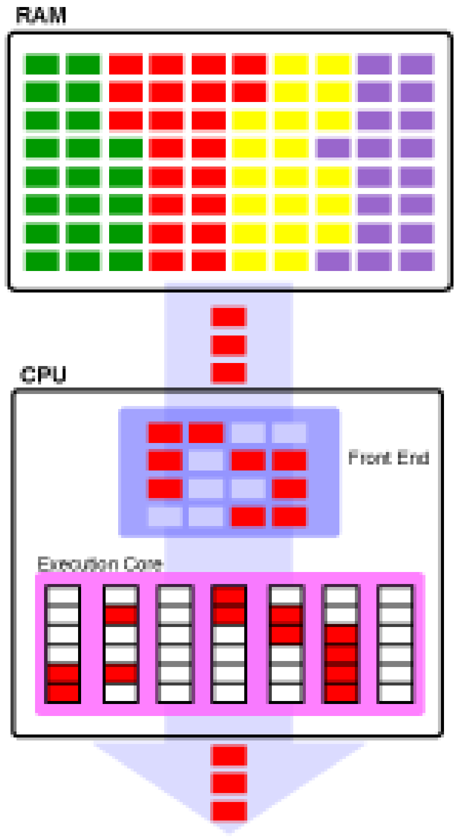
MultiThreading

HSBC Technology and Services

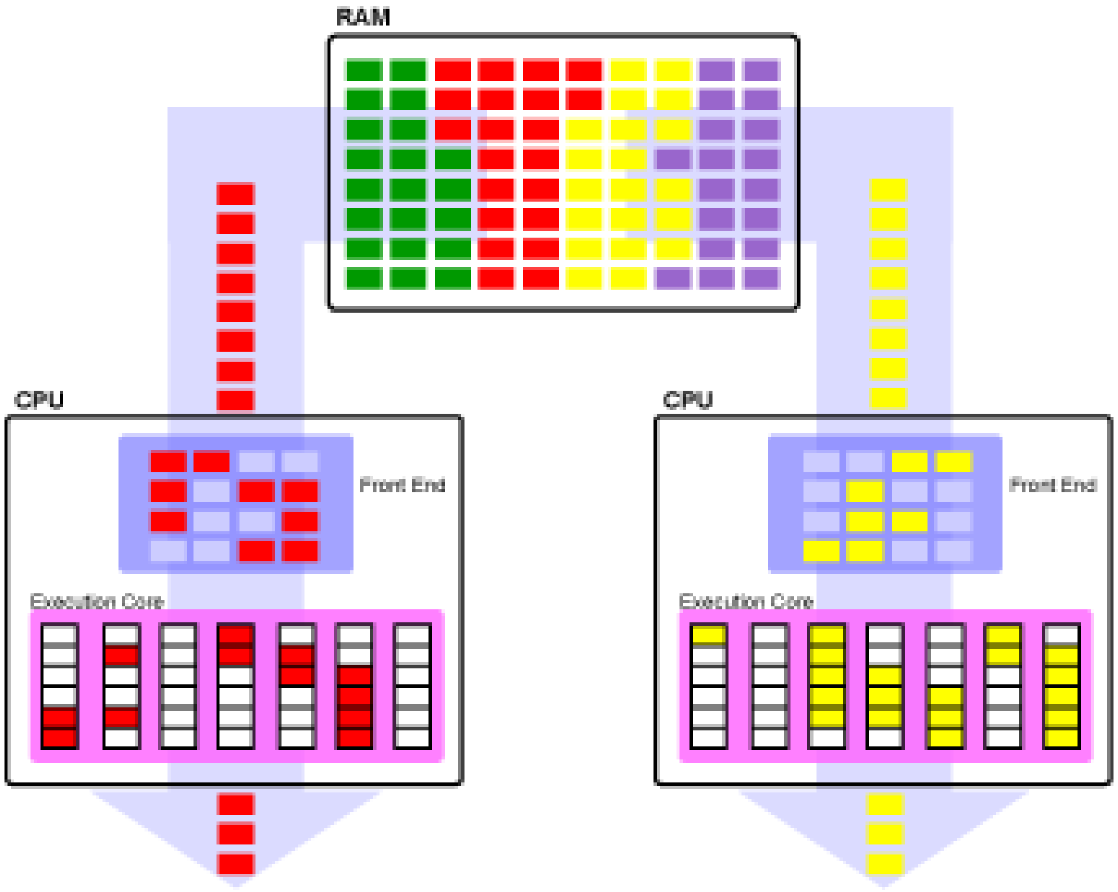
HSBC
The world's local bank

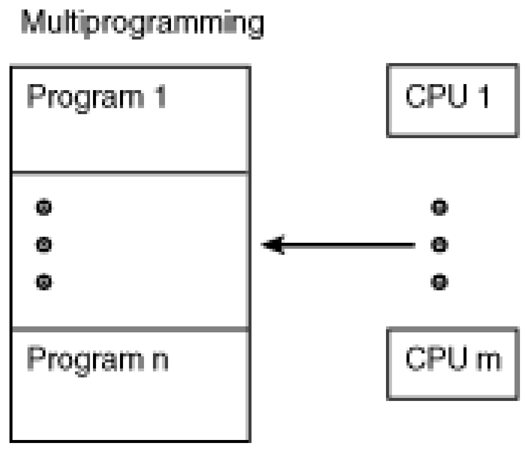
Multithreading

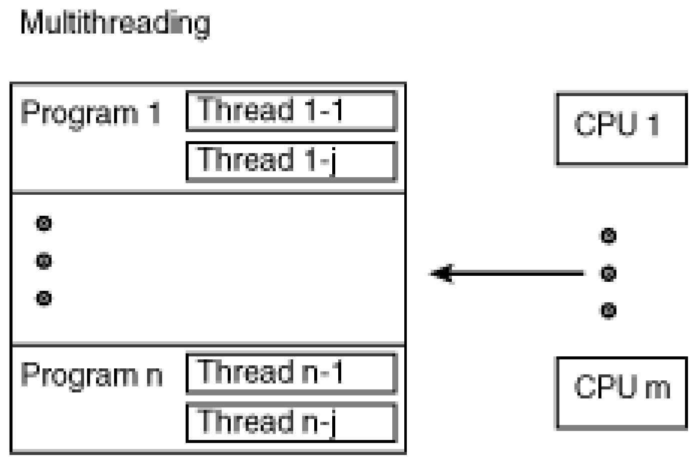
- A multithreaded program contains two or more parts that can run concurrently. Each part of such program is called Thread.
- Process based multitasking
- Thread based multitasking



}


RAM
CPU
Front End
Execution Core
CPU
Front End
Execution Core


Multiprogramming
Program 1
Program n
CPU 1
CPU m


Multithreading
Program 1
Thread 1-1
Thread 1-j
Program n
Thread n-1
Thread n-j
CPU 1
CPU m

Important terms

- A Thread is smallest unit of dispatch able code.
- Address space
- Context switching
- Interthread communication

Java Thread Model

- **Event loop with Polling**
- **Thread states**
- **Thread Priorities**
- **Synchronization**
- **Messaging**

Thread Objects

Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
- To abstract thread management from the rest of your application, pass the application's tasks to an *executor*.

- **Java Thread Class**
- **Runnable Interface**
    - getName
    - getPriority
    - isAlive
    - join
    - run
    - sleep
    - start

Pausing Execution with Sleep

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

The Main Thread

- **When a java program starts up, one thread begins running immediately. This is usually called the main thread of your program.it is important for two reasons**
  - It is the thread from which other “child” thread will be spawned.
  - It must be the last thread to finish execution.

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

Creating a Thread

- Implementing Runnable
  - ThreadDemo.java
- Extending Thread
  - ExtendThread.java

Creating Multiple Threads

- **MultiThreadDemo.java**

Using isAlive()

- **final boolean isAlive()**

Joins

- final void join() throws InterruptedException

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify

- DemoJoin.java

Thread Priorities

- MIN_PRIORITY = 0
- MAX_PRIORITY = 10
- NORM_PRIORITY = 5

- final void setPriority(int level)
- final int getPriority( )

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. Java provides unique language-level support for it.
- A monitor is an object that is used as mutually exclusive lock. Only one thread can own a monitor at a given time.

}

- **You can synchronize your code in either of two ways**
  - Using synchronized methods
    - Synch.java
  - The synchronized statement
    - Synch1.java

InterThread Communication

- **wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor.**
- **notify() wakes up the first thread that called wait() on the same object.**
- **notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.**

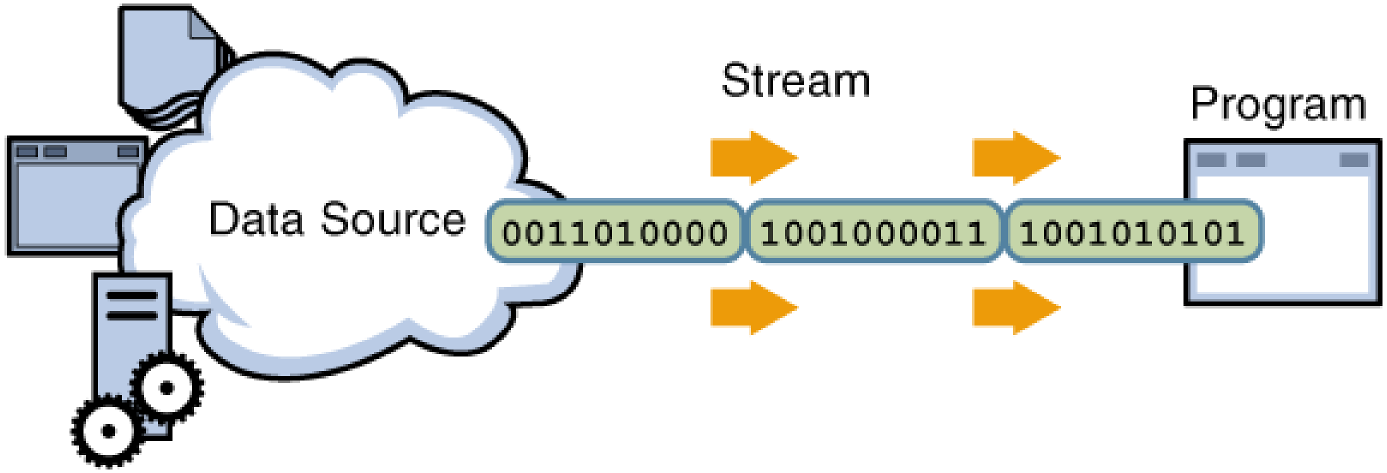
DeadLock

- **Deadlock occurs when two thread have circular dependency on a pair of synchronized objects.**

Input/Output

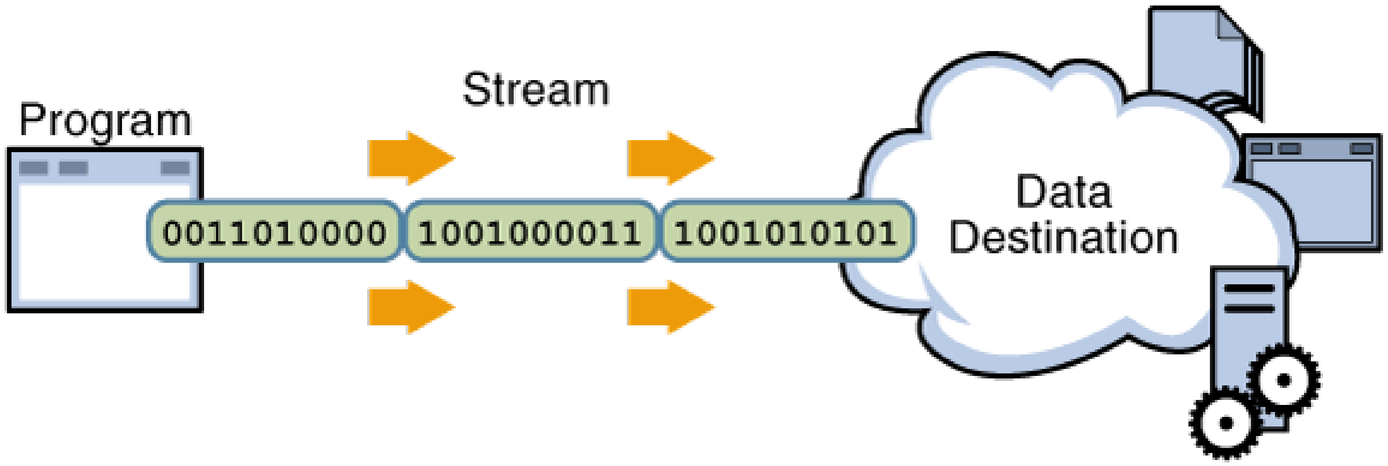
- Streams
  - An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays .
  - Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways

} A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time



Reading information into a program.

} A program uses an *output stream* to write data to a destination, one item at time:



Writing information from a program.

}

} Byte stream and Character Stream

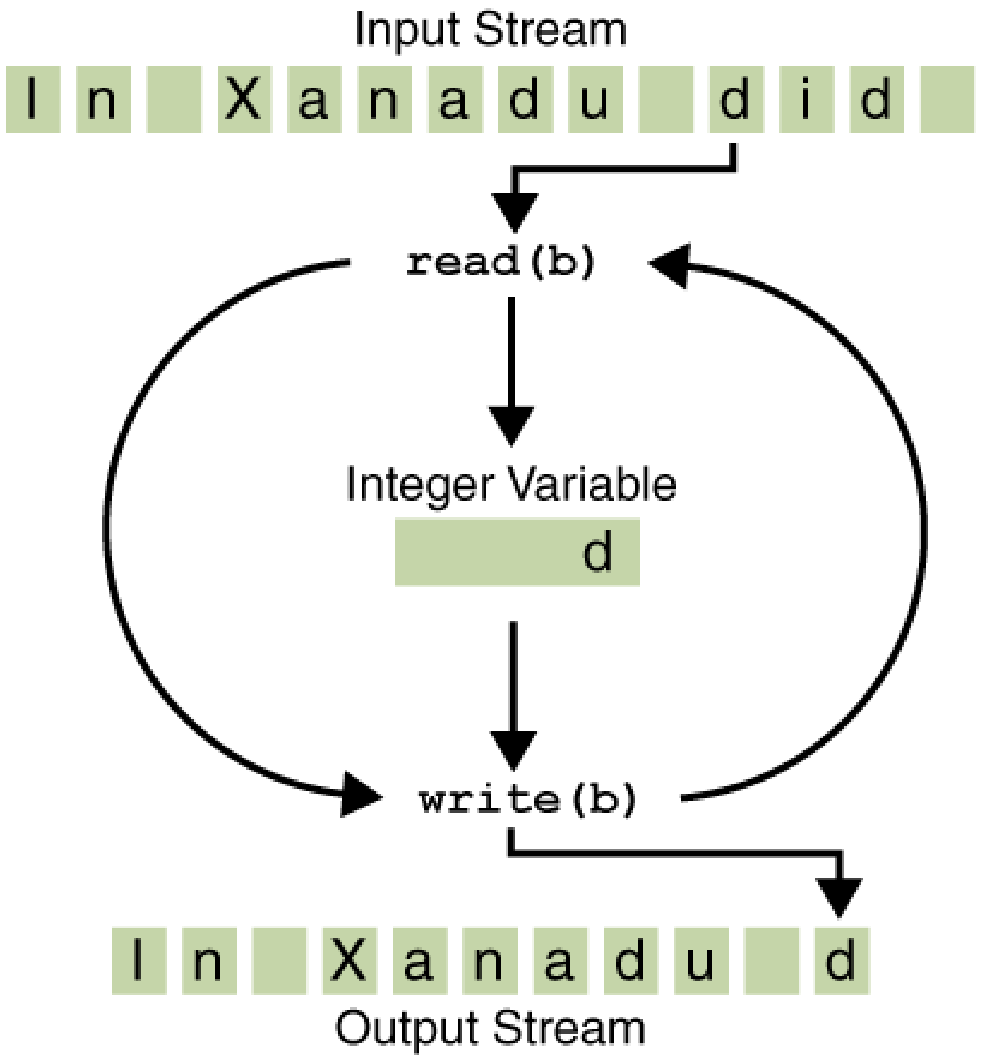
} Byte Stream classes

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

– **read() and write() function**

– **CopyBites.java**

}


Input Stream
I n X a n a d u d i d
read(b)
Integer Variable
d
write(b)
I n X a n a d u d
Output Stream

Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should be avoided. Since xanadu.txt contains character data, the best approach is to use character Stream, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

Character Streams

- Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**
- **read() and write() function**

CopyCharacters.java

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. **FileReader**, for example, uses **FileInputStream**, while **FileWriter** uses **FileOutputStream**

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating system

CopyLines.java

Buffered Streams

Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams

Scanning

Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators.

ScanXan.java

To use a different token separator, invoke useDelimiter(), specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

```
s.useDelimiter(",\\s*");
```

Predefined Streams

- java.lang pacakage contains a class called System.
  - System contains three predefined variables, **in, out,** and **err.** These fields are declared as public and static within System.

Reading Console Input

- *Java does not have a generalized console input method that parallels the standard C function **scanf()** or C++ input operators*
- *In java , console input is accomplished by reading from System.in.*
  - *BufferdReader(Reader inputReader)*
  - *InputStreamReader(InputStream inputStream)*

  - *BRRead.java*

Reading Strings

String readLine() throws IOException

- BRReadLines.java

File I/O

Streams don't support all the operations that are common with disk files. In this part of the lesson, we'll focus on non-stream file I/O. There are two topics:

- File is a class that helps you write platform-independent code that examines and manipulates files and directories.
- FileStuff.java
- Random access files support nonsequential access to disk file data.

Writing console output

- Console output is most easily accomplished with print() and println().another option is write()
  - Void write(int byteval) throws IOException
  - WriteDemo.java

The PrintWriter Class

- For real world programs ,the recommended method of writing to the console when using java is through a **PrintWriter** stream.
- **PrintWriter** defines several constructors
  - **PrintWriter(OutputStream *outputstream,* boolean *flushOnNewLine)***

Here **output stream** *is* an object of type OutputStream, and *flushOnNewLine* controls whether Java flushes the output stream every time a newline ('\n') character is output. If *flushOnNewLine* is true, flushing automatically takes place.

PrintWriterDemo.java

Reading Writing files

- Two of the most used stream classes are FileInputStream and FileOutputStream, which create byte stream linked to files.
- FileInputStream(String *FileName)* throws FileNotFoundException
- FileOutputStream(String *FileName)* throws FileNotFoundException

HSBC

The world's local bank

About HSBC Technology and Services

HSBC Technology and Services (HTS) is a pivotal part of the Group and seamlessly integrates technology platforms and operations with an aim to re-define customer experience and drive down unit cost of production. Its solutions connect people, devices and networks across the globe and combine domain expertise, process skills and technology to deliver unparalleled business value, thereby enabling HSBC to stay ahead of competition by addressing market changes quickly and developing profitable customer relationships.

Presenter's Contact Details:
Name: Lorem ipsum
Role: Lorem ipsum
Direct: +00 00 000
Email: loremipsum@hsbc.com

Name: Lorem ipsum
Role: Lorem ipsum
Direct: +00 00 000
Email: loremipsum@hsbc.com