The Java™ J2SE Gouge Card

© 2003 Pulp Free Press – All Rights Reserved
ISBN: 1-932504-06-0
by Rick Miller

Source Files

- End in `.java` extension
- Conventionally contain only one public class
- But can contain many classes
- `class` file generated for each class
- Can contain three top-level elements. If present they must appear in this order:
  - --package declaration
  - --import statements
  - --class definitions

-Example: MyClass.java source file

```
package com.pulpfreepress.gouge;

import java.util.*;

public class MyClass{
     public static void main(String[] args){
         System.out.println("Hello World!");
     }
}
```

Compiling Source Files

-Use javac command line tool
-Example:

```
javac MyClass.java
```

Running Java Applications

-Use java command line tool
-Example:

```
java MyClass
```

-Use -server argument to invoke HotSpot server compiler version
-Example:

```
java -server MyClass
```

Keywords

abstract	float	short
boolean	for	static
break	*goto* †	super
byte	if	switch
case	implements	synchronized
catch	import	this
char	instanceof	throw
class	int	throws
const †	interface	transient
continue	long	true
default	native	try
do	new	void
double	null	volatile
else	package	while
extends	private	
false	protected	
final	public	
finally	return	

† reserved

Identifiers

- Used to name variables, methods, classes or labels
- Cannot use reserved or keywords
- Must begin with a letter, dollar sign or underscore
- Subsequent characters may be letters, underscores, dollar signs or digits

Examples:

count	//legal
!count	//illegal: can't start with !
_4count	//legal
$5count	//legal
3_count	//illegal: can't start with digit

Primitive Data Types

Type	Size(bits)	Min	Max
boolean	1	n/a	n/a
char	16	0	2^{16}-1
byte	8	-2^{7}	2^{7}-1
short	16	-2^{15}	2^{15}-1
int	32	-2^{31}	2^{31}-1
long	64	-2^{63}	2^{63}-1
float	32	+- 1.4E-45	+-3.4E+38
double	64	+-4.9E-324	+-1.7E+308

Wrapper Classes

- Each of the primitive data types has a corresponding wrapper class that has lots of neat functionality. The following tables lists the primitive types and their wrapper classes:

boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

-Each wrapper class provides static methods to perform helpful conversions.
Example:

```
// convert string to int with static method
String s = "25";
int i = Integer.parseInt(s);
// create Integer object
Integer I = new Iteger(15)
//convert to double using instance method
double d = I.doubleValue() //d == 15.0
```

Literals

Char literals
- Can be expressed by enclosing the desired character in single quotes:

```
char ch = 'g';
```

- Can use 4-digit hexadecimal unicode preceded by \u:

```
char ch = '\u2216';
```

- The following escape sequences are supported by Java:

'\n'	Newline
'\r'	Return
'\t'	Tab
'\b'	Backspace
'\f'	Formfeed
'\''	Single Quote
'\"'	Double Quote
'\\'	Backslash

Integral literals
- Can be expressed in decimal, octal, or hexadecimal
- Default is decimal
- Indicate octal prefix number with 0 (zero)
- Indicate hex prefix number with 0x or 0X
- Hex digits may be upper or lower case

Floating-point literals
A number will be interpreted as a floating-point literal if it contains one of the following:

A decimal point:	2490.0
The letter E or e:	2.49E+3
The suffix F or f:	2490.0f
The suffix D or d indicating 64-bit double literal:	2490.0D

String literals
- Is a string of chars enclosed in double quotes:

```
String s = "Strings are in double quotes."
```

Arrays

- An array is an ordered, homogeneous collection of primitive data types, references or other arrays. To use arrays remember three steps: declare, construct, initialize.

Declaring Arrays
- The following statement declares a reference to an array of integers:

```
int int_array[];
```

- Another way:

```
int[] int-array;
```

Constructing Arrays
- Next, construct the array object using the keyword new.
- Reserves memory in the amount determined by the size value you use.
- The following code constructs the integer array declared above and reserves enough memory to hold 100 integer values:

```
int_array = new int[100];
```

- The declaration and construction could be combined into one statement:

```
int int_array[] = new int[100];
```

- The size of the array can be set with a variable:

```
int size = 100;
int int_array[] = new int[size];
```

Initializing Arrays
- When arrays are constructed elements are initialized to values according to the following table:

byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
object reference	null

- You can combine declaration, construction, and initialization into one statement:

```
int int_array[] = {1,2,3,4,5,6,...,100}
```

- In this case, the size of the array is determined by the number of initializers contained within the brackets.

Accessing Array Elements
- Individual array elements can be accessed via the subscript operator [] and an integral value denoting the offset.
- The first element of an array has an offset of 0
- The following statement would set the variable temp to the value of the first element in int_array:

```
int temp = int_array[0];
```

- A 100 element array such as int_array would have elements numbered 0 - 99

Determining Array Length
- Java arrays have an attribute called length
- Appending length to the array name using a "." will yield the array's length.
- The following statement will set the variable temp to the value 100:

```
int temp = int_array.length;
```

Another Use
- The length attribute is a handy way to manipulate arrays without apriori knowledge of the array's actual length. The following code prints the contents of int_array to the java console:

```
for(int i = 0; i<int_array.length; i++)
    System.out.println(int_array[i]);
```

Multidimensional Arrays
- Implemented as an array of arrays
- The following code declares a two dimensional integer array:

```
int two_dim_array[][];
```

- Construction:

```
two_dim_array = new int[5][5];
```

- Initialization:

Elements initialized according to type as discussed above. In this case all elements are of type int and are initialized to zero.

- Combining declaration, construction, and initialization example:

```
int two_dim_array[][] = {
                        {1,2,3,4,5},
                        {1,2,3,4,5},
                        {1,2,3,4,5},
                        {1,2,3,4,5},
                        {1,2,3,4,5}};
```

- Java arrays can be triangular
- The following application creates multidimensional array in one statement combining declaration, construction, and initialization. Notice the use of the length attribute to determine the length of subarrays:

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

  int int_array[][] = {{1}, {1,2}, {1,2,3}};

  for(int i = 0; i<int_array.length; i++)
  for(int j = 0; j<int_array[i].length; j++)
      System.out.println( int_array[i][j] );
  }
}
```

Visibility Modifiers

- Used to control access to variables, methods and classes

Visibility Modifier	Effects
public	Visible everywhere
protected	Visible to owning class, subclasses & all classes in same package.
package	Default access when keywords public, private or protected are not used. More restrictive than protected but less than private. Visible to owning class, classes and subclasses in same package
private	Visible only to the owning class.

Operators & Operator Precedence

Precedence.	Operator	Operand Type	Association	Operation Performed
1	++	arithmetic	R	unary pre/post increment
	--	arithmetic	R	unary pre/post decrement
	+	arithmetic	R	unary plus
	-	arithmetic	R	unary minus
	~	integral	R	bitwise complement
	!	boolean	R	logical complement
	(type)	any	L	cast
2	*	arithmetic	L	multiplication
	/	arithmetic	L	division
	%	arithmetic	L	modulus
3	+	arithmetic	L	addition
	-	arithmetic	L	subtraction
4	<<	integral	L	left shift
	>>	integral	L	right shift w/sign extend
	>>>	integral	L	right shift w/zero extend
5	<	arithmetic	L	less than
	<=	arithmetic	L	less than or equal to
	>	arithmetic	L	greater than
	>=	arithmetic	L	greater than or equal to
	instanceof	object,type	L	type comparison
6	==	primitive	L	equal to (same value)
	!=	primitive	L	not equal (different value)
	==	object	L	equal (same object)
	!=	object	L	not equal (different object)
7	&	integral	L	bitwise AND
	&	boolean	L	boolean AND
8	^	integral	L	bitwise XOR
	^	boolean	L	boolean XOR
9	|	integral	L	bitwise OR
	|	boolean	L	boolean OR
10	&&	boolean	L	conditional AND
11	||	boolean	L	conditional OR
12	?:	boolean, any, any	R	conditional ternary operator
13	=	variable, any	R	assignment
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	variable, any	R	assignment w/operation

Other Modifiers

Modifier	Usage
final	used to prevent further inheritance, overriding, or to create constants
abstract	declares a class or method abstract
static	declares a method or attribute static
native	declares a method to be native
transient	declares an attribute to be not part of the persistent state of the object.
synchronized	ensures two threads cannot modify the class or object simultaneously.
volatile	used on fields that can be accessed by unsynchronized threads.

Comments

-There are three types of comments:

```
// Single line
/* C block style */
/** Java Doc */
```

-Use intelligent identifier names to make code as self-commenting as possible. Use comments to explain salient points, introduce different sections of your program, explain the purpose of function, etc.

-Use single line comments sparingly as they add clutter.

```
/*
 *This is a C style block comment
 */
```

JavaDoc Comments

-Used to create HTML documentation pages

-Used in conjunction with javadoc tags like @author, @see, @param, etc.

-Place javadoc comments above class, method, and attribute declarations

-Use the javadoc tool to compile classes containing javadoc comments into HTML pages.

```
/**
* This is an example of a javadoc comment
*/
```

Control Structures

-Allow you to control statement execution sequence. Five primary control structures: for, if/else, while/do, do/while, & switch/case

for

-Used to perform or loop through one or more statements repeatedly. Idiomatic use:

```
for(int i=0; i<some_value; i++)
  statement;
```

-To execute more than one statement use braces:

```
for(int i=0; i<some_value; i++){
  statement_1;
  statement_2;
  statement_n;
}
```

while

-Used to perform or loop through one or more statements repeatedly. The while control structure places the conditional test at the top which means there is a possibility no statements will ever be executed. Idiomatic use:

```
while(some expression is ture)
  statement;
```

-To execute more than one statement use braces:

```
while(some expression is true){
  statement_1;
  statement_2;
  statement_n;
}
```

do/while

-Like the while structure except the body of the do statement will be executed at least once. Idiomatic use:

```
do
  statement;
while(some expression is true);
```

-To execute more than one statement add braces:

```
do{
  statement_1;
  statement_2;
  statement_n;
}while(some expression is true);
```

if/else

-Used to conditionally execute one or more statements.The else clause is optional. Idiomatic use:

```
if(some expression is true)
  statement;
```

-The else clause provides an optional execution path should the conditional test fail:

```
if(some expression is true)
  statement;
else
statement;
```

-To execute more than one statement add braces:

```
if(some expression is true){
  statement_1;
  statement_2;
  statement_n;
}else {
  statement_1;
  statement_2;
  statement_n;
}
```

-You can nest if/else statements. If you do, start the next if on the same line as the previous else:

```
if(some expression is true)
  statement;
else if(some expression is true)
  statement;
  else if(...)
```

-Nested if statements can be unsightly. There is an alternative.

switch

Used to provide multiple execution paths based upon the test of either a byte, short, char, or int value. Idiomatic usage assuming test_value is an int.

```
switch(test_value){
  case 1: statement_1;
          statement_2;
          statement_n;
          break;
  case n: statement_1;
          statement_n;
          break
  default: statement_1;
           break;
}
```

-Note that the use of braces to block statements following each case are optional.

-Always break out of a case unless you intend to allow execution to fall through to the next case.

-Always include a default case just to be safe.

Methods

-A method is a class-defined function containing zero, one, or more Java statements logically grouped together, callable by name, with zero, one, or more comma separated arguments, returning zero, or one value, to accomplish several purposes:

- Break the task of programming into small, manageable pieces
- Give objects behavior by manipulating either static class or object instance attributes.

-Unlike C++, there can be no stand-alone methods in Java. All methods must belong to a class. To "invoke" a method on an object means to call a class method via a class object reference. The following example declares a class Foo with function get_i():

```
public class Foo{
  private int i = 5;

  public int get_i(){return i;}
}
```

-To call get_i() you must first instantiate an object of type Foo:

```
Foo foo_obj = new Foo();
System.out.println(foo_obj.get_i());
```

-This results in the value 5 being printed to the Java console.

Method Overloading and Method Signatures

-A class method that shares the same name as another method in the same class but differing in either the number or types of arguments is overloaded.

-Altering the number or types of arguments in a method's argument list changes the method's signature. For example, a method f() could be overloaded in the following ways:

```
public int f(){}
private float f(int arg){}
protected double f(int arg1, int arg2){}
void f(double arg){};
```

-This also illustrates how visibility modifiers and return types can vary freely.

Overriding Methods

-Method overriding occurs in the context of inheritance where derived class methods share the same name as base class methods but require different behavior due to implementation requirements. An overriding method must have an identical argument list as well as the same return type:

```
public class Foo{
  public int f(){}
  public int f(int arg){}
}

public class Bar extends Foo{
  public int f(){} // f() is overriden
}

Foo foo_obj = new Foo();
Foo bar_obj = new Bar();

foo_obj.f(); //calls Foo's f()
bar_obj.f(); //calls Bar's f()
bar_obj.f(5); //calls Foo's f(int)
```

Polymorphic Behavior

-Polymorphic behavior is achieved by designing with abstract base classes

or interfaces and then substituting derived class objects where base class objects are expected. The following example illustrates polymorphic behavior by defining a Client class method that expects objects of type Foo, but substitutes derived class objects of type Bar_1 and Bar_2:

```
public class Client {
     public void print_message(Foo arg){
     arg.print_message();
     }
}
public abstract class Foo {
     public void print_message();
     }
}
```

-Next, extend Foo and implement the print_message() method differently in each Bar class:

```
public class Bar_1 extends Foo {
     public void print_message(){
     System.out.println("Bar_1 Object");
     }
}
public class Bar_2 extends Foo {
     public void print_message(){
     System.out.println("Bar_2 Object");
     }
}
```

- Instantiate one each of the Bar objects and one Client object:

```
Bar_1 b1 = new Bar_1();
Bar_2 b2 = new Bar_2();
Client c = new Client();
```

- Then call the Client print_message(Foo) method using the Bar objects:

```
c.print_message(b1);
c.print_message(b2);
```

Constructors

- A constructor is special method with the same name as the class in which it's defined.
- Do not specify a return type
- A constructor is called when an object is instantiated. The purpose of a constructor is to properly initialize object attributes to a safe state.
- They can be overloaded

The following class contains two constructors:

```
public class Foo {
     private int i;
     public Foo(){i = 3;}
     public Foo(int arg){i = arg;}
}
```

- Argument matching determines which constructor to call. The following code declares two Foo objects and calls each of the constructors:

```
Foo f1 = new Foo(); //default constructor,
                    //i set to 3
Foo f2 = new Foo(5); //i set to 5 using second
                     //constructor
```

Interfaces

- Allows "is-a" relationships to be implemented by providing a set of interface functions which must be implemented in classes wishing to be known by that interface.
- In Java, a class can extend the functionality of only one other class but can implement as many interfaces as desired. This fact causes many programmers to ask:

Q. What's the difference between abstract classes and interfaces?

A. The difference between abstract classes and interfaces is what one can contain and the other can't. An abstract class can contain implemented functions and attributes, some, or all of which can be inherited by derived classes.

Q. When should you choose to implement an abstract class as an interface instead?

A.The primary objective of inheriting the public interface of an abstract base class is to implement an "is-a" relationship and provide implementation specific behavior to each of the inherited methods. The same objective can, and should, be achieved via an interface.

It's helpful to think of interfaces as classes that exist solely to provide a public interface to derived classes.

Polymorphic Behavior

- Achieved with interfaces in the same way it's achieved with classes:

```
public class Client {
  public void print_message(Foo arg){
       arg.print_message();
       }
}
```

- Replace the words "abstract class" with the keyword "interface":

```
public interface Foo {
     public void print_message();
     }
}
```

- In the derived classes, replace the keyword "extends" with the keyword "implements":

```
public class Bar_1 implements Foo {
     public void print_message(){
        System.out.println("Bar_1 Object");
        }
}
public class Bar_2 implements Foo {
     public void print_message(){
       System.out.println("Bar_2 Object");
     }
}
```

- Instantiate one each of the Bar objects and one Client object:

```
Bar_1 b1 = new Bar_1();
Bar_2 b2 = new Bar_2();
Client c = new Client();
```

- Then call the Client print_message(Foo) method using the Bar objects:

```
c.print_message(b1);
c.print_message(b2);
```

Inheritance

- Achieved by either extending the functionality of a base class or by implementing an interface.
- Used to generalize class behavior. Define behavior common to all derived classes in the highest possible base class.
- The base class can be either an abstract class or a concrete class.
- Inherited member accessibility controlled via visibility modifiers public, protected, private, and package (the default when no visibility is specified).
- The rules:
  - -- You can only extend one class
  - -- You can implement many interfaces
- A fundamental concept of good object oriented design is to keep base class attributes private and only allow access to them via public base class methods.

Exception Handling

-Promotes good program design by forcing programmers to properly address possible error conditions. There are generally two things you can do with exceptions: Catch them or throw them.

Catching Exceptions

- When a method is defined to throw an exception, the method must be called within a try block and the exception handled in a catch block. The structure looks like this:

```
try{
     Some_object.exceptionThrowingMethod();
}
catch(Exception e){ }
```

-A try block can contain many statements.

- Although technically you don't have to respond to the thrown exception, just catch it, good programming practice, and your professional reputation, demands otherwise.
- The primary objective of catching exceptions is to return your program to a stable state by gracefully recovering from the error condition.
- You can catch many exceptions. For example:

```
try{
     Some_object.exceptionThrowingMethod();
}
catch(Exception_type_1 e){ //error handling code}
catch(Exception_type_2 e){ //error handling code}
catch(Exception_type_n e){ //eror handling code}
```

- The order in which you catch exceptions is important, and must proceed from the specific to the general.
- Try/catch blocks can be nested.

Throwing Exceptions

- Use the throws and throw keywords. The following method declares that it throws an IOException:

public void f() throws IOException{...}

- Inside the body of method f() would then be either a statement that has the possibility of throwing an exception upon execution and is not contained within a try/catch block, or, you gererate a new exception. Example:

```
public void f() throws IOException{
     IOException_Throwing_Statement;
     // or
     if(something_went_wrong)
     throw new IOException("Something's Wrong!");
}
```

- It's always a good idea to create and throw the exception on the same line.

Abstract Classes

- Serve to provide a public interface and functionality to derived classes.
- Differ from interfaces in that they can contain implemented functions and attributes.
- Cannot be instantiated

Polymorphic Behavior

- To achieve polymorphic behavior with abstract classes, design with abstract classes and substitute derived class objects.(*see Polymorphic Behavior, and Interfaces sections*)

Threads

-A thread is a distinct, controllable process that can be in one of several states of execution:

- -- Running
- -- Waiting
- -- Sleeping
- -- Suspended
- -- Blocked
- -- Ready
- -- Dead

- Start a thread by calling its start() method.
- Starting a tread causes its run() method to be scheduled for execution.
- Threads are implemented in two ways:

--Extend Thread class

-- Implement Runnable interface

- If you extend the Thread class you will need to supply a run method.

Example:

```
public class MyThread extends Thread{
   public void run(){
     System.out.println("Thread here!");
     }
}
```

- Create an instance of MyThread and call the start method:

```
MyThread thread_1 = new MyThread();
thread_1.start();
```

- If you implement the Runnable interface you will need to implement the run() method:

```
public class MyRunnable implements Runnable {
     public void run(){
       System.out.println("Runnable here!");
     }
}
```

- Then, create an instance of MyRunnable, followed by an instance of Thread, supplying the MyRunnable object as an argument to the Thread constructor:

```
MyRunnable runnable_1 = new MyRunnable();
Thread the_thread = new Thread(runnable_1);
```

- Now call the thread's start() method:

```
the_thread.start();
```

Thread Priority

- Threads have execution priority between 1 - 10.
- Default is 5 or the priority of the thread's parent process.
- Set a thread's priority with the setPriority() function.
- Determine a thread's priority with the getPriority() function.
- Methods yield() & sleep() are static and operate on the currently executing thread

Java 1.1 Event Handling

- Uses the event delegation paradigm.
- Components must register event listeners.
- A listener can be an object of any class that has implemented one or more of the listener interfaces. The following table lists the interfaces:

Interface	Interface Methods
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener	componentHidden(ComponentEvent) componetMoved(ComponentEvent) componentResized(ComponentEvent) ComponentShown(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent) componentRemoved(containerEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
ItemListener	itemStateChanged(ItemEvent)
KeyListener	keyPressed(keyEvent) keyReleased(keyEvent) KeyTyped(keyEvent)
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
TextListener	textValueChanged(TextEvent)

Interface	Interface Methods
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

- Add component event listeners by calling the corresponding add method listed below:

```
addActionListener(ActionListener)
addAdjustmentListener(AdjustmentListener)
addComponentListener(ComponentListener)
addContainerListener(ContainerListener)
addFocusListener(FocusListener)
addItemListener(ItemListener)
addKeyListener(KeyListener)
addMouseListener(MouseListener)
addMouseMotionListener(MouseMotionListener)
addTextListener(TextListener)
addWindowListener(WindowListener)
```

- There are three steps to implementing event handling.
  1. Implement the required listener interfaces
  2. Create an instance of the listener object
  3. Add the listener object to the component of interest
- The following code declares a new class that implements the ActionListener interface and registers itself with a Button object:

```
public class MyListener implements
      ActionListener {
   Button button1= new Button("Press Me");
   button1.addActionListener(this);
   public void actionPerformed(ActionEvent e){
     if(e.getActionCommand.equals("Press Me")){
      // put code here to handle action Press Me
   {
   // other methods that may use button omitted
}
```

- Components can handle their own events if extended and the processActionEvent(ActionEvent) method is overridden.
- Components can also be extended and implement a listener interface. For example, if you needed a Button to handle its own ActionEvents your code might look like this:

```
public class MyButton extends Button
                    implements ActionListener{
   public MyButton(String label){
     super(label);
     addActionListener(this);
   }
   public void ActionPerformed(ActionEvent e){
   // event processing code goes here
   }
}
```

Layout Managers

- Used to automatically position GUI components within container.
- Different containers have different default layout managers
- You can create your own layout managers

The following table lists the layout managers:

Layout Manager Name	Description
BorderLayout	Lays out components in North, South, East, West, and Center positions. It is the default layout manger for windows and dialogs.
BoxLayout	Used by Box container.
CardLayout	Components are displayed as large as the container.
FlowLayout	Components are flowed into the container like words on a page. It is the default layout manager for Panel.
GridBagLayout	Arranges components in a container grid comprising varying sized cells.
GridLayout	Components are uniformly sized and arranged in a grid.
OverlayLayout	Overlaps components. (Rarely used)
ScrollPaneLayout	Used by JScrollPane.
ViewPortLayout	Used by JViewPort

-Set the layout of a container by calling the setLayout() method.
-Example:

```
JPanel p1 = new JPanel();
p1.setLayout(new GridLayout(2,3,0,0));
```

Console I/O

Output
-To write text to the console use the System.out object's print() or println() methods
-Example:

```
System.out.println("Java is fun!");
```

Input
-To read text from the console use a System.in object to create an InputStreamReader object, which is then used to create a BufferedReader object.
-Example

```
import java.io.*;
InputStreamReader input_stream;
input_stream = new InputStreamReader(System.in);
BufferedReader console;
console = new BufferedReader(input_stream);
```

- Now, use the readLine() function to read a line of text from the console

```
console.readLine()
```

-To extract the first char create a String object and use the charAt() method
-Example:

```
String s = console.readLine();
char c = s.charAt(0);
```

conversions
-Use wrapper classes to perform conversions
-Example:

```
String s = console.readLine();
int i = 0;
try{
  i = Integer.parseInt(s)
}catch(NumberFormatException e){
    System.out.println("Not an integer!");
}
```

StringTokenizer Class

-Found in the java.util package
-Parses strings into tokens based on default or user-defined delimiters
-Has methods to manipulate tokens
-Example:

```
String s = "Java is fun!";
StringTokenizer st = new StringTokenizer(s);
System.out.println(st.countTokens());
while(st.hasMoreTokens()){
  System.out.print(st.nextToken()+" ");
}
```

File I/O (Text Files)

Output
-Use the File, FileWriter, and PrintWriter classes of the java.io package
-Example:

```
String filename; //initialized somewhere else
try{
    File f = new File(filename);
    FileWriter fw  = new FileWriter(f);
    PrintWriter pr = new PrintWriter(fw);
    for(int i=0; i<10; i++){
      pr.println("I love Java!");
    }
    pr.close(); //close and flush buffer
    }catch(IOException e){}
```

Input
- Use the File, FileReader, and BufferedReader classes of the java.io package
- Example:

```
String filename; //initialized somewhere else
try{
   File f = new File(filename);
   FileReader fr = new FileReader(f);
   BufferedReader br = new BufferedReader(fr);
   String line_in;
   while((line_in = br.readLine()) != null)
     System.out.println(line_in);
   br.close(); //close and flush buffer
  }catch(IOException e){}
```

Sockets (Client/Server Programming)

-Used by clients and servers to talk to each other
Client Responsibilities
-Clients need to open a socket to a server
Server Responsibilities
-Serves must listen on a specific port using a ServerSocket object and wait for incoming client connections by calling the accept() method.
-The accept() method blocks until an incoming connection is detected
-The accept() returns a Socket object
-Once the server has a socket it can talk to the client
-Example Server Code:

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

public class SimpleServer{
 public static void main(String[] args){
 try{
    ServerSocket ss = new ServerSocket(5557);
    Socket s = ss.accept();//wait here for connect
    //when connected create reader and writer
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStream-
Reader(is);
    BufferedReader br = new BufferedReader(isr);
    OutputStream os = s.getOutputStream();
    PrintWriter pw = new PrintWriter(os, true);
    //then service client requests
    String in_line;
    System.out.println("Ready to read stuff!");
    while((in_line = br.readLine()) != null ){
      System.out.println("From Client: " +
                         in_line);
       pw.println("From Server: " + in_line);
      }
   ss.close();
   s.close();
   }catch(IOException e){  e.toString();}
 }
}
```

Example Client Code:

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

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

    Socket s = new Socket("127.0.0.1",5557);
    InputStream is = s.getInputStream();
    InputStreamReader isr =
                      new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    OutputStream os = s.getOutputStream();
    PrintWriter pw =new PrintWriter(os, true);
    InputStreamReader console_reader;
    console_reader =
              new InputStreamReader(System.in);
    BufferedReader console;
    console = new BufferedReader(console_reader);
    String console_input;
    System.out.println("Ready to write stuff!");
    while((console_input = console.readLine())
                                    != null){
        pw.println(console_input);
        System.out.println(br.readLine());
    }
   s.close();
   }catch(IOException e){}
 }
}
```

Executable Jar Files

-Are compressed class archives that contain a manifest file that specifies the name of the main application class
-Can be run with the java tool using the -jar option
Creating
-Write your program as usual and compile to create class files
-Create a manifest file (ordinary text file) with a Main-class: line in it
-Create the jar file using the jar command line tool
-Example: MyClass.class & manifest.txt
-Here's the contents of manifest.txt:

```
Main-class: MyClass
```

-Using manifest.txt create the jar file with the jar command:

```
jar cmf manifest.txt MyClass.jar MyClass
  c = create new archive
  m = manifest file to use
  f = name of the archive file to create
```

Note: you can create jars with one class file or whole package structures. The application class name given in the manifest file must be fully package qualified.
Executing
-Execute the jar file using the java tool with the -jar option
-Example - to execute MyClass.jar created above

```
java -jar MyClass.jar
```

Halting Program Execution

-Use System.exit()