UNIT IV

UNIT IV

Multithreading: Fundamentals, Thread class, Runnable interface, Creating multiple threads, Life cycle of thread, Thread priorities, Synchronization, Thread communication, Suspending, Resuming and Stopping threads.

Applets: Basics, skeleton, Initialization and termination, Repainting, Status window, Passing parameters.

Networking: Basics, Networking classes and interfaces, InetAddress, Inet4Address and Inet6Address, TCP/IP Client Sockets, URL, URLConnection, HttpURLConnection, The URI class, Cookies, TCP/IP Server sockets, Datagrams.

MULTITHREADING: FUNDAMENTALS

- ✓ There are two distinct types of multitasking: **process-based** and **thread-based**.
- ✓ A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- ✓ In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
- ✓ Multitasking threads require less overhead than multitasking processes.
- ✓ Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum.
- ✓ Threads exist in several states. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface

- ✓ Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- ✓ **Thread** encapsulates a thread of execution.
- ✓ To create a new thread, the program will either extend **Thread** or implement the **Runnable** interface.
- ✓ The **Thread** class defines several methods that help manage threads.

The Main Thread

- ✓ When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of the program, because it is the one that is executed when program begins. The main thread is important for two reasons:
  - It is the thread from which other "child" threads will be spawned.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- ✓ Although the main thread is created automatically when the program is started, it can be controlled through a **Thread** object. To do so, we must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

CREATING A THREAD

- ✓ We create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:
  - Can implement the **Runnable** interface.
  - can extend the **Thread** class, itself.

Implementing Runnable

- ✓ The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. We can construct a thread on any object that implements **Runnable**.
- ✓ To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:
  public void run( )
- ✓ Inside **run( )**, we will define the code that constitutes the new thread. The thread will end when **run( )** returns.
- ✓ After creating a class that implements **Runnable**, instantiate an object of type **Thread** from within that class. **Thread** defines several constructors.
  Thread(Runnable *threadOb*)
- ✓ In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.
- ✓ After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**.
- ✓ The **start( )** method is shown here:
  void start( )
- ✓ Here is an example that creates a new thread and starts it running:

```
// Create a thread by implementing Runnable.
class MyThread implements Runnable {
        String thrdName;
                MyThread(String name) {
                thrdName = name;
        }
        // Entry point of thread.
        public void run() {
                System.out.println(thrdName + " starting.");
                try {
                        for(int count=0; count < 10; count++) {
                                Thread.sleep(400);
                                System.out.println("In " + thrdName +", count is " + count);
                        }
                }
                catch(InterruptedException exc) {
                        System.out.println(thrdName + " interrupted.");
                }
                System.out.println(thrdName + " terminating.");
        }
}
class UseThreads {
        public static void main(String[] args) {
                System.out.println("Main thread starting.");

                // First, construct a MyThread object.
                MyThread mt = new MyThread("Child #1");

                // Next, construct a thread from that object.
                Thread newThrd = new Thread(mt);
                // Finally, start execution of the thread.
                newThrd.start();
                        for(int i=0; i < 50; i++) {
                        System.out.print(.);
                        try {
                                Thread.sleep(100);
                        }
```

```
                    catch(InterruptedException exc) {
                        System.out.println("Main thread interrupted.");
                    }
                }
                System.out.println("Main thread ending.");
            }
        }
```

Simple Improvements:

- ✓ It is possible to have a thread begin execution as soon as it is created.
- ✓ It is possible to give the name of thread when it is created.
- ✓ For this use the following version of **Thread** constructor.
  Thread(Runnable threadOb, String name)
- ✓ Name of the thread can be obtained by calling **getName()** defined by **Thread**.
  final String getName()
- ✓ Name of the thread can be set by using setName()
  final void setName(String threadName)

```
// Improved MyThread.
class MyThread implements Runnable {
    Thread thrd;
    // Construct a new thread.
    MyThread(String name) {
        thrd = new Thread(this, name);
        thrd.start(); // start the thread
    }
    // Begin execution of new thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrd.getName() +", count is " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " terminating.");
    }
}

class UseThreadsImproved {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");
        MyThread mt = new MyThread("Child #1");
        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }
        System.out.println("Main thread ending.");
    }
}
```

CREATING MULTIPLE THREADS:

✓ It is possible to create many threads as it needs.

// Create multiple threads.

```
class MyThread implements Runnable {
        Thread thrd;

        // Construct a new thread.
        MyThread(String name) {
                thrd = new Thread(this, name);
                thrd.start(); // start the thread
        }

        // Begin execution of new thread.
        public void run() {
                System.out.println(thrd.getName() + " starting.");

                try {
                        for(int count=0; count < 10; count++) {
                                Thread.sleep(400);
                                System.out.println("In " + thrd.getName() +", count is " + count);
                        }
                }
                catch(InterruptedException exc) {
                        System.out.println(thrd.getName() + " interrupted.");
                }
                System.out.println(thrd.getName() + " terminating.");
        }
}

class MoreThreads {

        public static void main(String[] args) {
                System.out.println("Main thread starting.");

                MyThread mt1 = new MyThread("Child #1");
                MyThread mt2 = new MyThread("Child #2");
                MyThread mt3 = new MyThread("Child #3");

                for(int i=0; i < 50; i++) {
                        System.out.print(".");

                        try {
                                Thread.sleep(100);
                        }
                        catch(InterruptedException exc) {
                                System.out.println("Main thread interrupted.");
                        }
                }

                System.out.println("Main thread ending.");
        }
}
```

DETERMINING WHEN A THREAD ENDS:

- ✓ Often the main thread is to finish last. This is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread.
- ✓ **Thread** provides two means by which we can determine if a thread has ended.
- ✓ First, we can call **isAlive( )** on the thread. Its general form is:
  final boolean isAlive( )
- ✓ The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

// Use isAlive().

```
class MoreThreads {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        do {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        } while (mt1.thrd.isAlive() ||mt2.thrd.isAlive() ||mt3.thrd.isAlive());

        System.out.println("Main thread ending.");
    }
}
```

- ✓ The method that will more commonly use to wait for a thread to finish is called **join( )**, shown here:
  final void join( ) throws InterruptedException
- ✓ This method waits until the thread on which it is called terminates. Its name comes from
- ✓ The concept of the calling thread waiting until the specified thread *joins* it.

// Use join().

```
class JoinThreads {
    public static void main(String[] args) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");

        try {
            mt1.thrd.join();
            System.out.println("Child #1 joined.");
            mt2.thrd.join();
            System.out.println("Child #2 joined.");
            mt3.thrd.join();
            System.out.println("Child #3 joined.");
        }
        catch(InterruptedException exc) {
            System.out.println("Main thread interrupted. ");
        }
        System.out.println("Main thread ending.");
    }
}
```

THREAD PRIORITIES:

- ✓ Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- ✓ Thread priorities are integers that specify the relative priority of one thread to another.
- ✓ A higher priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*.
- ✓ The rules that determine when a context switch takes place are simple:
  - *A thread can voluntarily relinquish control*.
  - *A thread can be preempted by a higher-priority thread*.
- ✓ We can change a thread's priority by calling setPriority(), which is a member of a **Thread**.
  final void setPriority(int level)
- ✓ Here level specifies the new priority setting for the calling thread.
- ✓ 3 constants defiend in Thread class:
  - o public static int MIN_PRIORITY
  - o public static int NORM_PRIORITY
  - o public static int MAX_PRIORITY
- ✓ Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
- ✓ We can obtain the current priority setting by calling the getPriority() method of **Thread.**
  final int getPriority()

SYNCHRONIZATION:

- ✓ When using multiple threads, it is sometimes necessary to coordinate the activities of two or more. The process by which this is achieved is called **synchronization**.
- ✓ Key to synchronization in Java is the concept of the *monitor*, which controls the access to an object. A monitor works by implementing the concept of a lock.
- ✓ When an object is locked by one thread, access to the object by another thread is restricted.
- ✓ Synchronization is supported by the keyword synchronized.

USING SYNCHRONIZED METHODS

- ✓ Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- ✓ When the method is called, the calling thread enters the object's monitor, which then locks the object.
- ✓ While locked no other thread can enter the method on that object. When the thread returns from the method, the monitor unlocks the object, allowing it to be used by the next thread.

```
// Use synchronize to control access.
class SumArray {
    private int sum;
    synchronized int sumArray(int[] nums) {
            sum = 0; // reset sum
            for(int i=0; i<nums.length; i++) {
                    sum += nums[i];
                    System.out.println("Running total for " +Thread.currentThread().getName() +
                                    " is " + sum);
                    try {
                            Thread.sleep(10); // allow task-switch
                    }catch(InterruptedException exc) {
                            System.out.println("Thread interrupted.");
                    }
            }
            return sum;
    }
}
    class MyThread implements Runnable {
        Thread thrd;
        static SumArray sa = new SumArray();
        int[] a;
        int answer;
```

```
        // Construct a new thread.
        MyThread(String name, int[] nums) {
                thrd = new Thread(this, name);
                a = nums;
                thrd.start(); // start the thread
        }
        // Begin execution of new thread.
        public void run() {
                int sum;

                System.out.println(thrd.getName() + " starting.");

                answer = sa.sumArray(a);
                System.out.println("Sum for " + thrd.getName() +" is " + answer);

                System.out.println(thrd.getName() + " terminating.");
        }
}
class Sync {
        public static void main(String[] args) {
                int[] a = {1, 2, 3, 4, 5};

                MyThread mt1 = new MyThread("Child #1", a);
                MyThread mt2 = new MyThread("Child #2", a);

                try {
                        mt1.thrd.join();
                        mt2.thrd.join();
                }
                catch(InterruptedException exc) {
                        System.out.println("Main thread interrupted.");
                }
        }
}
```

O/P:

```
Child #1 starting.
Child #2 starting.
Running total for Child #1 is 1
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Running total for Child #2 is 1
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

If we remove synchronized from the declaration of sumArray(), then it is no longer synchronized and any number of threads may execute it concurrently.
Output is as follows when method is not synchronized.

```
Child #1 starting.
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 1
```

```
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #1 is 8
Running total for Child #2 is 11
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #1 is 29
Running total for Child #2 is 29
Sum for Child #2 is 29
Sum for Child #1 is 29
Child #2 terminating.
Child #1 terminating.
```

THE synchronized STATEMENT

- ✓ Creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- ✓ It is not possible to add synchronized to the appropriate methods within class that was created by a third party, and do not have access to the source code.
- ✓ The solution to this problem is quite easy: Simply put calls to the methods defined by this class inside a **synchronized** block.
- ✓ This is the general form of the **synchronized** statement:

```
synchronized(objectref) {
        // statements to be synchronized
}
```

- ✓ Here, *objectref is* a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

```
class SumArray {
    private int sum;
    int sumArray(int[] nums) {
        sum = 0; // reset sum
        for(int i=0; i<nums.length; i++) {
            sum += nums[i];
            System.out.println("Running total for " +Thread.currentThread().getName() +
                    " is " + sum);
            try {
                Thread.sleep(10); // allow task-switch
            }catch(InterruptedException exc) {
                System.out.println("Thread interrupted.");
            }
        }
        return sum;
    }
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int[] a;
    int answer;
    // Construct a new thread.
    MyThread(String name, int[] nums) {
        thrd = new Thread(this, name);
        a = nums;
        thrd.start(); // start the thread
    }

    // Begin execution of new thread.
    public void run() {

        System.out.println(thrd.getName() + " starting.");
```

```
                / / synchronize calls to sumArray()
                synchronized(sa) {
                        answer = sa.sumArray(a);
                }
                System.out.println("Sum for " + thrd.getName() +" is " + answer);

                System.out.println(thrd.getName() + " terminating.");
        }
    }

    class Sync {
        public static void main(String[] args) {
                int[] a = {1, 2, 3, 4, 5};

                MyThread mt1 = new MyThread("Child #1", a);
                MyThread mt2 = new MyThread("Child #2", a);

                try {
                        mt1.thrd.join();
                        mt2.thrd.join();
                }
                catch(InterruptedException exc) {
                        System.out.println("Main thread interrupted.");
                }
        }
    }
```

THREAD COMMUNICATION USING notify(),wait() and notifyAll()

- ✓ Consider the following situation. One thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.
- ✓ Java supports interthread communication mechanism with the **wait( )**, **notify( )**, and **notifyAll( )** methods.
- ✓ These methods are implemented as **final** methods in **Object**.
- ✓ All three methods can be called onlyfrom within a **synchronized** context.
- ✓ The rules for using these methods are actually quite simple:
  • **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
  • **notify( )** wakes up a thread that called **wait( )** on the same object.
  • **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

- ✓ These methods are declared within **Object**, as shown here:
  final void wait( ) throws InterruptedException
  final void notify( )
  final void notify All( )
- ✓ Additional forms of wait( ) exist that allow you to specify a period of time to wait.
- ✓ Consider the following sample program that implements a simple form of the producer/ consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PCFixed**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;
    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            System.out.println("Got: " + n);
            valueSet = false;
            notify();
            return n;
        }
    synchronized void put(int n) {
        if(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
            this.n = n;
            valueSet = true;
            System.out.println("Put: " + n);
            notify();
    }
}
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }    }
```

O/P:

```
Put: 0
Press Control-C to stop.
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
Put: 6
Got: 6
Put: 7
Got: 7
Put: 8
Got: 8
Put: 9
Got: 9
Put: 10
Got: 10
```

Suspending, Resuming, and Stopping Threads

- ✓ Sometimes, suspending execution of a thread is useful.
- ✓ The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2.
- ✓ Prior to Java 2, a program used **suspend ( ), resume ( )** and **stop ()** which are methods defined by **Thread**, to pause, restart and stop the execution of a thread.
- ✓ The **suspend( )** method of the **Thread** class was deprecated by Java 2. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.
- ✓ The **resume( )** method is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart.
- ✓ The **stop( )** method of the **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop( )** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.
- ✓ In later versions of Java, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.
- ✓ Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate.

```
// Suspending, resuming, and stopping a thread.

class MyThread implements Runnable {
    Thread thrd;
    boolean suspended;
    boolean stopped;
    MyThread(String name) {
        thrd = new Thread(this, name);
        suspended = false;
        stopped = false;
        thrd.start();
    }
    // This is the entry point for thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int i = 1; i < 1000; i++) {
                System.out.print(i + " ");
                if((i%10)==0) {
                    System.out.println();
                    Thread.sleep(250);
                }
                // Use synchronized block to check suspended and stopped.
                synchronized(this) {
                    while(suspended) {
                        wait();
                    }
                    if(stopped) break;
                }
            }
        }
        catch (InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " exiting.");
```

```
        }
        // Stop the thread.
        synchronized void myStop() {
                stopped = true;

                // The following ensures that a suspended thread can be  stopped.
                suspended = false;
                notify();
        }

        // Suspend the thread.
        synchronized void mySuspend() {
                suspended = true;
        }

        // Resume the thread.
        synchronized void myResume() {
                suspended = false;
                notify();
        }
}
class Suspend {
        public static void main(String[] args) {
                MyThread ob1 = new MyThread("My Thread");
                        try {
                        Thread.sleep(1000); // let ob1 thread start executing

                        ob1.mySuspend();
                        System.out.println("Suspending thread.");
                        Thread.sleep(1000);

                        ob1.myResume();
                        System.out.println("Resuming thread.");
                        Thread.sleep(1000);

                        ob1.mySuspend();
                        System.out.println("Suspending thread.");
                        Thread.sleep(1000);

                        ob1.myResume();
                        System.out.println("Resuming thread.");
                        Thread.sleep(1000);

                        ob1.mySuspend();
                        System.out.println("Stopping thread.");
                        ob1.myStop();
                }
                catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                }
                // wait for thread to finish
                try {
                        ob1.thrd.join();
                }
                catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                }

                System.out.println("Main thread exiting.");
        }
}
```

OUTPUT:

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
81 Resuming thread.
82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

APPLETS

- ✓ *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- ✓ There are two general varieties of applets: those based solely on the Abstract Window Toolkit (AWT) and those based on Swings. Both AWT and Swing support creation of Graphical User Interface (GUI).
- ✓ Let examine a simple applet:

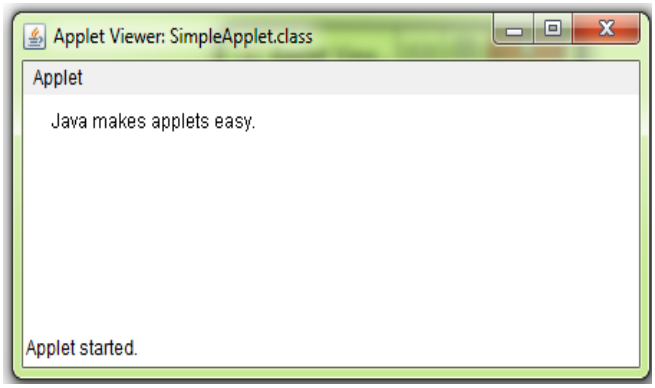
```
// A minimal AWT-based applet.
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Java makes applets easy.", 20, 20);
    }
}
```

- ✓ This applet begins with two **import** statements.
- ✓ The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface.
- ✓ The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass (either directly or indirectly) of **Applet**.
- ✓ The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program.
- ✓ Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output.
- ✓ The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
- ✓ Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form: void drawString(String *message*, int *x*, int *y*)
- ✓ Here, *message* is the string to be output beginning at *x*,*y*. In a Java window, the upperleft corner is location 0, 0.
- ✓ The applet does not have a **main( )** method.
- ✓ An applet begins execution when the name of its class is passed to an applet viewer or to a network browser.
- ✓ There are two ways in which you can run an applet:
  - Executing the applet within a Java-compatible web browser.
  - Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.
- ✓ To execute **SimpleApplet** with an applet viewer:
  1. Edit a Java source file.
  2. Compile your program.
  3. Execute the applet viewer, specifying the name of your applet's source file.

  To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer SimpleApplet.java. Now Html file is not required but it is for testing purpose only.
- ✓ The SimpleApplet source file look like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Java makes applets easy.", 20, 20);
    }
}
```

- ✓ The window produced by **SimpleApplet**, as displayed by the applet viewer, is:



A COMPLETTE APPLET SKELETON

- ✓ Most trivial applets override a set of methods that provide the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- ✓ These lifecycle methods are **init(), start(), stop()** and **destroy()** and they are defined by Applet.
- ✓ A fifth method, **paint()** is commonly override by AWT-based applets even though it is not a lifecycle method.
- ✓ These four lifecycle methods plus **paint()** can be assembled into the skeleton as shown below:

```
// An AWT-based Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
        // Called first.
        public void init() {
                // initialization
        }

        // Called second, after init(). Also called whenever the applet is restarted.
        public void start() {
                // start or resume execution
        }

        // Called when the applet is stopped.
        public void stop() {
                // suspends execution
        }

        //Called when applet is terminated. This is the last method executed.
        public void destroy() {
                // perform shutdown activities
        }

        // Called when an AWT-based applet's window must be restored.
        public void paint(Graphics g) {
                // redisplay contents of window
        }
}
```

APPLET INITITALIZATION AND TERMINATION:

- ✓ When an applet begins the following methods are called in this sequence:
  1. **init()**
  2. **start()**
  3. **paint()**
- ✓ When applet is terminated the following sequence of method calls takes place
  1. **stop()**
  2. **destroy()**
- ✓ The **init()** method is the first method to be called. In **init()** the applet will initialize variables and perform any other startup activities
- ✓ The **start()** method is called after init(). It is also called to restart an applet after it has been stopped. **start()** might be called more than once during the life cycle of an applet.
- ✓ The **paint()** method is called each time an AWT based applet's output must be redrawn.
- ✓ When the page containing the applet is left, the **stop()** is called. **stop()** is used to suspend any child threads created by the applet and to perform any other activities required to put the applet in a safe, idle state.
- ✓ The **destroy()** method is called when the applet is no longer needed. It is used to perform any shutdown operations required of the applet

REQUESTING REPAINT:

- ✓ An AWT- based applet writes to its window only when its **paint()** method is called by the run-time system.
- ✓ Whenever an applet needs to update the information displayed in its window, it simply calls **repaint()**
- ✓ The **repaint()** method is defined by the AWT's **Component** class and inherited by **Applet**. It causes the run-time system to execute a call to the applet's **paint()** method.
- ✓ The simplest version of **repaint()** is :

```
void repaint()
```

```
/*  A simple banner applet.
  This applet creates a thread that scrolls the message contained in msg right to left across
  the applet's window.  */
import java.awt.*;
import java.applet.*;
public class Banner extends Applet implements Runnable {
    String msg = " Java Rules the Web ";
    Thread t;
    boolean stopFlag;
    // Initialize t to null.
    public void init() {
        t = null;
    }
    // Start thread when the applet is needed.
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }
     // Entry point for the thread that runs the banner.
    public void run() {
        // Request a repaint every quarter second.
        for( ; ; ) {
                try {
                    repaint();
                    Thread.sleep(250);
                    if(stopFlag) break;
                } catch(InterruptedException exc) {
                System.out.println("thread interrupted");
            }
        }
    }
```

```
        // Pause the banner.
        public void stop() {
                stopFlag = true;
                t = null;
        }

        // Display the banner.
        public void paint(Graphics g) {
                char ch;
                ch = msg.charAt(0);
                msg = msg.substring(1, msg.length());
                msg += ch;
                g.drawString(msg, 50, 30);
        }
    }
```

USING THE STATUS WINDOW

- ✓ In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.
- ✓ To do so, call **showStatus()**, which is defined by **Applet**. The general form is:
  void showStatus(String msg)
- ✓ Here msg is the string to be displayed
- ✓ The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors.

//Using the Status Window

```
import java.awt.*;
import java.applet.*;
/*  <applet code="SimpleApplet.class" width=200 height=60> </applet> */
public class ShowStatus extends Applet {
    public void paint(Graphics g) {
            g.drawString("Java makes applets easy.", 200, 60);
            showStatus("this is java applet");
    }
}
```

PASSING PARAMETERS TO APPLETS

- ✓ Parameters can be passed to an applet. Parameter specifies some setting or attribute associated with the applet.
- ✓ To pass a parameter to an applet, use PARAM attribute of the APPLET tag, specifying the paramerer's name and value. To retrieve a parameter, use the getParameter() method,defined by Apple.Its general form is:
  String getParameter(String paramName)

// Pass a parameter to an applet.

```
import java.awt.*;
import java.applet.*;
/*  <applet code="Param" width=300 height=80>
<param name=author value="knreddy">
<param name=purpose value="Demonstrate Parameters">
<param name=version value=2>
</applet>     */
public class Param extends Applet {
    String author;
    String purpose;
    int ver;
    public void start() {
            String temp;
            author = getParameter("author");
            if(author == null) author = "not found";
            purpose = getParameter("purpose");
```

```
            if(purpose == null) purpose = "not found";
            temp = getParameter("version");
            try {
                if(temp != null)
                    ver = Integer.parseInt(temp);
                else
                    ver = 0;
            } catch(NumberFormatException exc) {
                ver = -1; / / error code
             }
        }
        public void paint(Graphics g) {
            g.drawString("Purpose: " + purpose, 10, 20);
            g.drawString("By: " + author, 10, 40);
            g.drawString("Version: " + ver, 10, 60);
        }
    }
```

- ✓ The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program.
- ✓ When an instance variable is declared as **transient**, then its value need not persist when an object is stored.
- ✓ Sometimes, knowing the type of an object during run time is useful. Java provides the run-time operator **instanceof:**
- ✓ The **instanceof** operator has this general form:
  objref instanceof *type*
- ✓ Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**.
- ✓ By modifying a class, a method, or interface with **strictfp**, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by **strictfp**, all the methods in the class are also modified by **strictfp** automatically.
- ✓ Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.
- ✓ **assert** is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program.
- ✓ The **assert** keyword has two forms. The first is shown here:
  assert *condition*;
- ✓ Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown.
- ✓ The second form of **assert** is shown here:
  assert *condition: expr*;
- ✓ In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails.

NETWORKING BASICS

- ✓ The core of Java's networking support is the concept of a *socket*. A socket identifies an endpoint in a network.
- ✓ Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a *port,* which is a numbered socket on a particular machine.
- ✓ A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique.
- ✓ Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.
- ✓ *Transmission Control Protocol* (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data.
- ✓ A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.
- ✓ A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4).
- ✓ IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks.
- ✓ The name of an Internet address, called its *domain name,* describes a machine's location in a name space.
- ✓ For example, **www.SREC.com** is in the *COM* top-level domain; it is called *SREC*, and *www* identifies the server for web requests.
- ✓ An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS).* This enables users to work with domain names, but the Internet operates on IP addresses.

THE NETWORKING CLASSES AND INTERFACES

- ✓ Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.
- ✓ The classes contained in the **java.net** package are shown here:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager	MulticastSocket	StandardSocketOption (Added by JDK 7.)
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

- ✓ The **java.net** package's interfaces are:

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily (Added by JDK 7.)	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption (Added by JDK 7.)	

InetAddress class

- ✓ The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.
- ✓ The **InetAddress** class has no visible constructors. To create an **InetAddress** object, use one of its static methods.
  static InetAddress getLocalHost( ) throws UnknownHostException
  static InetAddress getByName(String *hostName*) throws UnknownHostException
- ✓ The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException.**
- ✓ There are several methods that can be called on an instance of **InetAddress**.
  String getHostAddress()
  String getHostName()
- ✓ The **getHostAddress()** method returns a string that lists the host IP address using its numeric form. The **getHostName()** address returns the name that represents the host address.

// Demonstrate InetAddress.

```
import java.net.*;

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

    try {
      InetAddress address = InetAddress.getByName("www.mcgraw-hill.com");
      System.out.println("Host name: " + address.getHostName());
      System.out.println("Address: " + address.getHostAddress());

      System.out.println();

      address = InetAddress.getByName("www.knreddycse.weebly.com");
      System.out.println("Host name: " + address.getHostName());
      System.out.println("Address: " + address.getHostAddress());

      System.out.println();

      address = InetAddress.getByName("www.srec.com");
      System.out.println("Host name: " + address.getHostName());
      System.out.println("Address: " + address.getHostAddress());
    }
    catch (UnknownHostException exc) {
      System.out.println(exc);
    }
  }
}
```

OUTPUT:

```
Host name: www.mcgraw-hill.com
Address: 184.26.168.92

Host name: www.knreddycse.weebly.com
Address: 199.34.228.53

Host name: www.srec.com
Address: 184.168.221.59
```

Inet4Address and Inet6Address

- ✓ Beginning with version 1.4, Java has included support for IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a new-style IPv6 address.

TCP/IP Client Sockets

- ✓ TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.
- ✓ There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients.
- ✓ The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers.
- ✓ The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges.
- ✓ The creation of a **Socket** object implicitly establishes a connection between the client and server.
- ✓ **Socket** defines several constructors:
  Socket(String hostname, int port) throws UnknownHostException, IOException
- ✓ **Socket** defines several instance methods.

InetAddress getInetAddress()	Returns InetAddress associated with Socket object. It returns null if Scoket is not connected
int getPort()	Returns the port number on the server. Otherwise it returns 0
int getLocalPort()	Returns local port number .It returns -1 if Socket is not bound to a port

- ✓ We can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods

InputStream getInputStream() throws IOException	Returns the input stream associated with the invoking socket.
OutputStream getOutputStream() throws IOException	Returns the output stream associated with the invoking socket.

- ✓ Several other methods are available, including **connect( ),** which allows you to specify a new connection; **isConnected( ),** which returns true if the socket is connected to a server; **isBound( ),** which returns true if the socket is bound to an address; and **isClosed( ),** which returns true if the socket is closed. To close a socket, call **close( )**.
- ✓ Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try** with-resources block to manage a socket.

```
// Demonstrate Sockets.
import java.net.*;
import java.io.*;

class SocketDemo {
  public static void main(String[] args) {
    int ch;
    Socket socket = null;

    try {
      // Create a socket connected to whois.internic.net, port 43.
      socket = new Socket("whois.internic.net", 43);

      // Obtain input and output streams.
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();
```

```
        // Construct a request string.
        String str = (args.length == 0 ? "mcgraw-hill.com" :
                                  args[0]) + "\n";
        // Convert to bytes.
        byte[] buf = str.getBytes();

        // Send request.
        out.write(buf);

        // Read and display response.
        while ((ch = in.read()) != -1) {
          System.out.print((char) ch);
        }
      }
      catch(IOException exc) {
        System.out.println(exc);
      }
      finally {
        try {
          if(socket != null) socket.close();
        } catch(IOException exc) {
          System.out.println("Error closing socket: " + exc);
        }
      }
    }
  }

// Use automatic resource management to close a socket.
import java.net.*;
import java.io.*;
class SocketDemo {
  public static void main(String[] args) {
    int ch;

    // Create a socket connected to internic.net, port 43. Manage this
    // socket with a try-with-resources block.
    try ( Socket socket = new Socket("whois.internic.net", 43) ) {

      // Obtain input and output streams.
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();

      // Construct a request string.
      String str = (args.length == 0 ? "mcgraw-hill.com" :
                                  args[0]) + "\n";
      // Convert to bytes.
      byte[] buf = str.getBytes();

      // Send request.
      out.write(buf);

      // Read and display response.
      while ((ch = in.read()) != -1) {
        System.out.print((char) ch);
      }
    } catch(IOException exc) {
      System.out.println(exc);
    }
    // The socket is now closed.
  }
}
```

THE URL CLASS

- ✓ The URL stands foe Uniform Resource Locator.
- ✓ The URL provides a way to uniquely identify or address information on the Internet.
- ✓ Java provides support for URLs with the **URL** class.
- ✓ All URLs share the same basic format, although some variation is allowed.
- ✓ Here are two examples:
  http://www.mhhe.com/ and **http://www. mhhe.com:80/index.htm**.
- ✓ A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP;
- ✓ The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:).
- ✓ The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.)
- ✓ The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource. Thus, **http://www.mhhe.com/** is the same as **http://www.mhee.com/index.htm**.
- ✓ Java's **URL** class has several constructors; each can throw a **MalformedURLException**.
  URL(String urlSpecifier) throws MalformedURLException
  Here, *urlSpecifier is a string that specifies a complete URL.*
- ✓ The next two forms of the constructor allow you to break up the URL into its component parts:
  URL(String protocolName, String hostName, int port, String path)
  throws MalformedURLException
  URL(String protocolName, String hostName, String path)
  throws MalformedURLException
- ✓ There are methods defined by **URL** that let to obtain the individual components of a URL.
  They are:
  String getProtocol()
  String getHost()
  String getFile()
  int getPort()

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String[] args) {

    try {
      URL url = new URL("http://www.knreddycse.weebly.com:80/index.html");

      System.out.println("Protocol: " + url.getProtocol());
      System.out.println("Port: " + url.getPort());

      System.out.println("Host: " + url.getHost());
      System.out.println("File: " + url.getFile());
    }
    catch (MalformedURLException exc) {
      System.out.println("Invalid URL: " + exc);
    }
  }
}
```

OUTPUT:

```
Protocol: http
Port: 80
Host: www.knreddycse.weebly.com
File: /index.html
```

URLConnection class

- ✓ **URLConnection** is a general-purpose class for accessing the attributes of a remote resource.
- ✓ These attributes are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.
- ✓ **URLConnection** defines several methods. Here is a sampling:

Method	Description
int getContentLength()	Returns the size in bytes of the content associated with the resource. If the length is unavailable,-1 is returned.
long getContentLengthLong()	Returns the size in bytes of the content associated with the resource. If the length is unavailable,-1 is returned.(added by JDK7)
String getContentType()	Returns the type of content found in the resource. Returns null if the content type is not available.
long getDate()	Returns the time and date of the response represented in terms of mill seconds since January 1, 1970 GMT. Zero is returned if the time and date are not available.
long getExpiration()	Returns the expiration time and date of the response represented in terms of mill seconds since January 1, 1970 GMT. Zero is returned if the expiration date is not available.
String getHeaderField(int idx)	Returns the value of the header field at the idx. Returns null if the value if idx exceeds the number of fields.
String getHeaderField(String filedName)	Returns the value of the header field whose name is specified by fieldName. Returns null if the specified name is not found
String getHeaderFieldKey(int idx)	Returns the header field key at index idx. Returns null if the value of idx exceeds the number of fields
Map<String,List<String>> getHeaderFields()	Returns a map that contains all of the header fields and values
long getLastModified()	Returns the time and date, represented in terms of milli seconds since January 1, 1970 GMT, of the last modification of the resource. Zero if returned if the last modified date is unavailable.
InputStream getInputStream() throws IOException	Returns an InputStream that is linked to the connection.
OutputStream getOutputStream() throws IOException	Returns an OutputStream that is linked to the connection.

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.*;
class UCDemo
{
  public static void main(String[] args) {
    InputStream in = null;
    URLConnection connection = null;
    try {
      URL url = new URL("http://www.mcgraw-hill.com");
      connection = url.openConnection();
      // get date
      long d = connection.getDate();
      if(d==0)
        System.out.println("No date information.");
```

```
        else
          System.out.println("Date: " + new Date(d));
        // get content type
        System.out.println("Content-Type: " + connection.getContentType());
        // get expiration date
        d = connection.getExpiration();
        if(d==0)
          System.out.println("No expiration information.");
        else
          System.out.println("Expires: " + new Date(d));
        // get last-modified date
        d = connection.getLastModified();
        if(d==0)
          System.out.println("No last-modified information.");
        else
          System.out.println("Last-Modified: " + new Date(d));
        // get content length
        long len = connection.getContentLengthLong();
        if(len == -1)
          System.out.println("Content length unavailable.");
        else
          System.out.println("Content-Length: " + len);
        if(len != 0) {
          System.out.println("=== Content ===");
          in = connection.getInputStream();
          int ch;
          while (((ch = in.read()) != -1)) {
            System.out.print((char) ch);
          }
        } else {
          System.out.println("No content available.");
        }
      } catch(IOException exc) {
        System.out.println("Connection Error: " + exc);
      } finally {
        try {
          if(in != null) in.close();
        } catch(IOException exc) {
          System.out.println("Error closing connection: " + exc);
        }
      }
    }
  }
```

HttpURLConnection

- ✓ Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection.**
- ✓ **HttpURLConnection** in obtained, by calling **openConnection( )** on a **URL** object, but it must cast the result to **HttpURLConnection.**
- ✓ Once a reference to an **HttpURLConnection** object is obtained, we can use any of the methods inherited from **URLConnection.**
- ✓ There are several methods defined by **HttpURLConnection.**

String getRequestMethod()	Returns a string representing how URL requests are made. The default is GET. Other options, such as POST are available
int getResponseCode() throws IOException	Returns HTTP response code, -1is returned if no response code can obtained. An IOException throws if connection fails
String getResponseMessage() throws IOException	Returns the response message associated with the response code. Returns null if no message is available. An IOException throws if connection fails

// Demonstrate HttpURLConnection.

```
import java.net.*;
import java.io.*;
import java.util.*;

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

    try {
      URL url = new URL("http://www.mcgraw-hill.com");
         HttpURLConnection connection =
           (HttpURLConnection) url.openConnection();

      // Display request method.
      System.out.println("Request method is " +
                         connection.getRequestMethod());

      // Display response code.
      System.out.println("Response code is " +
                         connection.getResponseCode());

      // Display response message.
      System.out.println("Response Message is " +
                         connection.getResponseMessage());

      // Get a list of the header fields and a set
      // of the header keys.
      Map<String, List<String>> hdrMap = connection.getHeaderFields();
      Set<String> hdrKeys = hdrMap.keySet();

      System.out.println("\nHere is the header:");

      // Display all header keys and values.
      for(String k : hdrKeys) {
        System.out.println("Key: " + k +
                           "  Value: " + hdrMap.get(k));
      }
    } catch(IOException exc) {
      System.out.println(exc);
    }
  }
}
```

The URI Class

- ✓ The **URI** class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs.
- ✓ In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies

- ✓ The **java.net** package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session.
- ✓ The classes are **CookieHandler, CookieManager,** and **HttpCookie.**
- ✓ The interfaces are **CookiePolicy** and **CookieStore.**

TCP/IP Server Sockets

- ✓ The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports.
- ✓ **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket,** it will register itself with the system as having an interest in client connections.
- ✓ The constructors for **ServerSocket** reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be.
- ✓ The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

ServerSocket(int port) throws IOException	Creates socket on the specified port with a queue length of 50
ServerSocket(int port, int maxQueue) throws IOException	Creates socket on the specified port with a queue length of maxQueue
ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException	Creates socket on the specified port with a queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds

- ✓ **ServerSocket** has a method called **accept( ),** which is a blocking call that will wait for a client to initiate communications and then return with a normal **Socket** that is then used for communication with the client.

DATAGRAMS

- ✓ Datagrams provide an alternative to the TCP/IP style networking.
- ✓ *Datagrams* are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.
- ✓ Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s.

DatagramSocket

- ✓ **DatagramSocket** defines four public constructors. They are:
  DatagramSocket( ) throws SocketException
  DatagramSocket(int *port*) throws SocketException
  DatagramSocket(int *port,* InetAddress *ipAddress*) throws SocketException
  DatagramSocket(SocketAddress *address*) throws SocketException
- ✓ **DatagramSocket** defines many methods. Two of the most important are **send()** and **receive()**:
  void send(DatagramPacket *packet)* throws IOException
  void receive(DatagramPacket *packet*) throws IOException
- ✓ The **send( )** method sends a packet to the port specified by *packet*. The **receive( )** method waits for a packet to be received from the port specified by *packet* and returns the result.
- ✓ **DatagramSocket** also defines the **close**( ) method, which closes the socket. Beginning with JDK 7, **DatagramSocket** implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try**-with-resources block.

DatagramPacket

- ✓ **DatagramPacket** defines several constructors. They are:
  DatagramPacket(byte *data* [ ], int *size*)
  DatagramPacket(byte *data* [ ], int *offset,* int *size*)
  DatagramPacket(byte *data* [ ], int *size,* InetAddress *ipAddress,* int *port*)
  DatagramPacket(byte *data* [ ], int *offset,* int *size,* InetAddress *ipAddress,* int *port*)
- ✓ **DatagramPacket** defines several methods that give access to the address and port number of a packet, as well as the raw data and its length.
- ✓ In general, the **get** methods are used on packets that are received and the **set** methods are used on packets that will be sent.

InetAddress getAddress()	Returns the address of the source(for datagrams being received) or destination (for datagrams being sent)
byte[] getData()	Returns the byte array that contains the data buffer. Mostly used to retrieve data from the datagram after it has been received
int getLength()	Returns the number of bytes of data contained in the buffer. This may be less than the size of the underlying byte array
int getOffset()	Returns the starting index of the data in the buffer
int getPort()	Returns the port number used by the host on the other side of the connection
void setData(byte[] data)	Sets the packets data to *data*, the offset to zero, and the length to the number of bytes in *data.*
void setData(byte[] data, int idx, int size)	Sets the packets data to *data*, the offset to idx, and the length to *size.*
void setLength(int size)	Sets the packets data to *size.* This value plus the offset must not exceed the length of the underlying bytes array

A Datagram Example:

The following example demonstrates datagrams by implementing a very simple client and server. In this example the server reads string entered at the keyboard and sends them to the client. The client simply waits until it receives a packet and then displays the string. This process continues until 'stop' is entered. In that case, both the client and server terminate.

```
// Demonstrate datagrams -- server side.
import java.net.*;
import java.io.*;
class DGServer {
  // These ports were chosen arbitrarily. You must use
  // unused ports on your machine.
  public static int clientPort = 50000;
  public static int serverPort = 50001;
  public static DatagramSocket ds;
  public static void dgServer() throws IOException {
    byte[] buffer;
    String str;
    BufferedReader conin = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters. Enter 'stop' to quit.");
    for(;;) {
      // read a string from the keyboard
      str = conin.readLine();
      // convert string to byte array for transmission
      buffer = str.getBytes();
      // send a new packet that contains the string
      ds.send(new DatagramPacket(buffer, buffer.length,InetAddress.getLocalHost(), clientPort));
      // quit when "stop" is entered
      if(str.equals("stop")) {
        System.out.println("Server Quits.");
        return;
      }
    }
```

```
  }
  public static void main(String[] args) {
    ds = null;

    try {
      ds = new DatagramSocket(serverPort);
      dgServer();
    } catch(IOException exc) {
      System.out.println("Communication error: " + exc);
    } finally {
      if(ds != null) ds.close();
    }
  }
}
```

// Demonstrate datagrams -- client side.

```
import java.net.*;
import java.io.*;

class DGClient {
  // This ports was choosen arbitrarily. You must use
  // an unused port on your machine.
  public static int clientPort = 50000;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static void dgClient() throws IOException {
    String str;
    byte[] buffer = new byte[buffer_size];

    System.out.println("Receiving Data");
    for(;;) {
      // create a new packet to receive the data
      DatagramPacket p = new DatagramPacket(buffer, buffer.length);
      // wait for a packet
      ds.receive(p);
      // convert buffer into String
      str = new String(p.getData(), 0, p.getLength());
      // display the string on the client
      System.out.println(str);
      // quit when "stop" is received.
      if(str.equals("stop")) {
        System.out.println("Client Stopping.");
        break;
      }
    }
  }
  public static void main(String[] args) {
    ds = null;
    try {
      ds = new DatagramSocket(clientPort);
      dgClient();
    } catch(IOException exc) {
      System.out.println("Communication error: " + exc);
    } finally {
      if(ds != null) ds.close();
    }
  }
}
```