Input/Output: files, stream classes, reading console input. Threads: thread model, use of Thread class and Runnable interface, thread synchronization, multithreading.

Streams

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

Byte Streams and Character Streams

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

Character streams provide a convenient means for handling
input and output of characters. They use Unicode and, therefore, can be internationalized.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream.

to use the stream classes, you must **import java.io.**

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer.

The Predefined Streams

all Java programs automatically import the java.lang package. This package
defines a class called System, which encapsulates several aspects of the run-time environment.

System also contains three predefined stream variables: in, out, and err. These fields are declared as public, static, and final within System. This means that they can be used by any other part of your program and without reference to a specific System object.
System.out refers to the standard output stream. By default, this is the console.

System.in refers to standard input, which is the keyboard by default.
System.err refers to the standard error stream, which also is the console by default.

System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.

Reading Console Input

only way to perform console input was to use a byte stream.

A commonly used constructor is shown here:

BufferedReader(Reader inputReader)

inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters.

To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

InputStreamReader(InputStream inputStream)

Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters

To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is

int read( ) throws IOException

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
public static void main(String args[]) throws IOException
{
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
c = (char) br.read();
System.out.println(c);
} while(c != 'q');
}
}
```

Here is a sample run:
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q

Reading Strings

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
public static void main(String args[]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

```
// A tiny editor.
import java.io.*;
class TinyEdit {
public static void main(String args[]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str[] = new String[100];
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
for(int i=0; i<100; i++) {
str[i] = br.readLine();
if(str[i].equals("stop")) break;
}
System.out.println("\nHere is your file:");
```

```
// display the lines
for(int i=0; i<100; i++) {
if(str[i].equals("stop")) break;
System.out.println(str[i]);
}
}
}
```

Here is a sample run:
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.

stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.

Writing Console Output

Console output is most easily accomplished with print( ) and println( ).

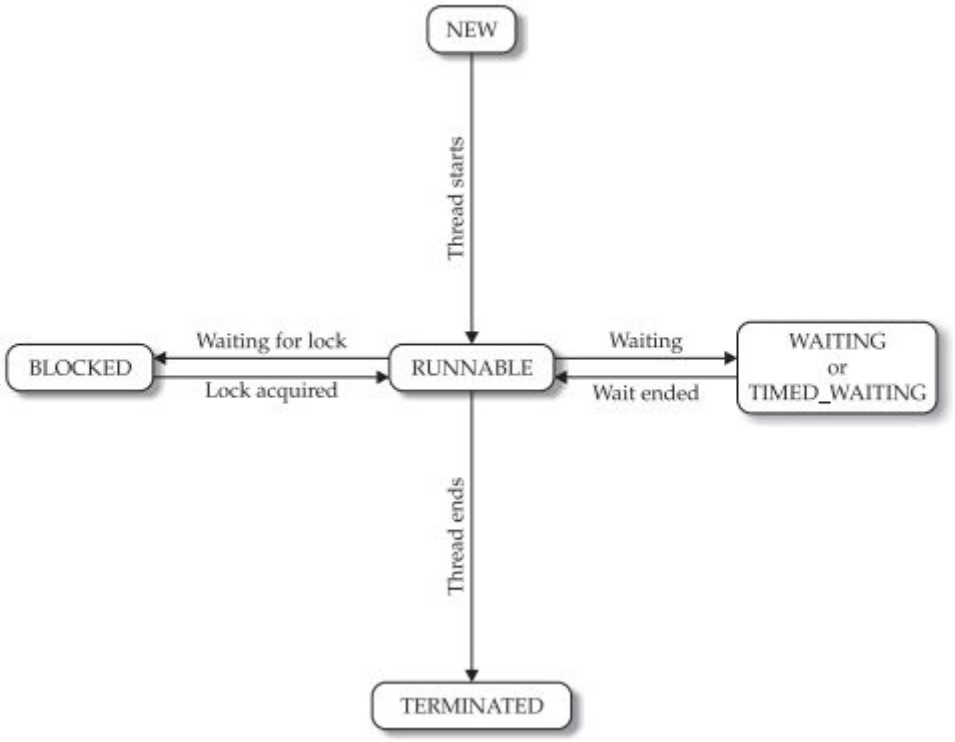
The simplest
form of write( ) defined by PrintStream is shown here:

```
void write(int byteval)
```

byteval is declared as an integer,

```
// Demonstrate System.out.write().
class WriteDemo {
public static void main(String args[]) {
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

THREAD
Thread's State



Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your 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.

general form is shown here:

static Thread currentThread( )

Creating a Thread

Java defines two ways in which this can be accomplished:

• You can implement the Runnable interface.

• You can extend the Thread class, itself.

(I)Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface.To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run( )

Inside run( ), you will define the code that constitutes the new thread.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

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 second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }


    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

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

Inside NewThread's constructor, a new Thread object is created by the following statement:

t = new Thread(this, "Demo Thread");

OUTPUT

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

(II)Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Notice the call to super( ) inside NewThread. This invokes the following form of the Thread constructor:

public Thread(String threadName)

OUTPUT

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Creating Multiple Threads

your program can spawn as many threads as it needs.

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {




      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}

class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");

    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

OUTPUT

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Key to synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
```

```
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
      System.out.println("]");
    }
  }

  class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
      target = targ;
      msg = s;
      t = new Thread(this);
      t.start();
    }
    public void run() {
      target.call(msg);
    }
  }

  class Synch {
    public static void main(String args[]) {
      Callme target = new Callme();
      Caller ob1 = new Caller(target, "Hello");
      Caller ob2 = new Caller(target, "Synchronized");
      Caller ob3 = new Caller(target, "World");

      // wait for threads to end
      try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
```

Here is the output produced by this program:

```
Hello[Synchronized[World]
]
]
```

You can synchronize your code in either of two ways.

1.Using Synchronized Methods

To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.

To fix the preceding program, you must serialize access to call( ). That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede call( )'s definition with the keyword synchronized, as shown here:

```
class Callme {
  synchronized void call(String msg) {
  ...
```

This prevents other threads from entering call( ) while another thread is using it. After synchronized has been added to call( ), the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

2.The synchronized Statement

This is the general form of the synchronized statement:

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

Here, object 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.
Here is an alternative version of the preceding example, using a synchronized block within the run( ) method:

```
// synchronize calls to call()
public void run() {
  synchronized(target) { // synchronized block
    target.call(msg);
  }
```

OUTPUT

```
[Hello]
[Synchronized]
[World]
```

Multithreading

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!