Java Tutorial - Fall 99

Part4: Multithreading, Useful Java Classes, I/ O and Networking

Instructors: Geoffrey Fox , Nancy McCracken, Tom Scavo
Syracuse University
111 College Place
Syracuse
New York 13244-4100

Threads are part of the Java Language!

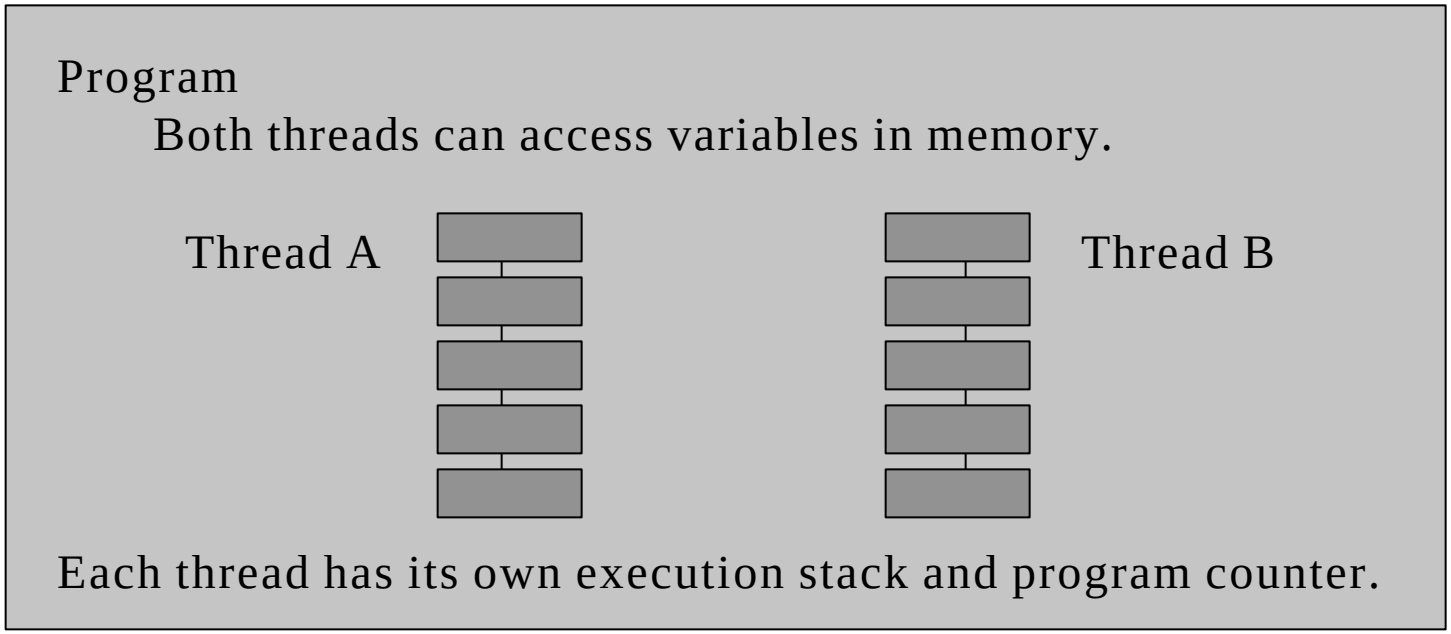
(a more serious discussion than in part III of tutorial)

Initial Remarks on Threads

- Java is remarkable for threads being built into the language
- Threads are "light-weight" processes (unlike UNIX processes), which communicate by a combination of shared memory and message passing
  - This communication mechanism is employed naturally by Java
- Java threads are limited and for those coming from an HPCC background, we note Java threads have no immediate support for some key parallel computing concepts (see work of Chandy at Caltech) such as distributed memory (threads running in separate operating system instances)

Thread Concurrency

- Each thread is a sequence of steps within a program:



- Two or more threads can give the appearance of running at the same time even on a single CPU by sharing the CPU. Each thread gives up execution voluntarily (by executing yield(), etc.) or because its time slice has ended.
- Note that the Java system already has concurrently running threads for garbage collection, window management, etc.

Thread Execution

- Threads are implemented by a scheduler in Java, which asks the local operating system to run threads in the "runnable" state.
- Typically, the OS runs each thread in turn for a "time slice". However, some operating systems (early versions of Solaris, e.g.) run a thread to completion unless another thread of higher priority preempts the running thread.
- Java threads are based on a locking mechanism using monitors for synchronization, introduced by Hoare in 1974.

How to Use Threads

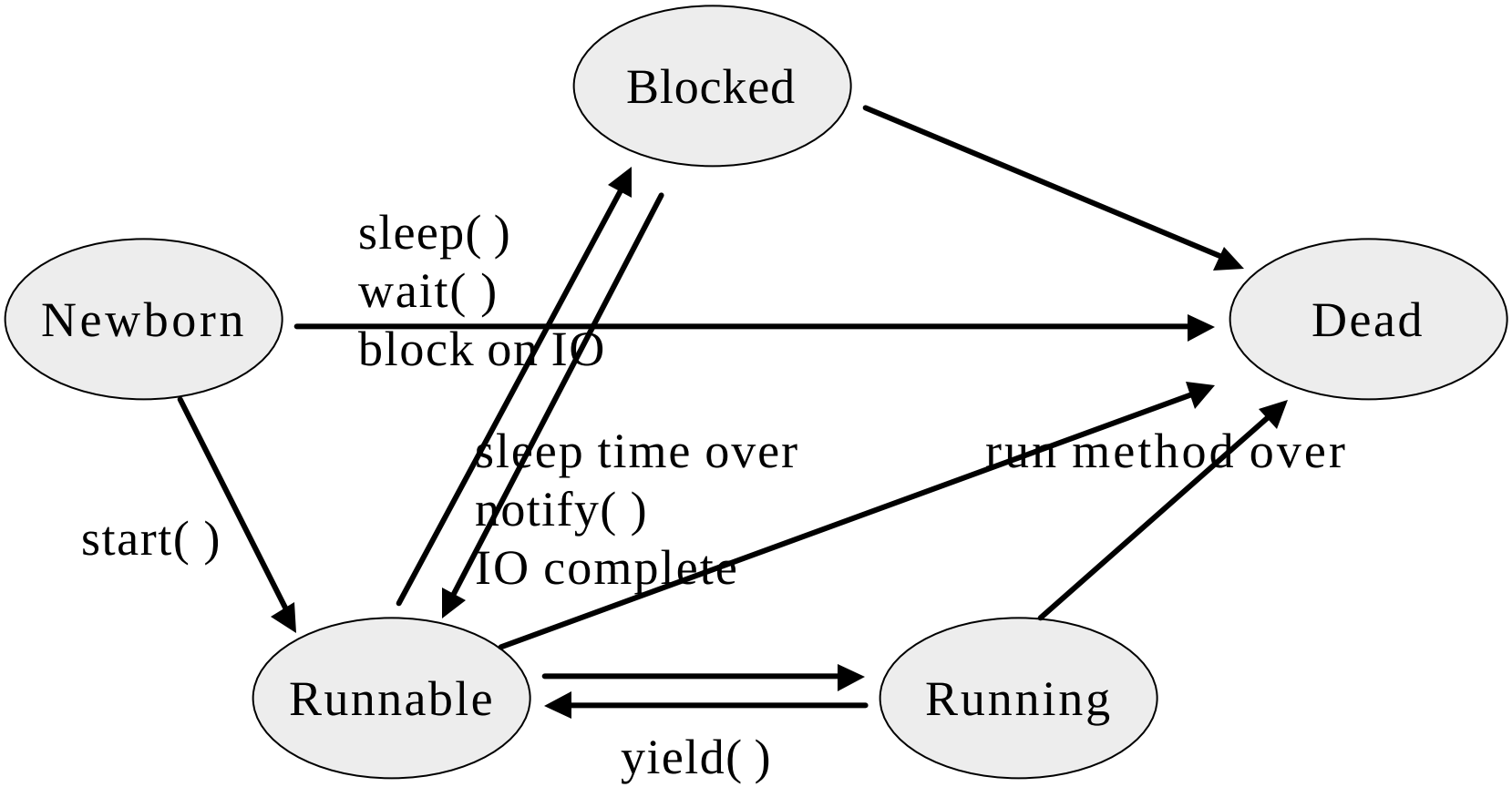
- One can implement threads in two ways:
  - First, by subclassing the Thread class
  - Second, by implementing the Runnable interface
- A class that implements the Runnable interface (including the Thread class itself) must implement the run() method containing the "body" of the thread.
- The Runnable interface makes it possible for an applet to utilize threads. (Remember, an applet extends the Applet class and so cannot multiply inherit from the Thread class.)

Subclassing the Thread Class

- One way to create threads is to write a separate class that subclasses the Thread class.
  - The main line of execution is put in a method called run(), which overrides the method of the same name from the Thread class.
- Instances of this subclass are instantiated like this:
  - MyThread mt = new MyThread( );
- Thread control:
  - When the thread is created, it does not automatically start running. The class that creates it must call the Thread method start().
  - Other methods may be called: Thread.sleep( ), Thread.yield( ), and join( ). (Note that sleep( ) and yield( ) are static methods.)

The Life of a Thread

- A thread is always in one of the five states shown in this diagram, which includes the most common methods for changing state:



Moving out of a Blocked State

- A thread must move out of a blocked state (or the not runnable state) into the runnable state using the opposite of whatever put it into the blocked state:
  - If a thread has been put to sleep(), the specified timeout period must expire.
  - If a thread called wait(), then someone else using the resource for which it is waiting must call notify() or notifyAll().
  - If a thread is waiting for the completion of an input or output operation, then the operation must finish.
- There is a method isAlive() that returns true if the method is running, runnable or blocked, and returns false if the method is a new thread or dead.

Thread Priorities

- Every thread has a priority, which can be set by the user with setPriority(int) using constants MIN_PRIORITY (1), MAX_PRIORITY(10), or NORM_PRIORITY, or it inherits the priority of the thread it was created from.
- Whenever the thread scheduler picks a thread to run, it picks the highest priority thread that is currently runnable, which is fixed priority scheduling. (If there is more than one thread with the same priority, each thread gets a turn in some order.) Lower priority threads are not run as long as there is a runnable higher priority thread.
- The scheduling is also preemptive: if a higher priority thread becomes available, it is run.
- The Java Run-time itself does not have time slicing, but time slicing may be provided by the underlying operating system.

Synchronization

- In Java, two threads can communicate by accessing a shared variable (shared-memory model).
- If two threads can both modify an object, that is, they can both execute a method that assigns to a shared variable, then the modifications must be synchronized.
- This is easy - just declare the method to be synchronized! Java will associate a lock with each object containing the method - only one synchronized method from that object can be executed at a time.
- Suppose more than one thread can access an account:
  - public class Account
  - { int bankBalance; ...
  - public synchronized void CreditAcct(int amt)
  - {... bankBalance += amt; ... }}

More on Synchronization

- Sometimes while a thread is executing a synchronized method, it wants to wait for some condition to become true. Therefore, it may need to give up the lock on the synchronized method for a time.
- This is implemented by the monitor is associated with the instance of the class; it has a lock and a queue.
- The queue holds all threads waiting to execute a synchronized method.
  - A thread enters the queue by calling wait() inside the method or when another thread is already executing the method.
  - When a synchronized method returns, or when a method calls wait(), another thread may access the object.
  - As always, the scheduler chooses the highest-priority thread among those in the queue.
  - If a thread is put into the queue by calling wait(), it can't be scheduled for execution until some other thread calls notify().

Thread wait

- If a thread must wait for the state of an object to change, it should call wait() inside a synchronized method.
  - void wait()
  - void wait(int timeout)
  - These methods cause the thread to wait until notified or until the timeout period expires, respectively.
- Without a timeout, the thread waits until either notify() or notifyAll() is called. (See next foil.)
    - » wait() is called by the thread owning the lock associated with a particular object; wait() releases this lock (atomically, i.e., safely)
- With a timeout, wait can be used in the place of sleep, except that a notifyAll will wake it up, unlike sleep which always waits until completion of the sleep time.

Thread notify

- void notify()
- void notifyAll()
  - These methods must be called from a synchronized method.
  - These methods notify a waiting thread or threads.
- notify() notifies the thread associated with the given synchronization object that has been waiting the longest time
- notifyAll() notifies all threads associated with the given object and is therefore safer than notify()
- One can mark a variable as "threadsafe" to inform the compiler that only one thread will be modifying this variable.

Threads and Synchronization - Example

- Suppose that several threads are updating a bank balance (i.e., several threads can access one instance of class Account below). Then a thread that finds insufficient funds to debit an account can wait until another thread adds to the account:
  - public class Account
  - {int bankBalance; ...
  - public synchronized void DebitAcct (int amt)
  - { while ((bankBalance - amt) < 0) wait();
  - bankBalance -= amt; ... }
  - public synchronized void CreditAcct (int amt)
  - { bankBalance += amt;
  - notify(); ... } }

Useful Java Classes

The Overarching Object Class

- Public class Object is the root of the class hierarchy. Every Java class has Object as its ultimate parent and so any object (object with a small "o" is any instance of a class) can use methods of Object.
- Methods of Object include:
  - clone() creates a clone of the object
  - equals(Object) compares two objects, returning a boolean result
  - getClass() returns a descriptor of type Class (a child of Object) defining the class of the object
  - toString() returns a String representation of the object. It is expected that each subclass will override this method
  - wait(...) in various forms causes threads to wait
  - finalize() executed when the object is deleted by system (i.e., garbage collected)

Determining and Testing Class of Object

- Suppose we have an object called obj. We get the class of obj by:
  - Class class = obj.getClass();
- and its name by:
  - String name = class.getName();
- One can also use instanceof in following fashion:
  - "foo" instanceof String
- evaluates to true, but
  - (new mPoint(x,y)) instanceof String
- evaluates to false.

java.lang.Object Wrappers

- Primitive types such as int, char, float, etc. are NOT classes. Thus one cannot use methods such as
  - int var;
  - var.toString();
- ALL primitive types have associated wrappers:
  - Character myChar = new Character('A');
- The Character class has methods such as:
  - if (myChar.equals(ch)) ...
  - System.out.print(myChar.toString());
- There are also many static (class) methods:
  - ch = Character.toLowerCase(myChar);
  - if (Character.isUpperCase(myChar)) ...
- The methods in a wrapper class are also useful to convert types, such as a String to a Double.

The java.lang.Math class

- This class provides standard mathematical functions, using types int, long, float and double.
- It is a static class, meaning that you only use the methods and never create "Math objects".
- The methods include
  - » IEEEremainder, abs, ceil, cos, exp, floor, log, max, min, pow, random, sin, sqrt, and other trig functions.
  - – The random number generator is a linear congruential generator, which is fast but not random enough for many scientific applications.

The Date class

- This class provides an implementation of "date" structures. Date has methods to create and compare dates, obtain and set the time, and convert dates to strings.
- The Date constructor creates today's date:
  - Date today = new Date();
- In Java 1.1, most Date methods have been deprecated in favor of the Calendar class:
  - Calendar date1 = Calendar.getInstance();
  - date1.set(999, 12, 31); / * Dec. 31, 999 */
  - Calendar date2 = Calendar.getInstance();
  - date2.set(1996, 12, 31, 23, 59, 59)
    - /* Dec.31,1996 at 23:59:59 */

The String Class

- Strings are fixed-length collections of Unicode characters.
- Usually a string is created from a string literal or by using the constructor on an array of characters:
  - String greeting = "Hello";
- or
  - char[ ] bunch = {'H', 'e', 'l', 'l', 'o'};
  - String greeting = new String(bunch);
- Once created, individual characters of a string cannot be changed in place. The following example uses String methods to create a new string:
  - String test = "Chicken soup with rice";
  - int n = test.indexOf('w');
  - String newtest = test.substring(1,n-1) + "is n" + test.substring(n+5);
  - / * giving "Chicken soup is nice" */

More on Strings and the StringBuffer Class

- String comparison is done with the methods equals() and equalsIgnoreCase(). Note that == tests if two strings are the same string instance, while equals() tests if two distinct strings have the same characters.
- Other methods include length(), charAt(int) and toLowerCase().
- The StringBuffer class has mutable strings, but with a fixed maximum size. Methods such as append(...) automatically extend the length of the string.

Example using StringBuffer

- This class returns an object of class String that reverses order of characters in its argument:
  - class ReverseString
  - { public static String reverse(String s)
  - { int i, len = s.length();
  - StringBuffer dest = new StringBuffer(len);
  - for(i = (len-1); i >= 0 ; i--)
  - { dest.append(s.charAt(i));
  - }
  - return dest.toString();
  - }
  - }

The Vector Class

- In Java, while you can give the size of an array at run time, you cannot dynamically change the size of an array during the computation. The vector class provides a data structure with just this property, but the restriction is that all of the elements must be of type Object.
  - Usually, we insert an element of any type and Java will convert it to an Object, but when you extract an element, you must explicitly cast it to convert it back to the type you want.
- A vector is created with an "initial capacity" and a "capacity increment". (The default is an initial capacity of 10 and an increment that doubles each time.) As you add elements, if the initial capacity is exceeded, then more memory is automatically allocated in the size of the capacity increment.
  - Vector shoes = new Vector();
  - Vector orders = new Vector(100, 10);

Methods for Vectors

- Elements are created with the addElement(...) method:
  - Order missouri = new Order();
  - orders.addElement(missouri);
- The object missouri of type Order is automatically converted to an Object and added to Vector instance orders defined on the previous foil.
- There are methods for indexing vectors. Like arrays, the indexing is zero-based.
  - x = (Typeofx) v.elementAt(i);
  - v.setElementAt(x, i);
- The length of the Vector may also be obtained:
  - int size = v.size;

The Hashtable class

- This class is similar to a Perl associative array (or hash). It can store a set of key-value pairs, neither of which can be null.
  - Hashtable staff = new Hashtable();
  - Employee harry = new Employee("Harry Hacker");
  - staff.put("987-98-9996", harry);
- Values are retrieved by indexing with a key. Like Vectors, Hashtables only store objects of type Object, so you must cast the result:
  - steve = (Employee) staff.get("149-26-7355");
- If there is no entry, a null value is returned.
- Performance of the Hashtable can be affected by giving an initialCapacity and a loadFactor for reallocation.

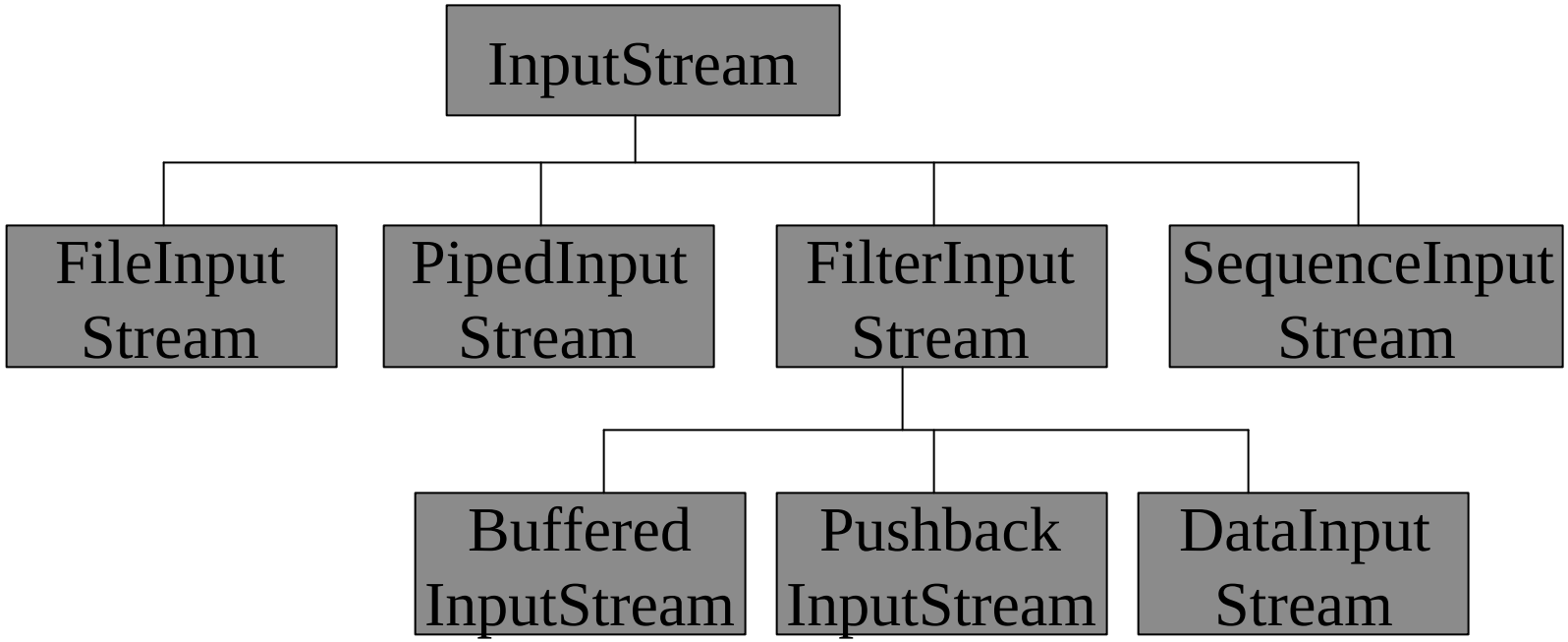
I/ O and the powerful Stream Zoo

I/ O Streams

- A stream is a sequence of bytes or characters.
- Stream sources and sinks include:
  - files
  - network connections
  - blocks of memory
  - threads
- That is, all types of streams are treated similarly.
- The most basic byte streams are InputStream and OutputStream. These classes have methods that can read or write a byte from or to a stream:
  - int read();
  - void write(int);
  - skip(long); available(); flush(); close();
- All of the above methods throw a possible IOException.
- The read() and write(int) methods "block" during transfer.

The Input Stream Zoo

- The subclasses of InputStream offer additional methods that write a byte stream in a more structured way or provide other functionality.
  - For example, to open a byte stream to an input file, use:
  - FileInputStream s = new FileInputStream("/ usr/ gcf/ file");

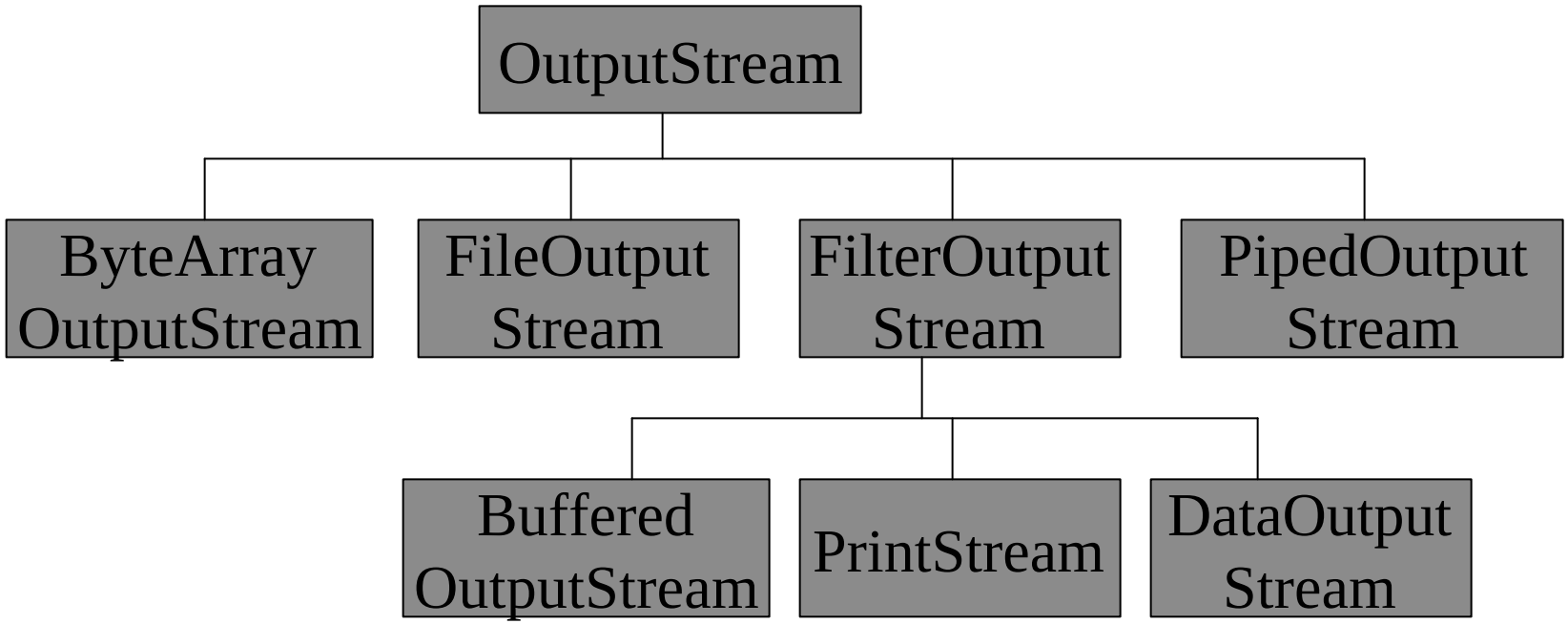


FilterInputStreams

- Subclasses of FilterInputStream are used to convert a raw InputStream to one with added value. You can define your own filters but useful ones are already provided in java.io:
  - BufferedInputStream -- establishes an intermediate buffer to service the stream
  - DataInputStream -- has methods to input other data types besides bytes (char, double, boolean, etc.)
  - PushbackInputStream -- allows one to "unread" a byte and put it back in the input stream
- These streams may be "chained" for added functionality:
  - DataInputStream in =
  - new DataInputStream (new FileInputStream(file));
  - or
  - BufferedInputStream in =
  - new BufferedInputStream (new FileInputStream(file));
  - where file is a filename string.

The Output Stream Zoo

- The subclasses of OutputStream are analogous to those of InputStream.
  - For example, to open a byte stream to an output file, use:
  - FileOutputStream s = new FileOutputStream("/ usr/ gcf/ file");



FilterOutputStreams

- DataOutputStream and BufferedOutputStream are two important FilterOutputStreams.
- To open a data stream to an output file, use:
  - DataOutputStream out =
  - new DataOutputStream (
  - new FileOutputStream(filename));
- where filename is a filename string.
- Note that DataOutputStream has methods to write any primitive type.
  - To open a buffered output stream, use:
  - BufferedOutputStream out =
  - new BufferedOutputStream (
  - new FileOutputStream(filename));
- Only bytes may be written to a BufferedOutputStream.

Character Streams

- Java 1.1 introduced Reader and Writer classes for character streams, which are used to read/ write text files.
- To construct a character output stream, for example:
  - PrintWriter out =
  - new PrintWriter(
  - new OutputStreamWriter(
  - new FileOutputStream(filename)));
- The OutputStreamWriter constructor takes a byte stream and converts it to a character stream. As a shortcut, use
  - PrintWriter out =
  - new PrintWriter(
  - new FileWriter(filename));
  - where FileWriter is a subclass of OutputStreamWriter.

Buffered Text I/ O

- For buffered text output, use the character stream:
  - BufferedWriter out =
  - new BufferedWriter(
  - new OutputStreamWriter(
  - new FileOutputStream(filename)));
- Similarly, for buffered text input, use:
  - BufferedReader in =
  - new BufferedReader(
  - new InputStreamReader(
  - new FileInputStream(filename)));
- Optionally use the subclasses FileWriter and FileReader for brevity (as in the previous foil).
- Note that the BufferedReader class has a handy readLine() method for sequential text input.

A Monster Chain

- The buffered output construct in the previous foil is of limited use since BufferedWriter has so few output methods. Instead, use the "monster" chain:
  - PrintWriter out =
  - new PrintWriter(
  - new BufferedWriter(
  - new OutputStreamWriter(
  - new FileOutputStream(filename))));
  - which can be shortened somewhat by using FileWriter as shown earlier.
- The PrintWriter class defines print(...) and println(...) methods for all primitive types, which unlike other Reader/ Writer classes never throw exceptions.

Standard Input/ Output

- The System class in java.lang provides the "standard" IO streams System.in, System.out, and System.err.
- System.in is an instance of InputStream.
- System.out and System.err are instances of PrintStream.
- PrintStream is a subclass of FilterOutputStream, which itself is a subclass of OutputStream.
- PrintStream objects should not be instantiated; use other subclasses of FilterOutputStream for byte streams or PrintWriter objects for character streams.
- PrintStream and PrintWriter define methods print(...) and println(...), which output any primitive type:
  - System.out.println("Enter character: ");
  - int ch = System.in.read();
  - System.out.println((char) ch);

SequenceInputStream

- The constructor of SequenceInputStream takes a pair of InputStreams and concatenates them together:
  - SequenceInputStream in =
  - new SequenceInputStream (
  - new FileInputStream(file1),
  - new FileInputStream(file2));
- Alternatively, SequenceInputStream takes a Java Enumeration type:
  - SequenceInputStream in =
  - new SequenceInputStream (
  - new FileListEnumerator(args));
  - where args is an array of command-line arguments and FileListEnumerator is a class that implements the Enumeration interface.

The File Class

- The File class defines methods and variables that provide access to the underlying file system in a machine-independent way.
- For example, there are methods getParent() and getPath(), as well as boolean methods isDirectory() and isFile(), plus many more.
- A very handy method is the list() method, which returns a string array of directory contents:
  - File dir = new File("/ tmp");
  - if (dir.exists() && dir.isDirectory())
  - String directory[] = dir.list();
- Instances of class File may be used in lieu of filename strings in InputStream constructors.

The FileDialog Class

- The FileDialog class is part of the AWT, a child of Dialog, and allows applications with a window interface to allow the user to “browse” the file system to select a file and directory.
  - fd = new FileDialog(parent, “title”, FileDialog.LOAD)
  - where the parent is the frame that created this dialog box
  - “title” is the title of the window
  - FileDialog.LOAD specifies that the dialog box is to show files that can be loaded, one can also use FileDialog.SAVE to specify that the dialog box is to show files that can be written.
- FileDialog methods getFile() and getDirectory() can return the file and directory that the user selected or specified.

Random Access

- The RandomAccessFile class offers all the functionality of DataInputStream and DataOutputStream combined, plus additional capabilities.
- To open a random access file for reading, use:
  - RandomAccessFile in =
  - new RandomAccessFile(filename, "r");
  - Such a file may be accessed sequentially with
  - in.readLine();
  - or randomly by repositioning the file pointer:
  - in.seek(offset);
  - where offset is a byte offset into the random file. (Use "rw" for read/ write access.)
- Random access files have no inherent structure; the structure must be imposed by the programmer.

StreamTokenizer class

- This class converts an instance of a Reader class to a StreamTokenizer. There is a similar StringTokenizer class.
- It parses the characters into a stream of “tokens” separated by white space, and skipping comments.
- Types of tokens:
  - TT_WORD, TT_NUMBER, TT_EOL, TT_EOF
- A set of flags controls aspects of the parsing.
- A typical application
  - creates an instance of StreamTokenizer
  - sets the flags to control the parsing
  - repeatedly calls a method nexttoken(), which advances the stream by one token and returns the type of the token. Based on the token type, the value of the token is either in the variable sval (WORDS) or nval (NUMBERS). sval has type String and nval has type double.

More on StreamTokenizer

- StreamTokenizer flags are set by the methods
  - eolIsSignificant(boolean) - whether EOL is returned as a token or treated as white space
  - slashStarComments(boolean) - whether to recognize C style comments
  - slashSlashComments(boolean) - whether to recognize C++ style comments
  - lowerCaseMode(boolean) - whether to convert all WORD tokens to lower case
- Other parsing properties
  - parseNumbers() - if false, parses only WORDS - default is to parse both WORD and NUMBER tokens.
  - whitespaceChars(int,int) - specifies to use all chars in the range between the two ints as white space.
  - Other methods can specify word chars.

Object Serialization

- ObjectInputStream and ObjectOutputStream allows you to read and write objects from any class (not just primitive types).
- Java objects are serialized with writeObject() and deserialized with readObject(). For example:
  - Vector lines = new Vector(256); ...
  - try {
  - new ObjectOutputStream(
  - new GZIPOutputStream(
  - new FileOutputStream(filename)));
  - out.writeObject(lines);
  - out.close();
  - } catch (IOException e) { }
- Only objects of classes that implement Serializable (or Externalizable) can be serialized. (The Serializable interface defines no methods.)
- Object variables not to be serialized are called transient.