread(ThreadGroup group, String name)` Allocates a new `Thread` object.

Method Summary

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws:

`IllegalThreadStateException` - if the thread was already started.

See Also:

`run()`, `stop()`

run

```
public void run()
```

If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

Subclasses of `Thread` should override this method.

Specified by:

`run` in interface `Runnable`

See Also:

`start()`, `stop()`, `Thread(ThreadGroup, Runnable, String)`

stop

A Runnable Example

```
public class LaunchEvent implements Runnable {
        private int start; // start time
        private String message;
        public LaunchEvent(int start, String message) {
                this.start = start;
                this.message = message;
        }
        public void run() {
                try {
                        Thread.sleep(20000 - (start * 1000)); // see next slide
                }
                catch (InterruptedException e) {} // do nothing
                System.out.println(message);
        }
}
```

Pausing Execution with Sleep

- The Thread class defines a number of methods useful for thread management.
  - These include static methods, which provide information about the thread, or affect the status of the thread.
  - **Thread.sleep()** is such a method.
    - It causes the current thread to suspend execution for a specified period.
    - This is an efficient way of making processor time available to the other threads or other applications.

sleep

```
public static void sleep(long millis)
                  throws InterruptedException
```

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.

Parameters:

millis - the length of time to sleep in milliseconds

Throws:

IllegalArgumentException - if the value of millis is negative

InterruptedException - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

sleep

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

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

Parameters:

millis - the length of time to sleep in milliseconds

nanos - 0-999999 additional nanoseconds to sleep

Throws:

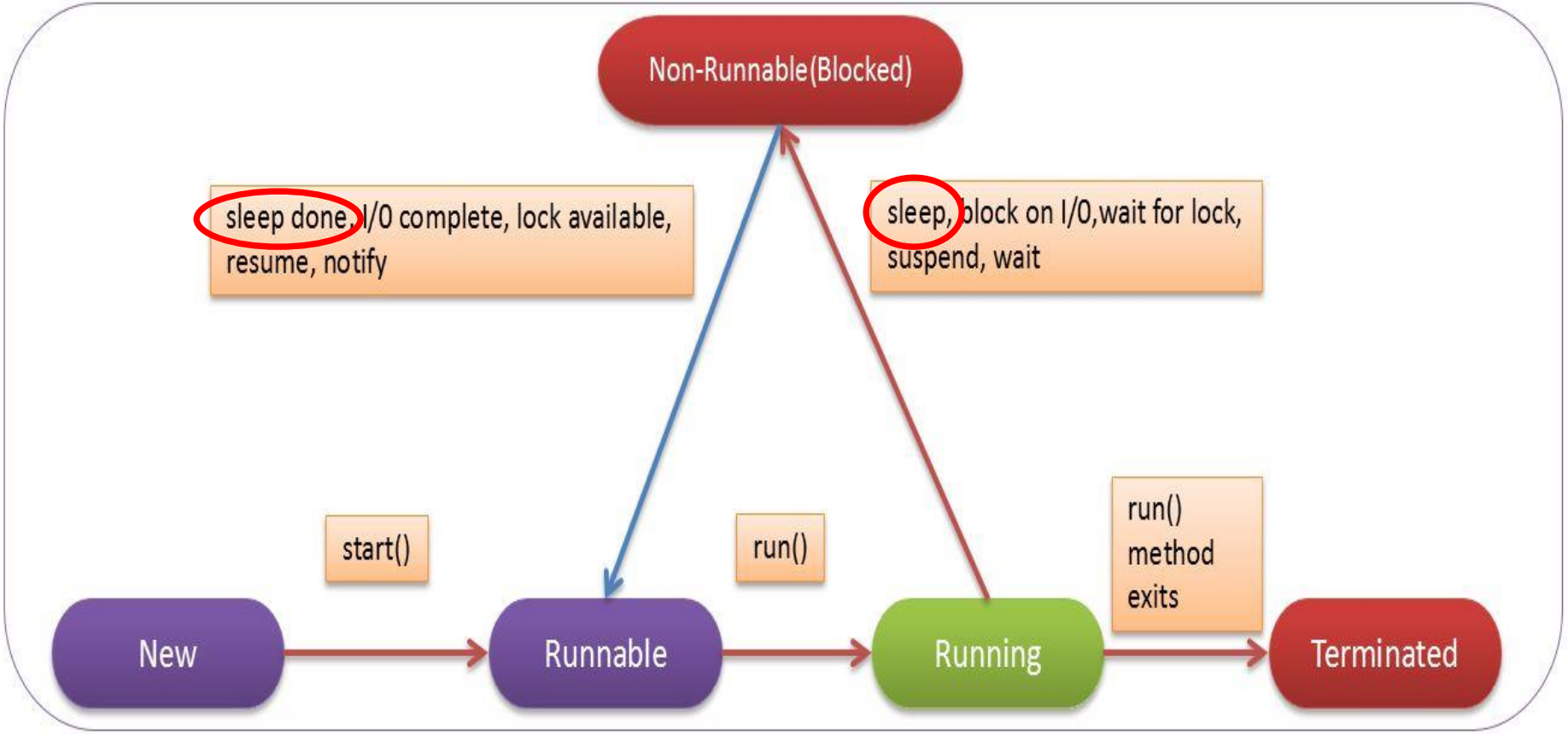
IllegalArgumentException - if the value of millis is negative, or the value of nanos is not in the range 0-999999

InterruptedException - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

Life cycle of a Thread (Thread States)

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows

1. **New** - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. **Runnable** - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. **Running** - The thread is in running state if the thread scheduler has selected it.
4. **Non-Runnable (Blocked) -** This is the state when the thread is still alive, but is currently not eligible to run.
5. **Terminated** - A thread is in terminated or dead state when its run() method exits.



A Runnable Example, Part 2

```
public class CountDownApp {
        public static void main(String[] args) {
                Runnable flood, ignition, liftoff;
                flood = new LaunchEvent(16, "Flood the pad!");
                ignition = new LaunchEvent(6, "Start engines!");
                liftoff = new LaunchEvent(0, "Liftoff!");

                new Thread(flood).start();
                // same as:
                // Thread t1 = new Thread(flood);
                // t1.start();
                new Thread(ignition).start();
                new Thread(liftoff).start();
        }
}
```

2. Subclassing Thread

- The **Thread** class itself implements Runnable, though its **run** method does nothing. An application can subclass **Thread**, providing its own implementation of **run**, as in the following example:

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
         Thread aThread = new HelloThread();
         aThread.start();
        //  or new HelloThread().start();
    }
}
```

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

A Subclassing Example

```
public class CountDownClock extends Thread {
        public void run() {
                for (int t = 20; t >= 0; t--) {
                        System.out.println("T minus " + t);
                        try {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e) {}
                }
        }
        public static void main(String[] args) {
                Thread clock = new CountDownClock();
                clock.start();
                // or just: new CountDownClock().start();
        }
}
```

A Subclassing+Runnable Example

```
public class CountDownApp2 {
        public static void main(String[] args) {
                Runnable flood, ignition, liftoff;
                flood = new LaunchEvent(16, "Flood the pad!");
                ignition = new LaunchEvent(6, "Start engines!");
                liftoff = new LaunchEvent(0, "Liftoff!");

                new CountDownClock().start();

                new Thread(flood).start();
                new Thread(ignition).start();
                new Thread(liftoff).start();
        }
}
```

Runnable vs. Subclassing Thread

- Which approach is better?
  - Subclassing the Thread class is easier to use in simple applications, but is limited by the fact that your class must be a descendant of Thread.
  - The Runnable approach is more general, because a Runnable object can subclass a class other than Thread.
  - Most people choose the Runnable approach.
  - Not only is the Runnable approach more flexible, but it is also applicable to the high-level thread management APIs. (To be discussed later)

```
import java.util.concurrent.TimeUnit;

public class ThreadExample implements Runnable {

  private String greeting; // Message to print to console

  public ThreadExample(String greeting) {
    this.greeting = greeting;
  }

  public void run() {
    while (true) {
      System.out.println(Thread.currentThread().getName() + ":  " + greeting);
      try {
        // Sleep 0 to 100 milliseconds
        TimeUnit.MILLISECONDS.sleep(((long) Math.random() * 100));
      } catch (InterruptedException e) {
      } // Should not happen
    }
  }

  public static void main(String[] args) {
    new Thread(new ThreadExample("Hello")).start();
    new Thread(new ThreadExample("Aloha")).start();
    new Thread(new ThreadExample("Ciao")).start();
  }
}
```

Textbook #1, pp 74-75

- Each of the three statements in main() does the following:
  1. Creates a new instance of ThreadExample with a different greeting string
  2. Passes this new instance to the constructor of Thread
  3. Calls the new Thread instance's **start()** method.

```
import java.util.concurrent.TimeUnit;

public class ThreadExample implements Runnable {

  private String greeting; // Message to print to console

  public ThreadExample(String greeting) {
    this.greeting = greeting;
  }

  public void run() {
    while (true) {
      System.out.println(Thread.currentThread().getName() + ":  " + greeting);
      try {
        // Sleep 0 to 100 milliseconds
        TimeUnit.MILLISECONDS.sleep((long) Math.random() * 100));
      } catch (InterruptedException e) {
      } // Should not happen
    }
  }

  public static void main(String[] args) {
    new Thread(new ThreadExample("Hello")).start();
    new Thread(new ThreadExample("Aloha")).start();
    new Thread(new ThreadExample("Ciao")).start();
  }
}
```

Textbook #1, pp 74-75

The **start()** method in each thread independently executes the **run()** method of ThreadExample, while the main thread terminates.

```
import java.util.concurrent.TimeUnit;

public class ThreadExample implements Runnable {

  private String greeting; // Message to print to console

  public ThreadExample(String greeting) {
    this.greeting = greeting;
  }

  public void run() {
    while (true) {
      System.out.println(Thread.currentThread().getName() + ":  " + greeting);
      try {
        // Sleep 0 to 100 milliseconds
        TimeUnit.MILLISECONDS.sleep(((long) Math.random() * 100));
      } catch (InterruptedException e) {
      } // Should not happen
    }
  }

  public static void main(String[] args) {
    new Thread(new ThreadExample("Hello")).start();
    new Thread(new ThreadExample("Aloha")).start();
    new Thread(new ThreadExample("Ciao")).start();
  }
}
```

The static method **Thread.currentThread()** returns a reference to the thread from which it is called, and **getName()** returns a string containing the name of that thread.

currentThread

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

Returns a reference to the currently executing thread object.

Returns:

the currently executing thread.

yield

```
public static void yield()
```

A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint.

Yield is a heuristic attempt to improve relative progression between threads that would otherwise over-utilise a CPU. Its use should be combined with detailed profiling and benchmarking to ensure that it actually has the desired effect.

It is rarely appropriate to use this method. It may be useful for debugging or testing purposes, where it may help to reproduce bugs due to race conditions. It may also be useful when designing concurrency control constructs such as the ones in the `java.util.concurrent.locks` package.

sleep

```
public static void sleep(long millis)
                  throws InterruptedException
```

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.

Parameters:

millis - the length of time to sleep in milliseconds

Throws:

getName

```
public final String getName()
```

Returns this thread's name.

Returns:

this thread's name.

See Also:

setName(String)

getThreadGroup

```
public final ThreadGroup getThreadGroup()
```

Returns the thread group to which this thread belongs. This method returns null if this thread has died (been stopped).

Returns:

this thread's thread group.

activeCount

```
public static int activeCount()
```

Returns an estimate of the number of active threads in the current thread's thread group and its subgroups. Recursively iterates over all subgroups in the current thread's thread group.

The value returned is only an estimate because the number of threads may change dynamically while this method traverses internal data structures, and might be affected by the presence of certain system threads. This method is intended primarily for debugging and monitoring purposes.

Returns:

an estimate of the number of active threads in the current thread's thread group and in any other thread group that has the current

```
import java.util.concurrent.TimeUnit;

public class ThreadExample implements Runnable {

  private String greeting; // Message to print to console

  public ThreadExample(String greeting) {
    this.greeting = greeting;
  }

  public void run() {
    while (true) {
      System.out.println(Thread.currentThread().getName() + ":  " + greeting);
      try {
        // Sleep 0 to 100 milliseconds
        TimeUnit.MILLISECONDS.sleep((long) Math.random() * 100));
      } catch (InterruptedException e) {
      } // Should not happen
    }
  }

  public static void main(String[] args) {
    new Thread(new ThreadExample("Hello")).start();
    new Thread(new ThreadExample("Aloha")).start();
    new Thread(new ThreadExample("Ciao")).start();
  }
}
```

After printing its instance's greeting message, each thread sleeps for a random amount of time (between 0 and 100 milliseconds).

Math.random() returns a random double between 0.0 and 1.0 (1.0 not included).

Thread.sleep() can be interrupted by another thread, in which case an InterruptedException is thrown.

interrupt

```
public void interrupt()
```

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.

If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.

If this thread is blocked in an I/O operation upon an `InterruptibleChannel` then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a `ClosedByInterruptException`.

If this thread is blocked in a `Selector` then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's `wakeup` method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

Interrupting a thread that is not alive need not have any effect.

Throws:

`SecurityException` - if the current thread cannot modify this thread

interrupted

```
public static boolean interrupted()
```

Tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it).

A thread interruption ignored because a thread was not alive at the time of the interrupt will be reflected by this method returning false.

Synchronizing Threads

- Threads are perfect for implementing servers, in which each client's processing is independent of that provided to every other client.
- However, it is a different story when client processing involves updating information that is **shared** across threads on the server.
- In that case, great care must be taken to ensure that different threads are properly **synchronized** with respect to the shared data; otherwise, the shared information can get into an inconsistent state, and the problem can be very difficult to trace.
  - See Goetz, Brian, with T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, **Java Concurrency in Practice**, Pearson Education, 2006 for more information on this topic.

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Thread Interference: An Example

- Errors may be introduced when multiple threads access shared data, such as an object of the following class:

```
class Counter {
   private int c = 0;
   public void increment() {  c++; }
   public void decrement() { c--; }
   public int value() { return c; }
 }
```

- Suppose Thread A invokes **increment** at about the same time Thread B invokes **decrement**. If the initial value of c is 0, their interleaved actions might follow this sequence:
  - Thread A: Retrieve c.
  - Thread B: Retrieve c.
  - Thread A: Increment retrieved value; result is 1.
  - Thread B: Decrement retrieved value; result is -1.
  - Thread A: Store result in c; c is now 1.
  - Thread B: Store result in c; c is now -1.

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Synchronized Methods

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

- It is now impossible for two invocations of synchronized methods on the same object to interleave.
  - When one thread is executing a synchronized method for an object, all other threads **block** until the first thread is done with the object.
  - When a synchronized method exits, it automatically establishes a **happens-before** relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Building Multi-Threaded Servers

- For the multi-threaded server example, we continue to use the simple Echo Protocol. (See **TCPEchoServerThread.java**)
- The code for the echo protocol is given in the java class **EchoProtocol** (as a Runnable object, see next slide).
  - This class encapsulates the per-client processing in the static method: **handleEchoClient().** (see next slides)
  - This **handleEchoClient()** code is almost identical to the connection-handling portion of **TCPEchoServer.java**, except that a **logging capability** has been added (to be discussed later)

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EchoProtocol implements Runnable {
  private static final int BUFSIZE = 32; // Size (in bytes) of I/O buffer
  private Socket clntSock;               // Socket connect to client
  private Logger logger;                 // Server logger

  public EchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
  }

  public static void handleEchoClient(Socket clntSock, Logger logger) {
    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      int recvMsgSize; // Size of received message
      int totalBytesEchoed = 0; // Bytes received from client
      byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
      // Receive until client closes connection, indicated by -1
      while ((recvMsgSize = in.read(echoBuffer)) != -1) {
        out.write(echoBuffer, 0, recvMsgSize);
        totalBytesEchoed += recvMsgSize;
```

Each instance of EchoProtocol contains a socket for the connection and a reference to the logger instance.

```
    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      int recvMsgSize; // Size of received message
      int totalBytesEchoed = 0; // Bytes received from client
      byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
      // Receive until client closes connection, indicated by -1
      while ((recvMsgSize = in.read(echoBuffer)) != -1) {
        out.write(echoBuffer, 0, recvMsgSize);
        totalBytesEchoed += recvMsgSize;
      }

      logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "
          + totalBytesEchoed + " bytes.");

    } catch (IOException ex) {
      logger.log(Level.WARNING, "Exception in echo protocol", ex);
    } finally {
      try {
        clntSock.close();
      } catch (IOException e) {
      }
    }
  }

  public void run() {
    handleEchoClient(clntSock, logger);
  }
}
```

The run() method simply invokes **handleEchoClient()** with the instance's Socket and Logger references

```
public static void handleEchoClient(Socket clntSock, Logger logger) {
  try {
    // Get the input and output I/O streams from socket
    InputStream in = clntSock.getInputStream();
    OutputStream out = clntSock.getOutputStream();

    int recvMsgSize; // Size of received message
    int totalBytesEchoed = 0; // Bytes received from client
    byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
    // Receive until client closes connection, indicated by -1
    while ((recvMsgSize = in.read(echoBuffer)) != -1) {
      out.write(echoBuffer, 0, recvMsgSize);
      totalBytesEchoed += recvMsgSize;
    }

    logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "
        + totalBytesEchoed + " bytes.");

  } catch (IOException ex) {
    logger.log(Level.WARNING, "Exception in echo protocol", ex);
  } finally {
    try {
      clntSock.close();
    } catch (IOException e) {
    }
  }
}

public void run() {
  handleEchoClient(clntSock, logger);
}
```

Alternatively, the server can call the static method directly.

```
public static void handleEchoClient(Socket clntSock, Logger logger) {
  try {
    // Get the input and output I/O streams from socket
    InputStream in = clntSock.getInputStream();
    OutputStream out = clntSock.getOutputStream();

    int recvMsgSize; // Size of received message
    int totalBytesEchoed = 0; // Bytes received from client
    byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
    // Receive until client closes connection, indicated by -1
    while ((recvMsgSize = in.read(echoBuffer)) != -1) {
      out.write(echoBuffer, 0, recvMsgSize);
      totalBytesEchoed += recvMsgSize;
    }

    logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "
        + totalBytesEchoed + " bytes.");

  } catch (IOException ex) {
    logger.log(Level.WARNING, "Exception in echo protocol", ex);
  } finally {
    try {
      clntSock.close();
    } catch (IOException e) {
    }
  }
}
```

Implements the echo protocol

Get the input/output streams from the socket

Receive and echo
Loop until the connection is closed, writing whatever is received back immediately.

Record the client connection details in the log

Handle exceptions:
log any exceptions.

The Thread-per-Client Server

- In a thread-per-client server, a new thread is created to handle each connection.
  - The server executes a loop that runs forever, listening for connections on a specified port and repeatedly accepting an incoming connection from a client and then spawning a new thread to handle that connection.
  - **TCPEchoServerThread.java** implements such thread-per-client server architecture.
    - It is very similar to the iterative server, but the main difference is that it creates a thread to handle the connection instead of handling it directly.
    - When several clients connect at approximately the same time, later ones do not have to wait for the server to finish with the earlier ones before they get service.
    - They all appear to receive service at the same time, albeit at a somewhat slower rate.

Textbook #1, p. 81
Download from the book site

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Logger;

public class TCPEchoServerThread {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(echoServPort);

    Logger logger = Logger.getLogger("practical");

    // Run forever, accepting and spawning a thread for each connection
    while (true) {
      Socket clntSock = servSock.accept(); // Block waiting for connection
      // Spawn thread to handle new connection
      Thread thread = new Thread(new EchoProtocol(clntSock, logger));
      thread.start();
      logger.info("Created and started Thread " + thread.getName());
    }
    /* NOT REACHED */
  }
}
```

Parameter parsing and server socket/logger creation

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Logger;

public class TCPEchoServerThread {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(echoServPort);

    Logger logger = Logger.getLogger("practical");

    // Run forever, accepting and spawning a thread for each connection
    while (true) {
      Socket clntSock = servSock.accept(); // Block waiting for connection
      // Spawn thread to handle new connection
      Thread thread = new Thread(new EchoProtocol(clntSock, logger));
      thread.start();
      logger.info("Created and started Thread " + thread.getName());
    }
    /* NOT REACHED */
  }
}
```

Since **EchoProtocol** implements the Runnable interface, we can give our new instance to the Thread constructor, and the new thread will execute the run() method of EchoProtocol (which in turn calls **handleEchoClient())** when start() is invoked.

Loop forever, handling incoming connections

Start the new thread for the connection and log it

The Logger Class

The java.util.logging Package

- Your server is up and running with thousands of clients per minute.
- Now a user reports a problem. How do you determine what happened?
  - Is the problem at your server? Perhaps the client is violating the protocol?
  - To deal with this scenario, most servers log their activities.
  - This practice is so common that Java now includes built-in logging facilities in the **java.util.logging** package.

Package java.util.logging

Provides the classes and interfaces of the Java™ 2 platform's core logging facilities.

See: Description

Interface Summary

Interface	Description
Filter	A Filter can be used to provide fine grain control over what is logged, beyond the control provided by log levels.
LoggingMXBean	The management interface for the logging facility.

Class Summary

Class	Description
ConsoleHandler	This `Handler` publishes log records to `System.err`.
ErrorManager	ErrorManager objects can be attached to Handlers to process any error that occurs on a Handler during Logging.
FileHandler	Simple file logging `Handler`.
Formatter	A Formatter provides support for formatting LogRecords.
Handler	A `Handler` object takes log messages from a `Logger` and exports them.
Level	The Level class defines a set of standard logging levels that can be used to control logging output.
Logger	A Logger object is used to log messages for a specific system or application component.
LoggingPermission	The permission which the SecurityManager will check when code that is running with a SecurityManager calls one of the logging control methods (such as Logger.setLevel).
LogManager	There is a single global LogManager object that is used to maintain a set of shared state about Loggers and log services.
LogRecord	LogRecord objects are used to pass logging requests between the logging framework and individual log Handlers.

The Logger Class

- The **Logger** class represents a logging facility.
  - Through an instance of this class, we can record the various server activities as shown in EchoProtocol.java.
- You may use several loggers in your server, each serving a different purpose and potentially behaving in a different way.
  - For example, you may have separate loggers for (normal) operations, security, and error messages.
- To get an instance of Logger, call the static factory method **Logger.getLogger()** as follows:

  Logger logger = Logger.**getLogger**("*practical*");

- In Java each logger is identified by a globally unique name.

compact1, compact2, compact3

java.util.logging

Class Logger

java.lang.Object
    java.util.logging.Logger

```
public class Logger
extends Object
```

A Logger object is used to log messages for a specific system or application component. Loggers are normally named, using a hierarchical dot-separated namespace. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component, such as java.net or javax.swing. In addition it is possible to create "anonymous" Loggers that are not stored in the Logger namespace.

Logger objects may be obtained by calls on one of the getLogger factory methods. These will either create a new Logger or return a suitable existing Logger. It is important to note that the Logger returned by one of the `getLogger` factory methods may be garbage collected at any time if a strong reference to the Logger is not kept.

Logging messages will be forwarded to registered Handler objects, which can forward the messages to a variety of destinations, including consoles, files, OS logs, etc.

Each Logger keeps track of a "parent" Logger, which is its nearest existing ancestor in the Logger namespace.

Each Logger has a "Level" associated with it. This reflects a minimum Level that this logger cares about. If a Logger's level is set to `null`, then its effective level is inherited from its parent, which may in turn obtain it recursively from its parent, and so on up the tree.

The log level can be configured based on the properties from the logging configuration file, as described in the description of the LogManager class. However it may also be dynamically changed by calls on the Logger.setLevel method. If a logger's level is changed the change may also affect child loggers, since any child logger that has `null` as its level will inherit its effective level from its parent.

On each logging call the Logger initially performs a cheap check of the request level (e.g., SEVERE or FINE) against the effective log level of the logger. If the request

getLogger

```
public static Logger getLogger(String name)
```

Find or create a logger for a named subsystem. If a logger has already been created with the given name it is returned. Otherwise a new logger is created.

If a new logger is created its log level will be configured based on the LogManager configuration and it will configured to also send logging output to its parent's Handlers. It will be registered in the LogManager global namespace.

Note: The LogManager may only retain a weak reference to the newly created Logger. It is important to understand that a previously created Logger with the given name may be garbage collected at any time if there is no strong reference to the Logger. In particular, this means that two back-to-back calls like `getLogger("MyLogger").log(...)` may use different Logger objects named "MyLogger" if there is no strong reference to the Logger named "MyLogger" elsewhere in the program.

Parameters:

`name - A name for the logger. This should be a dot-separated name and should normally be based on the package name or class name of the subsystem, such as java.net or javax.swing`

Returns:

`a suitable Logger`

Throws:

`NullPointerException - if the name is null.`

getLogger

```
public static Logger getLogger(String name,
                               String resourceBundleName)
```

Find or create a logger for a named subsystem. If a logger has already been created with the given name it is returned. Otherwise a new logger is created.

If a new logger is created its log level will be configured based on the LogManager and it will configured to also send logging output to its parent's Handlers. It will be registered in the LogManager global namespace.

The Logger Class

- Now that you have a Logger, what should you log?
  - If the server is operating normally, you may not want to log every single step the server takes, because logging consumes resources such as space for storing log entries and server processor time for writing each entry.
  - On the other hand, if you are trying to debug, you may want to log each and every step.

- To deal with this, logging typically includes the notion of the **level**, or **severity**, of log entries.
  - The **Level** class encapsulates the notion of the importance of messages.

compact1, compact2, compact3

java.util.logging

Class Level

java.lang.Object
    java.util.logging.Level

All Implemented Interfaces:

`Serializable`

```
public class Level
extends Object
implements Serializable
```

The Level class defines a set of standard logging levels that can be used to control logging output. The logging Level objects are ordered and are specified by ordered integers. Enabling logging at a given level also enables logging at all higher levels.

Clients should normally use the predefined Level constants such as Level.SEVERE.

The levels in descending order are:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

The Logger Class

- Each instance of Logger has a current level
  - Seven system-recognized instances of Level are defined:
    - severe, warning, info, config, fine, finer & finest (next slide)
  - They are defined as static fields of the class Level
  - Each level has an associated integer **value**, so that levels are comparable and can be ordered. (see official documentation for those values)
  - Other user-specific levels can be created, but there is rarely any need to do so.
- Each message to be logged also has an associated level
  - Messages with levels below the Logger instance's current level are discarded (i.e., not logged).

Field Summary

Fields

Modifier and Type	Field and Description
static Level	**ALL** ALL indicates that all messages should be logged.
static Level	**CONFIG** CONFIG is a message level for static configuration messages.
static Level	**FINE** FINE is a message level providing tracing information.
static Level	**FINER** FINER indicates a fairly detailed tracing message.
static Level	**FINEST** FINEST indicates a highly detailed tracing message.
static Level	**INFO** INFO is a message level for informational messages.
static Level	**OFF** OFF is a special level that can be used to turn off logging.
static Level	**SEVERE** SEVERE is a message level indicating a serious failure.
static Level	**WARNING** WARNING is a message level indicating a potential problem.

Constructor Summary

Constructors

The Logger Class

- When you log, where do the messages go?
  - The logger sends messages to one or more **Handlers**, which "handle" publishing the messages.
  - By default, a logger has a single **ConsoleHandler** that prints messages to **System.err**.
  - You can change the handler or add additional handlers to a logger (e.g., **FileHandler**).
  - Like a Logger, a handler has a minimum log level, so for a message to be published, its level must be above both the logger and handlers' threshold.

We may want to customize our logger by setting the minimum logging level or the handlers for logging messages.

Logger: Setting/Getting the level and handlers

```
Handler[] getHandlers()
void addHandler(Handler handler)
void removeHandler(Handler handler)

Level getLevel()
void setLevel(Level newLevel)
boolean isLoggable(Level level)
```

The `getHandlers()` method returns an array of all handlers associated with the logger. The `addHandler()` and `removeHandler()` methods allow addition/removal of handlers to/from the logger. The `getLevel()` and `setLevel()` methods get/set the minimum logging level. The `isLoggable()` method returns true if the given level will be logged by the logger.

LoggerTest.java

```
import java.io.IOException;
import java.util.logging.*;
public class LoggerTest {
        public static void main(String[] args) {
                Logger logger = Logger.getLogger("TEST");
                logger.setLevel(Level.SEVERE);
                System.out.println(logger.getLevel());
                // Add a FileHandler
                try {
                        Handler h = new FileHandler("MyLog.txt");
                        logger.addHandler(h);
                        h.setLevel(Level.SEVERE);
                } catch (SecurityException | IOException e) {
                        // Do something
                }
                logger.severe("Testing...");
        }
}
```

Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)

void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

Logger provides fine-grained logging facilities that differentiate between the level and even context (method call, exception, etc.) of the message:

Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)
```

The **severe**(), **warning**(), etc. methods log the given message at the level specified by the method name.

```
void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)

void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

The **entering**() and **exiting**() methods log entering and exiting the given method from the given class.

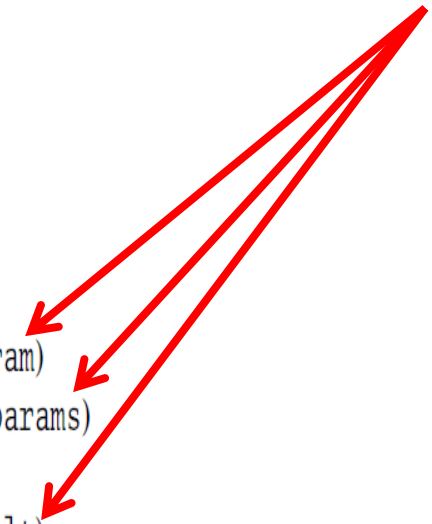
Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)

void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

You may optionally specify additional information such as parameters and return values.



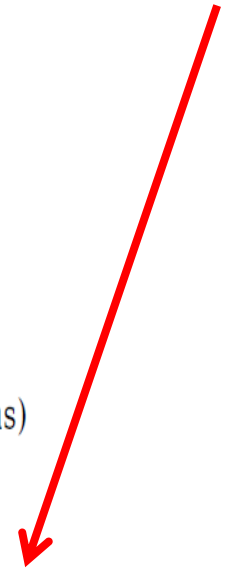
Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)

void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

The **throwing**() method logs an exception thrown in a specific method.



Logger: Logging a message

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)

void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)

void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

The **log**() methods provide a generic logging method where level, message, and (optionally) exception can be logged.

The Logger Class

- An important characteristic of Logger for our purposes is that it is **thread-safe**
  - Its methods can be called from different threads running concurrently without requiring additional synchronization among the callers.
  - Without this feature, different messages logged by different threads might end up being interleaved in the log!

Textbook #1, pp 76-77

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EchoProtocol implements Runnable {
  private static final int BUFSIZE = 32; // Size (in bytes) of I/O buffer
  private Socket clntSock;               // Socket connect to client
  private Logger logger;                 // Server logger

  public EchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
  }

  public static void handleEchoClient(Socket clntSock, Logger logger) {
    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      int recvMsgSize; // Size of received message
      int totalBytesEchoed = 0; // Bytes received from client
      byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
      // Receive until client closes connection, indicated by -1
      while ((recvMsgSize = in.read(echoBuffer)) != -1) {
        out.write(echoBuffer, 0, recvMsgSize);
        totalBytesEchoed += recvMsgSize;
```

```
    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();

      int recvMsgSize; // Size of received message
      int totalBytesEchoed = 0; // Bytes received from client
      byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
      // Receive until client closes connection, indicated by -1
      while ((recvMsgSize = in.read(echoBuffer)) != -1) {
        out.write(echoBuffer, 0, recvMsgSize);
        totalBytesEchoed += recvMsgSize;
      }

      logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "
          + totalBytesEchoed + " bytes.");

    } catch (IOException ex) {
      logger.log(Level.WARNING, "Exception in echo protocol", ex);
    } finally {
      try {
        clntSock.close();
      } catch (IOException e) {
      }
    }
  }

  public void run() {
    handleEchoClient(clntSock, logger);
  }
}
```

Thread Pools

Context Switch

- Every new thread consumes system resources:
  - Spawning a thread takes CPU cycles
  - Each thread has its own data structures (e.g., stacks) that consume system memory.
- In addition, when one thread blocks, the JVM saves its state, selects another thread to run, and restores the state of the chosen thread. (**context switch)**.
  - As the number of threads increases, more and more system resources are consumed by this kind of hread overhead.
  - Eventually, the system is spending more time dealing with context switching and thread management than with servicing connections.
  - At that point, adding an additional thread may actually increase client service time.

Thread Pools

- We can avoid this problem by limiting the total number of threads and reusing threads.
  - Instead of spawning a new thread for each connection, the server creates a **thread pool** on start-up by spawning a fixed number of threads.
  - When a new client connection arrives at the server, it is assigned to a thread from the pool.
  - When the thread finishes with the client, it returns to the pool, ready to handle another request.
  - Connection requests that arrive when all threads in the pool are busy are queued to be serviced by the next available thread.
- An example of the thread-pool paradigm is shown in **TCPEchoServerPool.java.**

Textbook #1, pp 82-83

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TCPEchoServerPool {

  public static void main(String[] args) throws IOException {

    if (args.length != 2) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port
    int threadPoolSize = Integer.parseInt(args[1]);

    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
```

```
public class TCPEchoServerPool {

  public static void main(String[] args) throws IOException {

    if (args.length != 2) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");
    }
    int echoServPort = Integer.parseInt(args[0]); // Server port
    int threadPoolSize = Integer.parseInt(args[1]);

    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
```

The port number to listen on and the number of threads are both passed as arguments to main().

```
public class TCPEchoServerPool {

  public static void main(String[] args) throws IOException {

    if (args.length != 2) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port
    int threadPoolSize = Integer.parseInt(args[1]);

    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
```

After parsing them we create the ServerSocket and Logger instances.

Note that both have to be declared final, because they are referenced inside the anonymous class instance created below.

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

For each loop iteration, an instance of an anonymous class that extends Thread is created.

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

When the start() method of this instance is called, the thread executes the run() method

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

Since there are *N* different threads executing the same loop, up to *N* threads can be blocked on *servSock*'s accept(), waiting for an incoming connection. The system ensures that only one thread gets a Socket for any particular connection

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

The static handleEchoClient() method encapsulates knowledge of the protocol details. It logs the connection when it finishes, as well as any exceptions encountered along the way.

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

Handle exception from accept()

```
    // Create a server socket to accept client connection requests
    final ServerSocket servSock = new ServerSocket(echoServPort);

    final Logger logger = Logger.getLogger("practical");

    // Spawn a fixed number of threads to service clients
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept(); // Wait for a connection
              EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " + thread.getName());
    }
  }
}
```

When the start() method of this instance is called, the thread executes the run() method

The Executor Interface

- In large-scale applications, it makes sense to separate thread management and creation from the rest of the application.
- Objects that encapsulate these functions are known as **executors**.
  - **Thread Pools** are the most common kind of executors.
- The java.util.concurrent package defines three executor interfaces:
  - **Executor**, a simple interface that supports launching new tasks.
  - **ExecutorService**, a sub-interface of Executor, which adds features that help manage the lifecycle of the individual tasks and of the executor itself.
  - **ScheduledExecutorService**, a sub-interface of ExecutorService, supports future and/or periodic execution of tasks. (Beyond scope)

The Executor Interface

- The Executor interface represents an object that executes Runnable instances according to some strategy, which may include details about queuing and scheduling, or how jobs are selected for execution.

- The Executor interface specifies a single method:

```
interface Executor {
        void execute(Runnable task);
}
```

compact1, compact2, compact3

java.util.concurrent

Interface Executor

All Known Subinterfaces:

ExecutorService, ScheduledExecutorService

All Known Implementing Classes:

AbstractExecutorService, ForkJoinPool, ScheduledThreadPoolExecutor, ThreadPoolExecutor

---

public interface **Executor**

An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

The Executor Interface

- The single method provided by the **Executor** interfaces, **execute()**, is designed to be a drop-in replacement for a common thread-creation method.

  - If **r** is a Runnable object, and **e** is an Executor object you can replace

    (new Thread(r)).start();

  - with

    e.execute(r);

The ExecutorService Sub-Interface

- The **ExecutorService** interface extends Executor to provide a more sophisticated facility that allows a service to be shut down, either gracefully or abruptly.

- Instances of ExecutorService can be obtained by calling various static factory methods of the convenience class **Executors**.

- The program **TCPEchoServerExecutor.java** illustrates the use of the basic Executor facilities.

Textbook #1, p. 85

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.logging.Logger;

public class TCPEchoServerExecutor {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(echoServPort);

    Logger logger = Logger.getLogger("practical");

    Executor service = Executors.newCachedThreadPool();  // Dispatch svc

    // Run forever, accepting and spawning a thread for each connection
    while (true) {
      Socket clntSock = servSock.accept(); // Block waiting for connection
      service.execute(new EchoProtocol(clntSock, logger));
    }
    /* NOT REACHED */
  }
}
```

The port is the only argument.

We create the **ServerSocket** and **Logge**r instances as before

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.logging.Logger;

public class TCPEchoServerExecutor {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(echoServPort);

    Logger logger = Logger.getLogger("practical");

    Executor service = Executors.newCachedThreadPool();  // Dispatch svc

    // Run forever, accepting and spawning a thread for each connection
    while (true) {
      Socket clntSock = servSock.accept(); // Block waiting for connection
      service.execute(new EchoProtocol(clntSock, logger));
    }
    /* NOT REACHED */
  }
}
```

The static factory method **newCachedThreadPool()** creates an instance of ExecutorService.

However, it first tries to reuse an existing thread. When a thread has been idle for at least **60** seconds, it is removed from the pool.

When **its execute()** method is invoked with a **Runnable** instance, the executor service creates a new thread to handle the task if necessary.

newCachedThreadPool


```
public static ExecutorService newCachedThreadPool()
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to `execute` will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using `ThreadPoolExecutor` constructors.

Returns:

the newly created thread pool

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new threads when needed.

Parameters:

threadFactory - the factory to use when creating new threads

Returns:

the newly created thread pool

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.logging.Logger;

public class TCPEchoServerExecutor {

  public static void main(String[] args) throws IOException {

    if (args.length != 1) { // Test for correct # of args
      throw new IllegalArgumentException("Parameter(s): <Port>");
    }

    int echoServPort = Integer.parseInt(args[0]); // Server port

    // Create a server socket to accept client connection requests
    ServerSocket servSock = new ServerSocket(echoServPort);

    Logger logger = Logger.getLogger("practical");

    Executor service = Executors.newCachedThreadPool();  // Dispatch svc

    // Run forever, accepting and spawning a thread for each connection
    while (true) {
      Socket clntSock = servSock.accept(); // Block waiting for connection
      service.execute(new EchoProtocol(clntSock, logger));
    }
    /* NOT REACHED */
  }
}
```

Note that in the steady state, the **cached thread pool** ends up having about the right number of threads, so that each thread stays busy and creation/destruction of threads is rare.

When a new connection arrives, a new EchoProtocol instance is created and passed to **execute()**, which either hands it off to an already-existing thread or creates a new one

The Executor Interface

- We can change dispatching strategies simply by changing the kind of **Executor** we instantiate.
  - For example, if we wanted to use a fixed-size thread pool as in our **TCPEchoServerPool**.java example, it is a matter of changing one line associated with setting the dispatch service:

    Executor service = Executors.**newFixedThreadPool**(threadPoolSize);

- We could switch to a single thread to execute all connections either by specifying a pool size of 1, or by the following call:

  Executor service = Executors.**newSingleThreadExecutor**();

  - In this Executor approach, if the single "worker" thread dies because of some failure, the Executor will replace it with a new thread.

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most `nThreads` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly `shutdown`.

Parameters:

`nThreads` - the number of threads in the pool

Returns:

the newly created thread pool

Throws:

`IllegalArgumentException` - if nThreads <= 0

newWorkStealingPool

```
public static ExecutorService newWorkStealingPool(int parallelism)
```

Creates a thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce contention. The parallelism level corresponds to the maximum number of threads actively engaged in, or available to engage in, task processing. The actual number of threads may grow and shrink dynamically. A work-stealing pool makes no guarantees about the order in which submitted tasks are executed.

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor()
```

Creates an Executor that uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent `newFixedThreadPool(1)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Returns:

the newly created single-threaded Executor

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)
```

Creates an Executor that uses a single worker thread operating off an unbounded queue, and uses the provided ThreadFactory to create a new thread when needed. Unlike the otherwise equivalent `newFixedThreadPool(1, threadFactory)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Parameters:

threadFactory - the factory to use when creating new threads

Returns:

the newly created single-threaded Executor

Throws:

Blocking and Timeouts

- Socket I/O calls may block for several reasons:
  - Data input methods, such as **read**(), block if data is not available.
  - A **write**() on a socket may block if there is not sufficient space to buffer the transmitted data.
  - The **accept**() method of ServerSocket() and the Socket constructor both block until a connection has been established.
- Long roundtrip times, high error rate connections, and slow (or deceased) servers may cause connection establishment to take a long time.
- In all of these cases, the method blocks until the request has been satisfied.
- A blocked method halts progress of the application, and makes the thread running it useless. The following slides provide some solutions.

setSoTimeout

For Socket

```
public void setSoTimeout(int timeout)
                  throws SocketException
```

Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds. With this option set to a non-zero timeout, a read() call on the InputStream associated with this Socket will block for only this amount of time. If the timeout expires, a **java.net.SocketTimeoutException** is raised, though the Socket is still valid. The option **must** be enabled prior to entering the blocking operation to have effect. The timeout must be > 0. A timeout of zero is interpreted as an infinite timeout.

Parameters:

`timeout` - the specified timeout, in milliseconds.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

JDK 1.1

See Also:

`getSoTimeout()`

For **accept**() and **read**(), we can set a bound on the maximum time (in milliseconds) to block, using the **setSoTimeout**() method of **Socket** and **ServerSocket**.

getSoTimeout

```
public int getSoTimeout()
                  throws SocketException
```

Returns setting for SO_TIMEOUT. 0 returns implies that the option is disabled (i.e., timeout of infinity).

Returns:

setSoTimeout

```
public void setSoTimeout(int timeout)
                  throws SocketException
```

For ServerSocket

Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds. With this option set to a non-zero timeout, a call to accept() for this ServerSocket will block for only this amount of time. If the timeout expires, a **java.net.SocketTimeoutException** is raised, though the ServerSocket is still valid. The option **must** be enabled prior to entering the blocking operation to have effect. The timeout must be > 0. A timeout of zero is interpreted as an infinite timeout.

Parameters:

`timeout` - the specified timeout, in milliseconds

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

JDK1.1

See Also:

`getSoTimeout()`

For **accept**() and **read**(), we can set a bound on the maximum time (in milliseconds) to block, using the **setSoTimeout**() method of **Socket** and **ServerSocket**.

getSoTimeout

```
public int getSoTimeout()
                  throws IOException
```

Retrieve setting for SO_TIMEOUT. 0 returns implies that the option is disabled (i.e., timeout of infinity).

Returns:

Method Summary

For **Socket** instances, we can also use the **available()** method of the socket's **InputStream** to check for available data before calling read().

Methods

Modifier and Type	Method and Description
int	**available**() Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void	**close**() Closes this input stream and releases any system resources associated with the stream.
void	**mark**(int readlimit) Marks the current position in this input stream.
boolean	**markSupported**() Tests if this input stream supports the mark and reset methods.
abstract int	**read**() Reads the next byte of data from the input stream.
int	**read**(byte[] b) Reads some number of bytes from the input stream and stores them into the buffer array b.
int	**read**(byte[] b, int off, int len) Reads up to len bytes of data from the input stream into an array of bytes.
void	**reset**() Repositions this stream to the position at the time the mark method was last called on this input stream.
long	**skip**(long n) Skips over and discards n bytes of data from this input stream.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

available(): A Reminder

public int available()

- If you do not want to wait until all the bytes you need are immediately available, you can use the **available()** method to determine how many bytes can be read without blocking.
  - This returns the minimum number of bytes you can read.
  - You may in fact be able to read more, but you will be able to read at least as many bytes as **available()** suggests:

```
int bytesAvailable = in.available(); // in is an InputStream
byte[] input = new byte[bytesAvailable];
int bytesRead = in.read(input, 0, bytesAvailable);
// continue with rest of program immediately...
```

Blocking and Timeouts

- Socket I/O calls may block for several reasons:
  - Data input methods, such as read(), block if data is not available.
  - A **write**() on a socket may block if there is not sufficient space to buffer the transmitted data.
  - The **accept**() method of ServerSocket() and the Socket constructor both block until a connection has been established.
- **Long roundtrip times, high error rate connections, and slow (or deceased) servers may cause connection establishment to take a long time.**
- In all of these cases, the method returns only after the request has been satisfied.
  - A blocked method halts progress of the application, and makes the thread running it useless.

Limiting Blocking: Socket Constructor

- The **Socket** constructor attempts to establish a connection to the host and port supplied as arguments, blocking until either the connection is established or a system-imposed timeout occurs.
  - Unfortunately, the system-imposed timeout is long (on the order of **minutes**), and Java does not provide any means of shortening it.
  - To fix this, call the **parameterless constructor for Socket**, which returns an unconnected instance.
  - To establish a connection, call the **connect()** method on the newly constructed socket and specify both a remote endpoint and timeout (milliseconds).

Constructor Summary

Constructors

Modifier	Constructor and Description
	Socket() Creates an unconnected socket, with the system-default type of SocketImpl.
	Socket(**InetAddress** address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
	Socket(**InetAddress** host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(**InetAddress** address, int port, **InetAddress** localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
	Socket(**Proxy** proxy) Creates an unconnected socket, specifying the type of proxy, if any, that should be used regardless of any other settings.
protected	**Socket**(**SocketImpl** impl) Creates an unconnected Socket with a user-specified SocketImpl.
	Socket(**String** host, int port) Creates a stream socket and connects it to the specified port number on the named host.
	Socket(**String** host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(**String** host, int port, **InetAddress** localAddr, int localPort) Creates a socket and connects it to the specified remote host on the specified remote port.

Method Summary

Method Summary

Methods

Modifier and Type	Method and Description
void	**bind**(SocketAddress bindpoint) Binds the socket to a local address.
void	**close**() Closes this socket.
void	**connect**(SocketAddress endpoint) Connects this socket to the server.
void	**connect**(SocketAddress endpoint, int timeout) Connects this socket to the server with a specified timeout value.
SocketChannel	**getChannel**() Returns the unique SocketChannel object associated with this socket, if any.
InetAddress	**getInetAddress**() Returns the address to which the socket is connected.
InputStream	**getInputStream**() Returns an input stream for this socket.
boolean	**getKeepAlive**() Tests if SO_KEEPALIVE is enabled.
InetAddress	**getLocalAddress**() Gets the local address to which the socket is bound.
int	**getLocalPort**() Returns the local port number to which this socket is bound.
SocketAddress	**getLocalSocketAddress**() Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.

Creating Unconnected Sockets (Revisited)

- If you give no arguments to the Socket constructor, it has nowhere to connect to:
- You can connect later by passing a SocketAddress to one of the connect() methods:

```
try {
        Socket socket = new Socket();
        // fill in socket options
        SocketAddress address = new
                InetSocketAddress("www.syr.edu", 80);
        socket.connect(address, 1000); // time out in 1000 miliseconds
        // work with the sockets...
} catch (IOException ex) {
        System.err.println(ex);
}
```

Blocking and Timeouts

- Socket I/O calls may block for several reasons:
  - Data input methods, such as **read**(), block if data is not available.
  - A **write**() on a socket may block if there is not sufficient space to buffer the transmitted data.
  - The **accept**() method of ServerSocket() and the Socket constructor both block until a connection has been established.
- Long roundtrip times, high error rate connections, and slow (or deceased) servers may cause connection establishment to take a long time.
- In all of these cases, the method returns only after the request has been satisfied.
  - A blocked method halts progress of the application, and makes the thread running it useless.

Limiting Blocking

- A **write()** call blocks until the last byte written is copied into the local buffer.
  - If the available buffer space is smaller than the size of the write, some data must be successfully transferred to the other end of the connection before the call to write() will return.
  - The amount of time that a write() may block is ultimately controlled by the receiving application.
  - *Java currently does not provide any way to cause a write() to time out, nor can it be interrupted by another thread.*
  - Therefore, any protocol that sends a large enough amount of data over a Socket instance can block for an unbounded amount of time.

Add Timeouts to the Echo Server

- Suppose we want to add ‘timeouts’ to the Echo protocol, with a **timelimit** on the amount of time taken to service each client.
  - We can implement it in such a way that after **timelimit** milliseconds, the protocol instance is terminated.
  - The protocol instance keeps track of the amount of time remaining, and uses **setSoTimeout()** to ensure that no read() call blocks for longer than that time.
  - **TimelimitEchoProtocol.java** implements this approach
    - To use it with **TCPEchoServerExecutor.java**, simply change the second line of the body of the while loop to:

service.execute(new TimelimitEchoProtocol(clntSock, logger));

Textbook #1, pp 88-89

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

class TimelimitEchoProtocol implements Runnable {
  private static final int BUFSIZE = 32;  // Size (bytes) buffer
  private static final String TIMELIMIT = "10000";  // Default limit (ms)
  private static final String TIMELIMITPROP = "Timelimit";  // Thread property

  private static int timelimit;
  private Socket clntSock;
  private Logger logger;

  public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
    // Get the time limit from the System properties or take the default
    timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP,TIMELIMIT));
  }

  public static void handleEchoClient(Socket clntSock, Logger logger) {

    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();
      int recvMsgSize;                          // Size of received message
      int totalBytesEchoed = 0;                 // Bytes received from client
```

The **TimelimitEchoProtocol** class is similar to the **EchoProtocol** class, except that it attempts to bound the total time an echo connection can exist to **10** seconds.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

class TimelimitEchoProtocol implements Runnable {
  private static final int BUFSIZE = 32;  // Size (bytes) buffer
  private static final String TIMELIMIT = "10000"; // Default limit (ms)
  private static final String TIMELIMITPROP = "Timelimit";  // Thread property

  private static int timelimit;
  private Socket clntSock;
  private Logger logger;

  public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
    // Get the time limit from the System properties or take the default
    timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP,TIMELIMIT));
  }

  public static void handleEchoClient(Socket clntSock, Logger logger) {

    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();
      int recvMsgSize;                          // Size of received message
      int totalBytesEchoed = 0;                 // Bytes received from client
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

class TimelimitEchoProtocol implements Runnable {
  private static final int BUFSIZE = 32;  // Size (bytes) buffer
  private static final String TIMELIMIT = "10000";  // Default limit (ms)
  private static final String TIMELIMITPROP = "Timelimit";  // Thread property

  private static int timelimit;
  private Socket clntSock;
  private Logger logger;

  public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
    // Get the time limit from the System properties or take the default
    timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP,TIMELIMIT));
  }

  public static void handleEchoClient(Socket clntSock, Logger logger) {

    try {
      // Get the input and output I/O streams from socket
      InputStream in = clntSock.getInputStream();
      OutputStream out = clntSock.getOutputStream();
      int recvMsgSize;                          // Size of received message
      int totalBytesEchoed = 0;                 // Bytes received from client
```

getProperty

```
public static String getProperty(String key,
                                 String def)
```

Gets the system property indicated by the specified key.

First, if there is a security manager, its `checkPropertyAccess` method is called with the `key` as its argument.

If there is no current set of system properties, a set of system properties is first created and initialized in the same manner as for the `getProperties` method.

Parameters:

`key` - the name of the system property.

`def` - a default value.

Returns:

the string value of the system property, or the default value if there is no property with that key.

Throws:

`SecurityException` - if a security manager exists and its `checkPropertyAccess` method doesn't allow access to the specified system property.

`NullPointerException` - if `key` is `null`.

`IllegalArgumentException` - if `key` is empty.

See Also:

`setProperty(java.lang.String, java.lang.String)`, `SecurityManager.checkPropertyAccess(java.lang.String)`, `getProperties()`

setProperty

```
public static String setProperty(String key,
                                 String value)
```

```
public static void handleEchoClient(Socket clntSock, Logger logger) {

  try {
    // Get the input and output I/O streams from socket
    InputStream in = clntSock.getInputStream();
    OutputStream out = clntSock.getOutputStream();
    int recvMsgSize;                          // Size of received message
    int totalBytesEchoed = 0;                 // Bytes received from client
    byte[] echoBuffer = new byte[BUFSIZE];    // Receive buffer
    long endTime = System.currentTimeMillis() + timelimit;
    int timeBoundMillis = timelimit;

    clntSock.setSoTimeout(timeBoundMillis);
    // Receive until client closes connection, indicated by -1
    while ((timeBoundMillis > 0) &&     // catch zero values
           ((recvMsgSize = in.read(echoBuffer)) != -1)) {
      out.write(echoBuffer, 0, recvMsgSize);
      totalBytesEchoed += recvMsgSize;
      timeBoundMillis = (int) (endTime - System.currentTimeMillis()) ;
      clntSock.setSoTimeout(timeBoundMillis);
    }
    logger.info("Client " + clntSock.getRemoteSocketAddress() +
                ", echoed " + totalBytesEchoed + " bytes.");
  } catch (IOException ex) {
    logger.log(Level.WARNING, "Exception in echo protocol", ex);
  }
}

public void run() {
  handleEchoClient(this.clntSock, this.logger);
}
```

At the time the **handleEchoClient()** method is invoked, a deadline is computed using the current time and the time bound.

```
public static void handleEchoClient(Socket clntSock, Logger logger) {

  try {
    // Get the input and output I/O streams from socket
    InputStream in = clntSock.getInputStream();
    OutputStream out = clntSock.getOutputStream();
    int recvMsgSize;                          // Size of received message
    int totalBytesEchoed = 0;                 // Bytes received from client
    byte[] echoBuffer = new byte[BUFSIZE];    // Receive buffer
    long endTime = System.currentTimeMillis() + timelimit;
    int timeBoundMillis = timelimit;

    clntSock.setSoTimeout(timeBoundMillis);
    // Receive until client closes connection, indicated by -1
    while ((timeBoundMillis > 0) &&     // catch zero values
           ((recvMsgSize = in.read(echoBuffer)) != -1)) {
      out.write(echoBuffer, 0, recvMsgSize);
      totalBytesEchoed += recvMsgSize;
      timeBoundMillis = (int) (endTime - System.currentTimeMillis()) ;
      clntSock.setSoTimeout(timeBoundMillis);
    }
    logger.info("Client " + clntSock.getRemoteSocketAddress() +
                ", echoed " + totalBytesEchoed + " bytes.");
  } catch (IOException ex) {
    logger.log(Level.WARNING, "Exception in echo protocol", ex);
  }
}

public void run() {
  handleEchoClient(this.clntSock, this.logger);
}
```

After each **read**(), the time between the current time and the deadline is computed, and the socket timeout is set to the remaining time.

Blocking and Timeouts

- In the server (**TCPEchoServerExecutor.java**), you only need to make the following change to use the time-limit version:

```
while (true) {
    Socket clntSock = servSock.accept(); // Block waiting for connection
    // service.execute(new EchoProtocol(clntSock, logger));
    service.execute(new TimelimitEchoProtocol(clntSock, logger));
}
```

Buffer Size

❖Send and Receive Buffer Size

- ❖When a **Socket** is created, the operating system must allocate buffers to hold incoming and outgoing data.
- ❖The default size is 64K.
- ❖You can reset the size by using the following methods in the Socket class.

getReceiveBufferSize

For Socket

```
public int getReceiveBufferSize()
                           throws SocketException
```

Gets the value of the SO_RCVBUF option for this `Socket`, that is the buffer size used by the platform for input on this `Socket`.

Returns:

the value of the SO_RCVBUF option for this `Socket`.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.2

See Also:

`setReceiveBufferSize(int)`

setKeepAlive

```
public void setKeepAlive(boolean on)
                  throws SocketException
```

Enable/disable SO_KEEPALIVE.

Parameters:

`on` - whether or not to have socket keep alive turned on.

setReceiveBufferSize

For Socket

```
public void setReceiveBufferSize(int size)
                          throws SocketException
```

Sets the SO_RCVBUF option to the specified value for this `Socket`. The SO_RCVBUF option is used by the platform's networking code as a hint for the size to set the underlying network I/O buffers.

Increasing the receive buffer size can increase the performance of network I/O for high-volume connection, while decreasing it can help reduce the backlog of incoming data.

Because SO_RCVBUF is a hint, applications that want to verify what size the buffers were set to should call `getReceiveBufferSize()`.

The value of SO_RCVBUF is also used to set the TCP receive window that is advertized to the remote peer. Generally, the window size can be modified at any time when a socket is connected. However, if a receive window larger than 64K is required then this must be requested **before** the socket is connected to the remote peer. There are two cases to be aware of:

1. For sockets accepted from a ServerSocket, this must be done by calling `ServerSocket.setReceiveBufferSize(int)` before the ServerSocket is bound to a local address.
2. For client sockets, setReceiveBufferSize() must be called before connecting the socket to its remote peer.

Parameters:

`size` - the size to which to set the receive buffer size. This value must be greater than 0.

Throws:

`IllegalArgumentException` - if the value is 0 or is negative.

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.2

getSendBufferSize

For Socket

```
public int getSendBufferSize()
                    throws SocketException
```

Get value of the SO_SNDBUF option for this `Socket`, that is the buffer size used by the platform for output on this `Socket`.

Returns:

the value of the SO_SNDBUF option for this `Socket`.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.2

See Also:

`setSendBufferSize(int)`

setReceiveBufferSize

```
public void setReceiveBufferSize(int size)
                        throws SocketException
```

Sets the SO_RCVBUF option to the specified value for this `Socket`. The SO_RCVBUF option is used by the platform's networking code as a hint for the size to set the underlying network I/O buffers.

Increasing the receive buffer size can increase the performance of network I/O for high-volume connection, while decreasing it can help reduce the backlog of

setSendBufferSize

```
public void setSendBufferSize(int size)
                       throws SocketException
```

For Socket

Sets the SO_SNDBUF option to the specified value for this `Socket`. The SO_SNDBUF option is used by the platform's networking code as a hint for the size to set the underlying network I/O buffers.

Because SO_SNDBUF is a hint, applications that want to verify what size the buffers were set to should call `getSendBufferSize()`.

Parameters:

`size` - the size to which to set the send buffer size. This value must be greater than 0.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

`IllegalArgumentException` - if the value is 0 or is negative.

Since:

1.2

See Also:

`getSendBufferSize()`

getSendBufferSize

```
public int getSendBufferSize()
                       throws SocketException
```

Buffer Size

❖Send and Receive Buffer Size

- ❖You can also specify the receive buffer size on a **ServerSocket**; however, this actually sets the receive buffer size for new Socket instances created by accept().

getReceiveBufferSize

```
public int getReceiveBufferSize()
                           throws SocketException
```

For ServerSocket

Gets the value of the SO_RCVBUF option for this `ServerSocket`, that is the proposed buffer size that will be used for Sockets accepted from this `ServerSocket`.

Note, the value actually set in the accepted socket is determined by calling `Socket.getReceiveBufferSize()`.

Returns:

the value of the SO_RCVBUF option for this `Socket`.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.4

See Also:

`setReceiveBufferSize(int)`

setPerformancePreferences

```
public void setPerformancePreferences(int connectionTime,
                                       int latency,
                                       int bandwidth)
```

setReceiveBufferSize

```
public void setReceiveBufferSize(int size)
                          throws SocketException
```

For ServerSocket

Sets a default proposed value for the SO_RCVBUF option for sockets accepted from this `ServerSocket`. The value actually set in the accepted socket must be determined by calling `Socket.getReceiveBufferSize()` after the socket is returned by `accept()`.

The value of SO_RCVBUF is used both to set the size of the internal socket receive buffer, and to set the size of the TCP receive window that is advertized to the remote peer.

It is possible to change the value subsequently, by calling `Socket.setReceiveBufferSize(int)`. However, if the application wishes to allow a receive window larger than 64K bytes, as defined by RFC1323 then the proposed value must be set in the ServerSocket **before** it is bound to a local address. This implies, that the ServerSocket must be created with the no-argument constructor, then setReceiveBufferSize() must be called and lastly the ServerSocket is bound to an address by calling bind().

Failure to do this will not cause an error, and the buffer size may be set to the requested value but the TCP receive window in sockets accepted from this ServerSocket will be no larger than 64K bytes.

Parameters:

`size` - the size to which to set the receive buffer size. This value must be greater than 0.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

`IllegalArgumentException` - if the value is 0 or is negative.

Since:

1.4

See Also:

`getReceiveBufferSize()`

Keep-Alive

- **Keep-Alive**
  - If no data has been exchanged for a while, each endpoint may be wondering if the other is still around.
  - TCP provides a keep-alive mechanism where, after a certain time of inactivity, a probe message is sent to the other endpoint.
  - If the endpoint is alive and well, it sends an acknowledgment.
  - After a few retries without acknowledgment, the probe sender gives up and closes the socket, causing an exception on the next attempted I/O operation.
  - Note that the application only sees keep-alive working if the probes **fail**.

setKeepAlive

```
public void setKeepAlive(boolean on)
                  throws SocketException
```

For Socket

Enable/disable SO_KEEPALIVE.

Parameters:

`on` - whether or not to have socket keep alive turned on.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.3

See Also:

`getKeepAlive()`

getKeepAlive

```
public boolean getKeepAlive()
                    throws SocketException
```

Tests if SO_KEEPALIVE is enabled.

Returns:

a `boolean` indicating whether or not SO_KEEPALIVE is enabled.

getKeepAlive

For Socket

```
public boolean getKeepAlive()
                    throws SocketException
```

Tests if SO_KEEPALIVE is enabled.

Returns:

a `boolean` indicating whether or not SO_KEEPALIVE is enabled.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

1.3

See Also:

`setKeepAlive(boolean)`

setTrafficClass

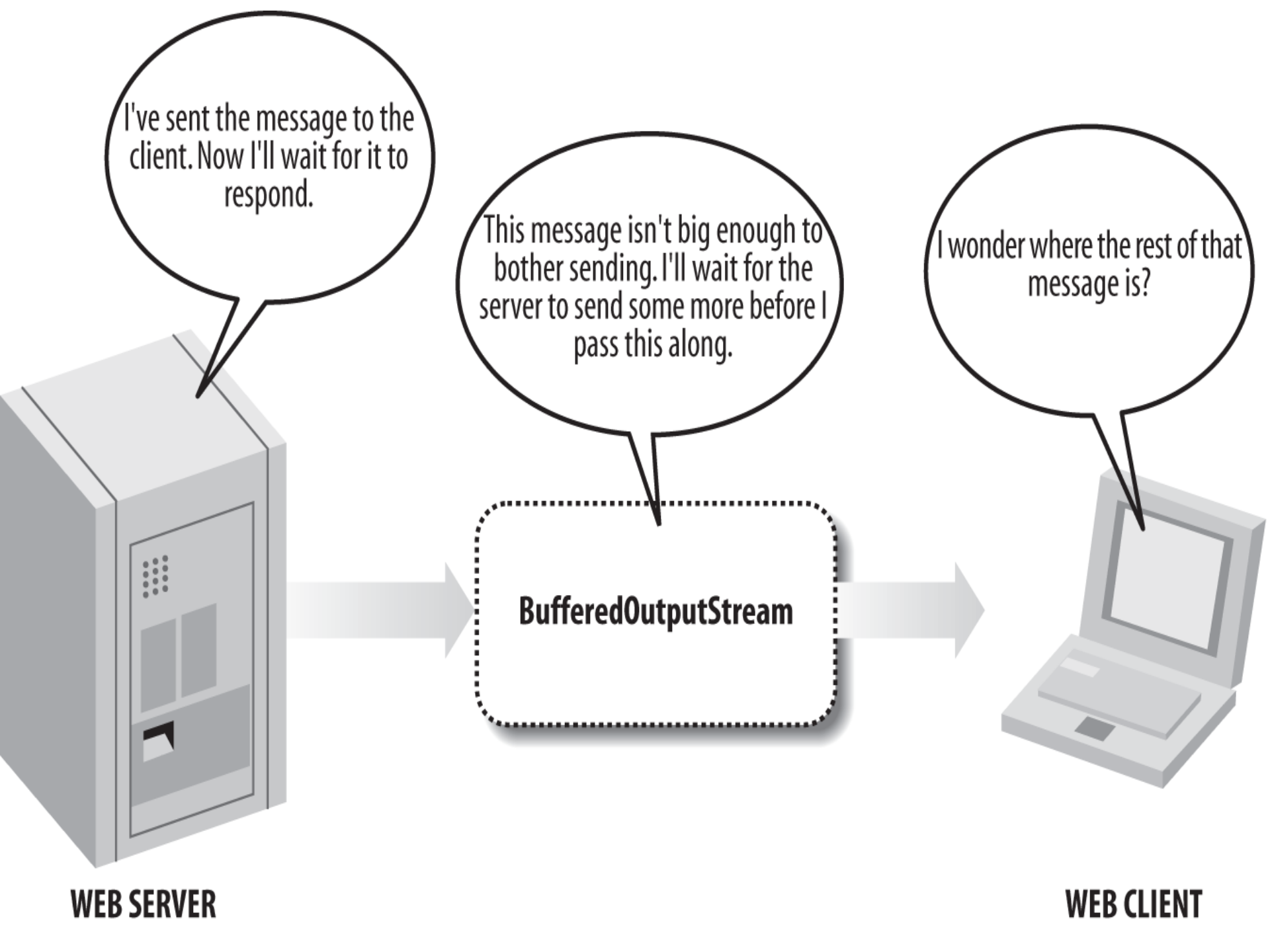
```
public void setTrafficClass(int tc)
                    throws SocketException
```

Sets traffic class or type-of-service octet in the IP header for packets sent from this Socket. As the underlying network implementation may ignore this value applications should consider it a hint.

The tc **must** be in the range `0 <= tc <= 255` or an IllegalArgumentException will be thrown.

Buffering Delays

- **Eliminating Buffering Delay**
  - TCP attempts to help you avoid sending small packets, which waste network resources. (See next slide)
  - It does this by buffering data until it has more to send.
  - While this is good for the network, your application may not be so tolerant of this buffering delay.
  - Fortunately, you can disable this behavior.

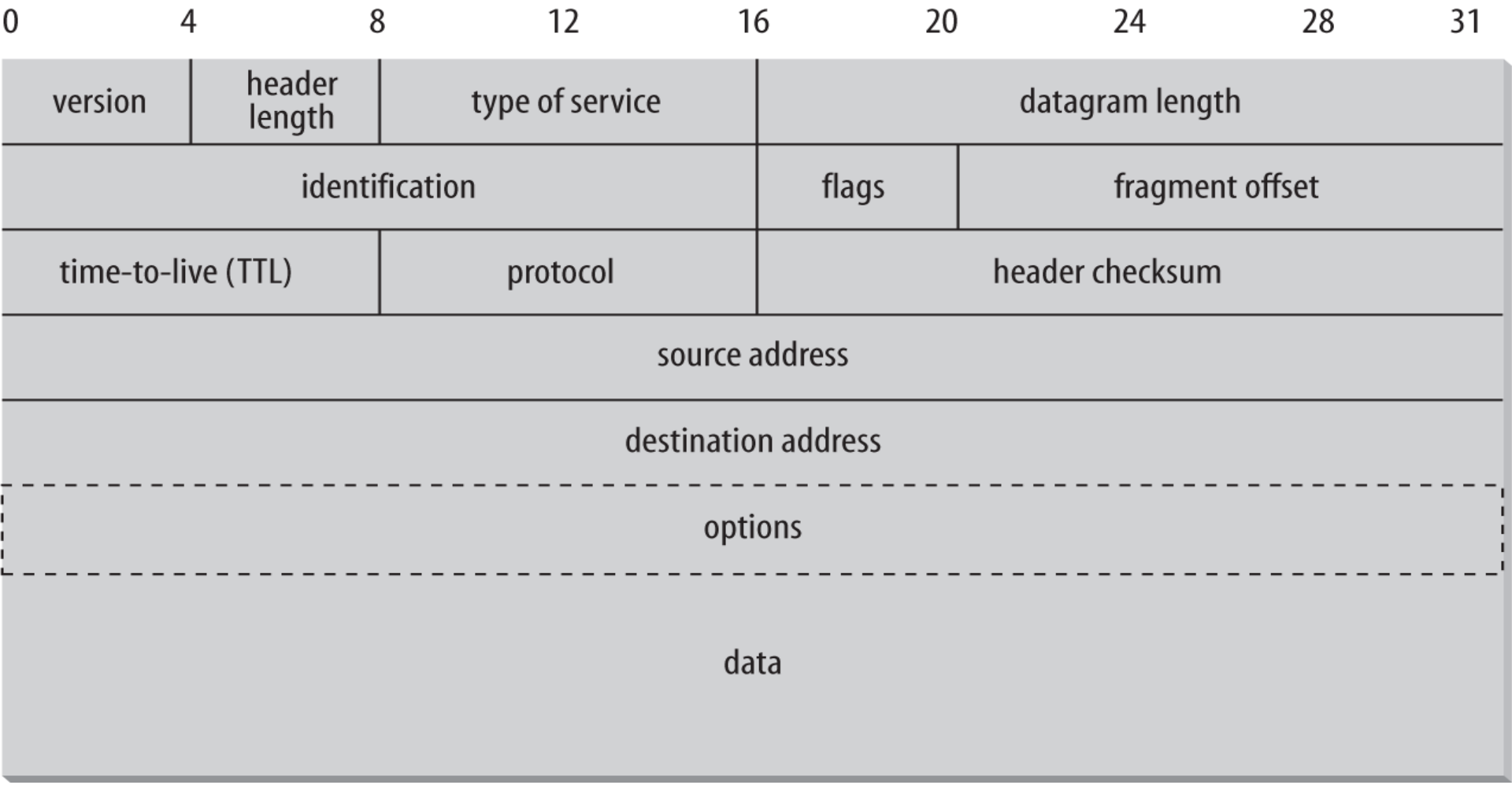

I've sent the message to the client. Now I'll wait for it to respond.
This message isn't big enough to bother sending. I'll wait for the server to send some more before I pass this along.
I wonder where the rest of that message is?
BufferedOutputStream
WEB SERVER
WEB CLIENT

Nagle's Algorithm

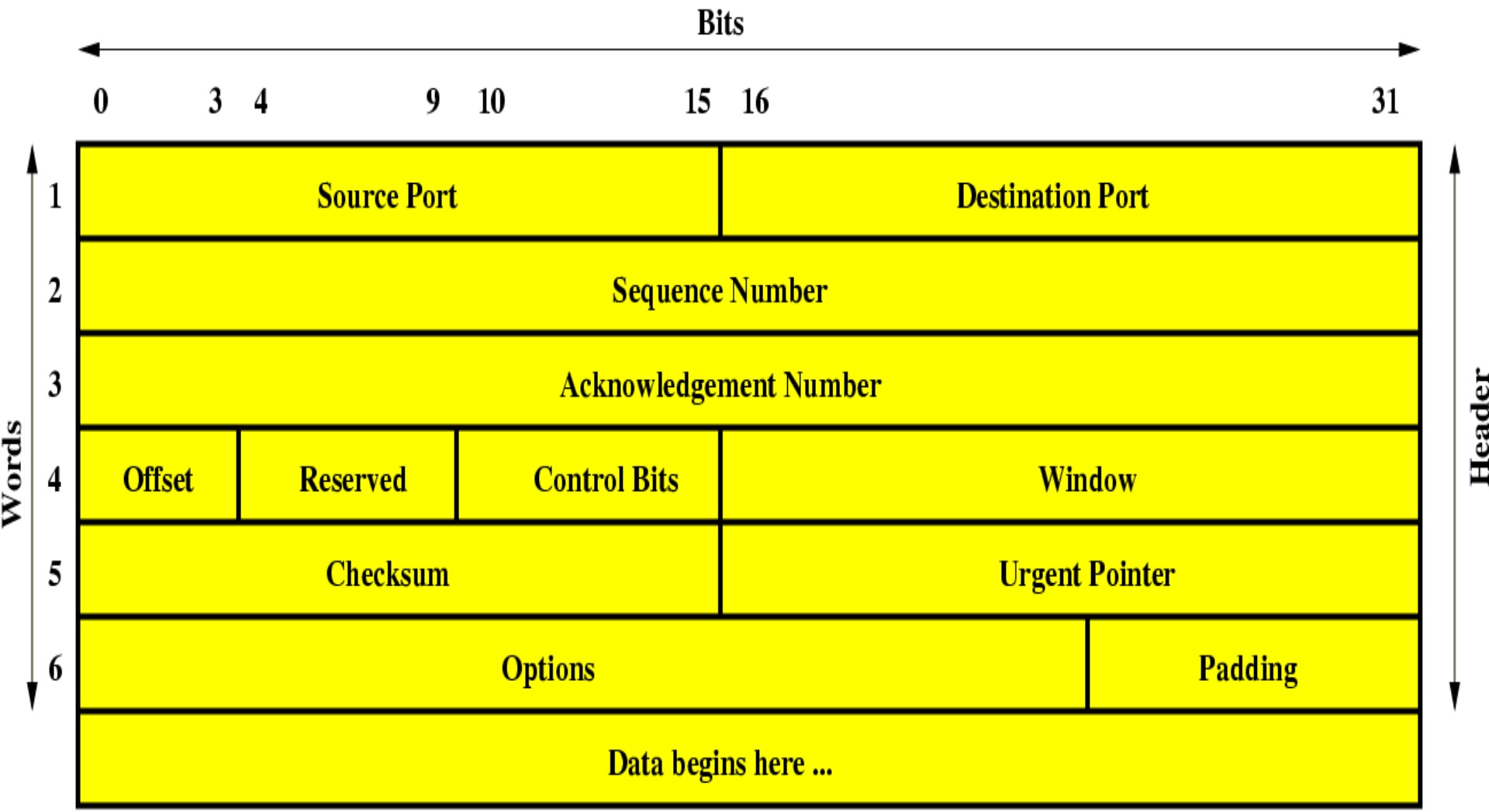
- **Nagle's algorithm**, named after John Nagle, is a means of improving the efficiency of TCP/IP networks by reducing the number of packets that need to be sent over the network.
- Nagle's document, **Congestion Control in IP/TCP Internetworks** (RFC 896) describes what he called the "**small packet problem**", where an application repeatedly emits data in small chunks, frequently only 1 byte in size.
  - Since TCP/IPv4 packets have a 40 byte header (20 bytes for TCP, 20 bytes for IPv4), this results in a 41 byte packet for 1 byte of useful information, a huge overhead.
- Over slow links, many such packets can be in transit at the same time, potentially leading to **congestion collapse**.
- Nagle's algorithm works by combining a number of small outgoing messages, and sending them all at once.
  - Specifically, as long as there is a sent packet for which the sender has received no acknowledgment, the sender should keep buffering its output until it has a full packet's worth of output. (Standard packet size: 1500 bytes)

Datagrams

- In both IPv4 and IPv6, data is sent across the internet layer in packets called datagrams:



TCP Packet Format



For Socket

setTcpNoDelay

```
public void setTcpNoDelay(boolean on)
                   throws SocketException
```

The **setTcpNoDelay()** method sets the elimination of buffering delay. A value of **true** means that buffering delay is disabled.

Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).

Parameters:

`on` - `true` to enable TCP_NODELAY, `false` to disable.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

JDK1.1

See Also:

`getTcpNoDelay()`

getTcpNoDelay

```
public boolean getTcpNoDelay()
                    throws SocketException
```

Tests if TCP_NODELAY is enabled.

Returns:

a `boolean` indicating whether or not TCP_NODELAY is enabled.

For Socket

getTcpNoDelay

```
public boolean getTcpNoDelay()
                      throws SocketException
```

The **getTcpNoDelay()** method gets the elimination of buffering delay. A value of **true** means that buffering delay is disabled.

Tests if TCP_NODELAY is enabled.

Returns:

a `boolean` indicating whether or not TCP_NODELAY is enabled.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error.

Since:

JDK1.1

See Also:

`setTcpNoDelay(boolean)`

setSoLinger

```
public void setSoLinger(boolean on,
                int linger)
                  throws SocketException
```

Enable/disable SO_LINGER with the specified linger time in seconds. The maximum timeout value is platform specific. The setting only affects socket close.

Parameters:

Closing Connections

- Network protocols are typically very specific about **who closes first**.
- In the echo protocol, Figure 4.1(a), the server dutifully echoes everything the client sends.
  - When the client is finished, it calls **close**().
  - After the server has received and echoed all of the data sent before the client's call to close(), its read operation returns a −1, indicating that the client is finished.
  - The server then calls **close()** on its socket.
  - The **close()** is a critical part of the protocol, because without it the server doesn't know when the client is finished sending characters to echo.

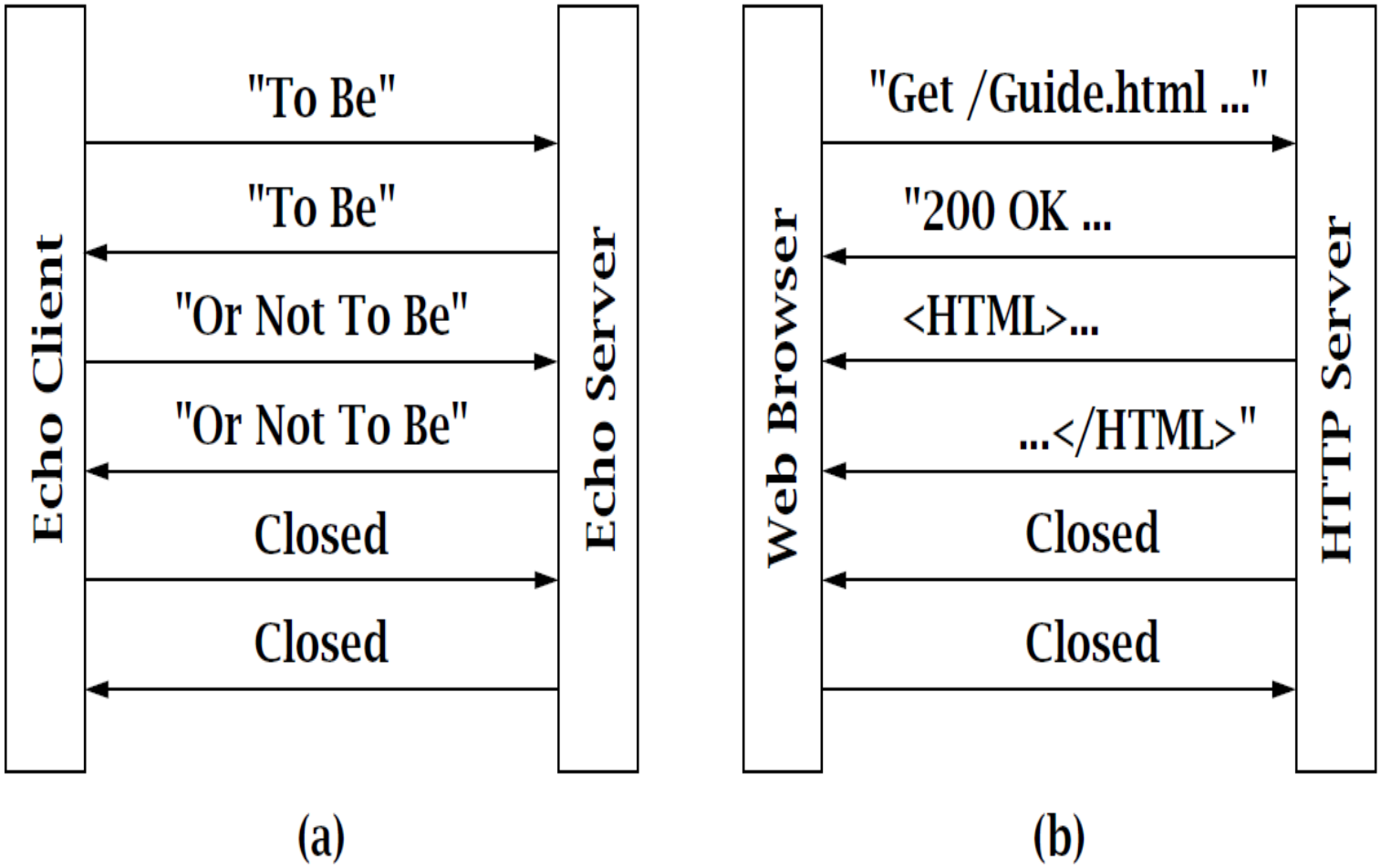


Figure 4.1: Echo (a) and HTTP (b) protocol termination.

Closing Connections

- In HTTP, Figure 4.1(b), it's the server that initiates the connection close.
  - The client sends a request ("GET") to the server, and the server responds by sending a header, followed by the requested file.
  - Since the client does not know the size of the file, the server must indicate the **end-of-file** by closing the socket.

Closing Connections

- Calling **close()** on a Socket terminates **both** directions (input and output) of data flow.
  - Once an endpoint (either client or server) closes the socket, it can no longer send or receive data.
    - Hence, **close()** can only be used to signal the other end when the caller is completely finished communicating.
- In the echo protocol, once the server receives the **close()** from the client, it immediately closes, because the client close() indicates that the communication is completed.
- HTTP works the same way, except that the server is the terminator.

Closing Connections

- Suppose you want build a **compression server** that takes a stream of bytes, compresses them, and sends the compressed stream back to the client. Which endpoint (client or server) should close the connection?
  - If the client sends an arbitrarily long stream of data to be compressed, the client should close the connection so that the server knows when the stream of bytes ends.

Closing Connections

- When should the client call close()?
  1. Immediately after it sends the last byte of data, or
  2. Wait until it receives all of the compressed data before it closes, as the echo protocol does.

  - If the client calls close() on the socket immediately after it sends the last byte of data, it will not be able to receive the last bytes of compressed data.
  - If you want the client to wait until it receives all of the compressed data before it closes, unfortunately, neither the server nor the client knows how many bytes to expect, so this will not work either.

Closing Connections

- What is needed is a way to tell the other end of the connection "I am through sending," without losing the ability to receive.
- Fortunately the **Socket** class in Java provides a way for you to do this:
  - The **shutdownInput()** method:
    - After shutdownInput(), the socket's input stream can no longer be used.
    - Any undelivered data is silently discarded.
    - Any attempt to read from the socket's input stream will return −1.
  - The **shutdownOutput()** method
    - After it's called, no more data may be sent on the socket's output stream.
    - Attempts to write to the output stream throw an IOException.
    - Any data written before the call to shutdownOutput() may be read by the remote socket.
    - After that, a read on the input stream of the remote socket will return −1.
  - An application calling **shutdownOutput** can continue to read from the socket and, similarly, data can be written after calling **shutdownInput**.

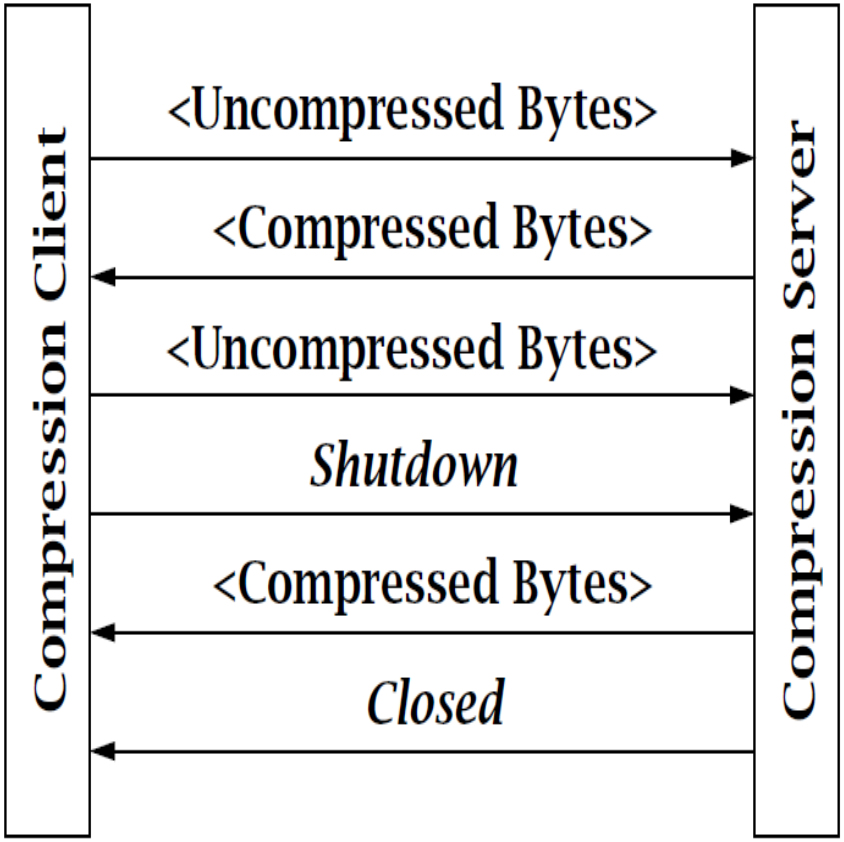


Figure 4.2: Compression protocol termination.

CompressProtocol.java

Textbook #1, pp. 105-106

```
0  import java.net.Socket;
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.io.OutputStream;
4  import java.util.zip.GZIPOutputStream;
5  import java.util.logging.Logger;
6  import java.util.logging.Level;
7
8  public class CompressProtocol implements Runnable {
9
10   public static final int BUFSIZE = 1024;   // Size of receive buffer
11   private Socket clntSock;
12   private Logger logger;
13
14   public CompressProtocol(Socket clntSock, Logger logger) {
15     this.clntSock = clntSock;
16     this.logger = logger;
17   }
18
19   public static void handleCompressClient(Socket clntSock, Logger logger) {
20     try {
21       // Get the input and output streams from socket
22       InputStream in = clntSock.getInputStream();
23       GZIPOutputStream out = new GZIPOutputStream(clntSock.getOutputStream());
24
25       byte[] buffer = new byte[BUFSIZE];   // Allocate read/write buffer
26       int bytesRead;                       // Number of bytes read
```

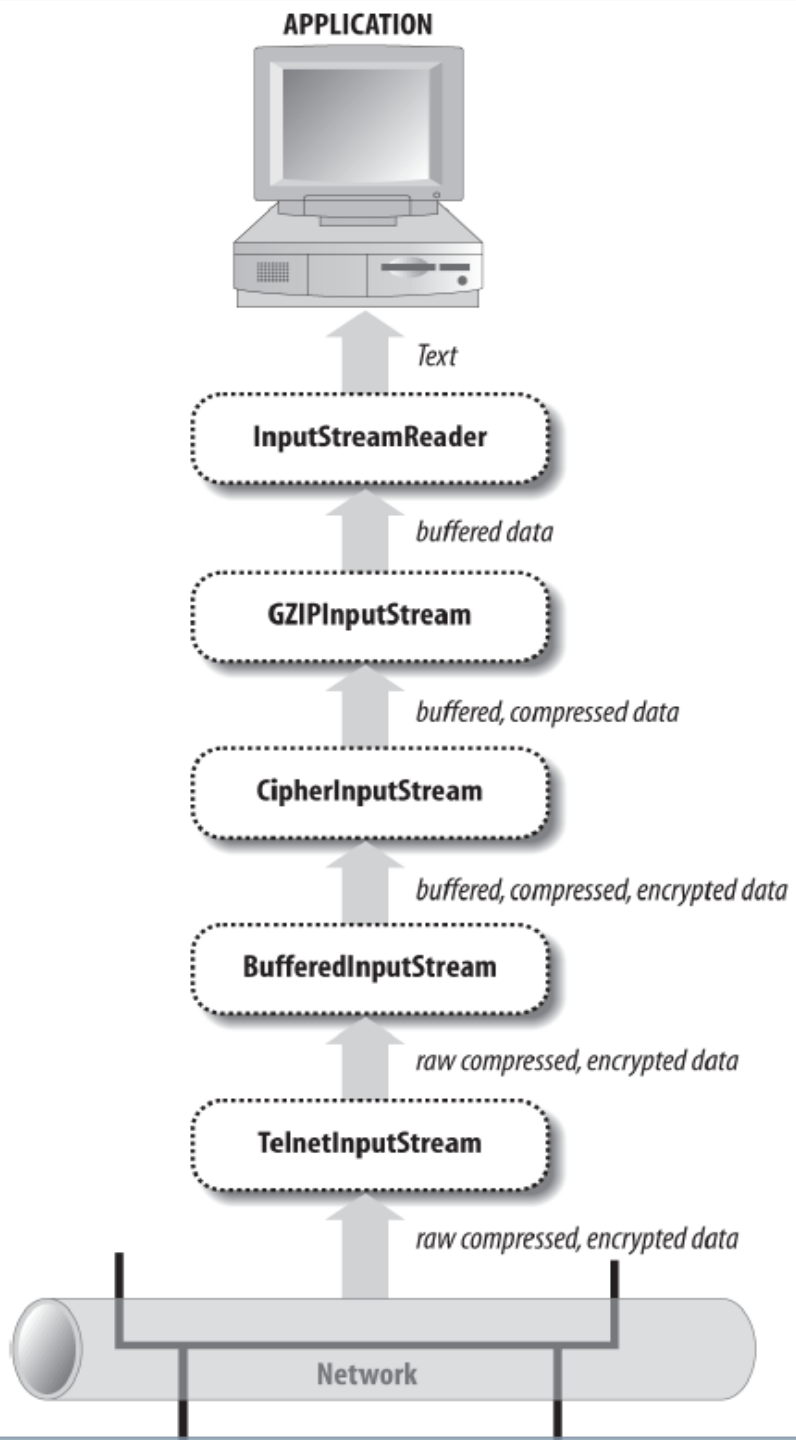
Variables and constructors

Given a socket connected to the compression client, **read the uncompressed bytes** from the client and **write the compressed bytes** back.

The socket's output stream is wrapped in a **GZIPOutputStream**.

I/O Class	Function
Buffered[Input/Output]Stream	Performs buffering for I/O optimization.
Checked[Input/Output]Stream	Maintains a checksum on data.
Cipher[Input/Output]Stream	Encrypt/Decrypt data.
Data[Input/Output]Stream	Handles read/write for primitive date types.
Digest[Input/Output]Stream	Maintains a digest on data.
GZIP[Input/Output]Stream	De/compresses a byte stream in GZIP format.
Object[Input/Output]Stream	Handles read/write objects and primitive data types.
PushbackInputStream	Allows a byte or bytes to be "unread."
PrintOutputStream	Prints string representation of data type.
Zip[Input/Output]Stream	De/compresses a byte stream in ZIP format.

Table 3.1: Java I/O Classes


APPLICATION
Text
InputStreamReader
buffered data
GZIPInputStream
buffered, compressed data
CipherInputStream
buffered, compressed, encrypted data
BufferedInputStream
raw compressed, encrypted data
TelnetInputStream
raw compressed, encrypted data
Network

```
27        // Receive until client closes connection, indicated by -1 return
28        while ((bytesRead = in.read(buffer)) != -1)
29          out.write(buffer, 0, bytesRead);
30        out.finish();      // Flush bytes from GZIPOutputStream
31
32        logger.info("Client " + clntSock.getRemoteSocketAddress() + " finished");
33      } catch (IOException ex) {
34        logger.log(Level.WARNING, "Exception in echo protocol", ex);
35      }
36
37      try {  // Close socket
38        clntSock.close();
39      } catch (IOException e) {
40        logger.info("Exception = " +  e.getMessage());
41      }
42    }
43
44    public void run() {
45      handleCompressClient(this.clntSock, this.logger);
46    }
47  }
```

The sequence of bytes written to this stream is compressed, using the GZIP algorithm

Calling **finish()** on the GZIPOutputStream is necessary to flush any bytes that may be buffered by the compression algorithm

The **run()** method simply calls the **handleCompressClient()** method

CompressProtocol.java

CompressClient.java

Textbook #1, pp. 103-104

```
 0  import java.net.Socket;
 1  import java.io.IOException;
 2  import java.io.InputStream;
 3  import java.io.OutputStream;
 4  import java.io.FileInputStream;
 5  import java.io.FileOutputStream;
 6
 7  /* WARNING: this code can deadlock if a large file (more than a few
 8   * 10's of thousands of bytes) is sent.
 9   */
10
11  public class CompressClient {
12
13    public static final int BUFSIZE = 256;  // Size of read buffer
14
15    public static void main(String[] args) throws IOException {
16
17      if (args.length != 3) { // Test for correct # of args
18        throw new IllegalArgumentException("Parameter(s): <Server> <Port> <File>");
19      }
20
21      String server = args[0];                // Server name or IP address
22      int port = Integer.parseInt(args[1]);  // Server port
23      String filename = args[2];              // File to read data from
24
25      // Open input and output file (named input.gz)
26      FileInputStream fileIn = new FileInputStream(filename);
27      FileOutputStream fileOut = new FileOutputStream(filename + ".gz");
28
29      // Create socket connected to server on specified port
30      Socket sock = new Socket(server, port);
31
32      // Send uncompressed byte stream to server
```

Application setup and parameter parsing

Create socket and open files

```
33      sendBytes(sock, fileIn);
34
35      // Receive compressed byte stream from server
36      InputStream sockIn = sock.getInputStream();
37      int bytesRead;                          // Number of bytes read
38      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
39      while ((bytesRead = sockIn.read(buffer)) != -1) {
40        fileOut.write(buffer, 0, bytesRead);
41        System.out.print("R");   // Reading progress indicator
42      }
43      System.out.println();      // End progress indicator line
44
45      sock.close();     // Close the socket and its streams
46      fileIn.close();   // Close file streams
47      fileOut.close();
48    }
49
50    private static void sendBytes(Socket sock, InputStream fileIn)
51        throws IOException {
52      OutputStream sockOut = sock.getOutputStream();
53      int bytesRead;                          // Number of bytes read
54      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
55      while ((bytesRead = fileIn.read(buffer)) != -1) {
56        sockOut.write(buffer, 0, bytesRead);
57        System.out.print("W");   // Writing progress indicator
58      }
59      sock.shutdownOutput();     // Finished sending
60    }
61  }
```

Invoke **sendBytes()** to transmit bytes

Given a **socket** connected to a compression server and the **file input stream**, read all of the uncompressed bytes from the file and write them to the socket output stream.

```
33      sendBytes(sock, fileIn);
34
35      // Receive compressed byte stream from server
36      InputStream sockIn = sock.getInputStream();
37      int bytesRead;                          // Number of bytes read
38      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
39      while ((bytesRead = sockIn.read(buffer)) != -1) {
40        fileOut.write(buffer, 0, bytesRead);
41        System.out.print("R");   // Reading progress indicator
42      }
43      System.out.println();      // End progress indicator line
44
45      sock.close();     // Close the socket and its streams
46      fileIn.close();   // Close file streams
47      fileOut.close();
48    }
49
50    private static void sendBytes(Socket sock, InputStream fileIn)
51        throws IOException {
52      OutputStream sockOut = sock.getOutputStream();
53      int bytesRead;                          // Number of bytes read
54      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
55      while ((bytesRead = fileIn.read(buffer)) != -1) {
56        sockOut.write(buffer, 0, bytesRead);
57        System.out.print("W");   // Writing progress indicator
58      }
59      sock.shutdownOutput();     // Finished sending
60    }
61  }
```

The while loop reads from the file input stream and repeats the bytes to the socket output stream until **end-of-file**, indicated by −1 from read(). Each write is indicated by a “W” printed to the console.

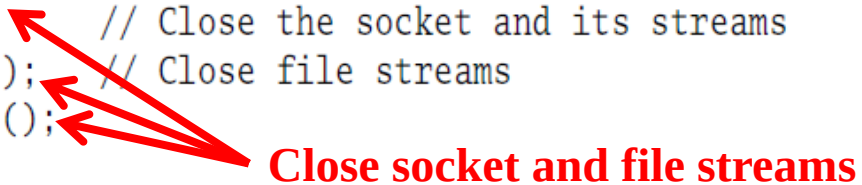
```
33      sendBytes(sock, fileIn);
34
35      // Receive compressed byte stream from server
36      InputStream sockIn = sock.getInputStream();
37      int bytesRead;                          // Number of bytes read
38      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
39      while ((bytesRead = sockIn.read(buffer)) != -1) {
40        fileOut.write(buffer, 0, bytesRead);
41        System.out.print("R");   // Reading progress indicator
42      }
43      System.out.println();      // End progress indicator line
44
45      sock.close();     // Close the socket and its streams
46      fileIn.close();   // Close file streams
47      fileOut.close();
48    }
49
50    private static void sendBytes(Socket sock, InputStream fileIn)
51        throws IOException {
52      OutputStream sockOut = sock.getOutputStream();
53      int bytesRead;                          // Number of bytes read
54      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
55      while ((bytesRead = fileIn.read(buffer)) != -1) {
56        sockOut.write(buffer, 0, bytesRead);
57        System.out.print("W");   // Writing progress indicator
58      }
59      sock.shutdownOutput();      // Finished sending
60    }
61  }
```

After reading and sending all of the bytes from the input file, **shut down the output stream**, notifying the server that the client is finished sending. The close will cause a −1 return from read() on the server.

```
33      sendBytes(sock, fileIn);
34
35      // Receive compressed byte stream from server
36      InputStream sockIn = sock.getInputStream();
37      int bytesRead;                          // Number of bytes read
38      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
39      while ((bytesRead = sockIn.read(buffer)) != -1) {
40        fileOut.write(buffer, 0, bytesRead);
41        System.out.print("R");   // Reading progress indicator
42      }
43      System.out.println();      // End progress indicator line
44
45      sock.close();     // Close the socket and its streams
46      fileIn.close();   // Close file streams
47      fileOut.close();
48    }
49
50    private static void sendBytes(Socket sock, InputStream fileIn)
51        throws IOException {
52      OutputStream sockOut = sock.getOutputStream();
53      int bytesRead;                          // Number of bytes read
54      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
55      while ((bytesRead = fileIn.read(buffer)) != -1) {
56        sockOut.write(buffer, 0, bytesRead);
57        System.out.print("W");   // Writing progress indicator
58      }
59      sock.shutdownOutput();     // Finished sending
60    }
61  }
```

The while loop receives the compressed data stream and writes the bytes to the output file until an end-of-stream is signaled by a −1 from read().

```
33      sendBytes(sock, fileIn);
34
35      // Receive compressed byte stream from server
36      InputStream sockIn = sock.getInputStream();
37      int bytesRead;                          // Number of bytes read
38      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
39      while ((bytesRead = sockIn.read(buffer)) != -1) {
40        fileOut.write(buffer, 0, bytesRead);
41        System.out.print("R");   // Reading progress indicator
42      }
43      System.out.println();      // End progress indicator line
44
45      sock.close();      // Close the socket and its streams
46      fileIn.close();    // Close file streams
47      fileOut.close();
48    }
49
50    private static void sendBytes(Socket sock, InputStream fileIn)
51        throws IOException {
52      OutputStream sockOut = sock.getOutputStream();
53      int bytesRead;                          // Number of bytes read
54      byte[] buffer = new byte[BUFSIZE];  // Byte buffer
55      while ((bytesRead = fileIn.read(buffer)) != -1) {
56        sockOut.write(buffer, 0, bytesRead);
57        System.out.print("W");   // Writing progress indicator
58      }
59      sock.shutdownOutput();     // Finished sending
60    }
61  }
```



Reading Assignments

Required:

- Textbook #1: **TCP/IP Sockets in Java**, Chapter 4: Beyond the Basics. You may skip Section 4.3 (Multicast), UDP-related topics in Section 4.4 & Section 4.6 Applets.

Recommended:

- Textbook #2: **Java Network Programming**, Chapter 3 Threads.