UNIT - II

Introducing Classes

and

String Handling

String Handling

- The String Constructors
- String Length
- Special String Operations
- Character Extraction
- String Comparison
- Searching Strings
- Modifying Strings
- String Buffer
- concept of mutable and immutable string
- Command line arguments and
- basics of I/O operations – keyboard input using BufferedReader & Scanner classes.

String Handling

A Few Words About Strings – *A brief overview*:

- **String,** is not a simple type. Nor is it simply an array of characters. Rather, **String** defines an object.
- The **String** type is used to declare string variables. You can also declare arrays of strings.
- A quoted string constant can be assigned to a **String** variable.
- A variable of type **String** can be assigned to another variable of type **String.** You can use an object of type **String** as an argument to **println( ).**

```
String str = "this is a test";
System.out.println(str);
```

- *Here, **str** is an object of type **String**. It is assigned the string "this is a test". This string is displayed by the **println( )** statement.*
- **String** objects have many special features and attributes that make them quite powerful and easy to use.

String Handling

Overview:

- As is the case in most other programming languages, in Java a *string* is a *sequence of characters*. But, unlike many other languages that *implement strings as character arrays, Java implements* strings as ***objects*** of type **String.**
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- ***For example***, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.

String Handling

Overview: ...

- Once a **String** object has been created, you cannot change the characters that comprise that string. You can still perform all types of string operations.
- Each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged(Immutable).
- Immutable strings can be implemented more efficiently than changeable ones(Mutable).
- Java provides two options: **StringBuffer** and **StringBuilder.** Both hold strings that can be modified after they are created.

String Handling

Overview: ...

- The **String, StringBuffer,** and **StringBuilder** classes are defined in ***java.lang,*** they are available to all programs automatically. All are declared **final**, (none of these classes may be sub-classed)
- **One last point:** The strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

String Handling

Overview: ...

- String is a sequence of characters.
- Java implements strings as objects of type **String.**
- It belongs to **java.lang (*java.lang.String*)**
- Once a String object is created, it is not possible to change the characters that comprise the string.
- When a modifiable string is needed, java provides two options:
  - ***java.lang.StringBuffer***
  - ***java.lang.StringBuilder***

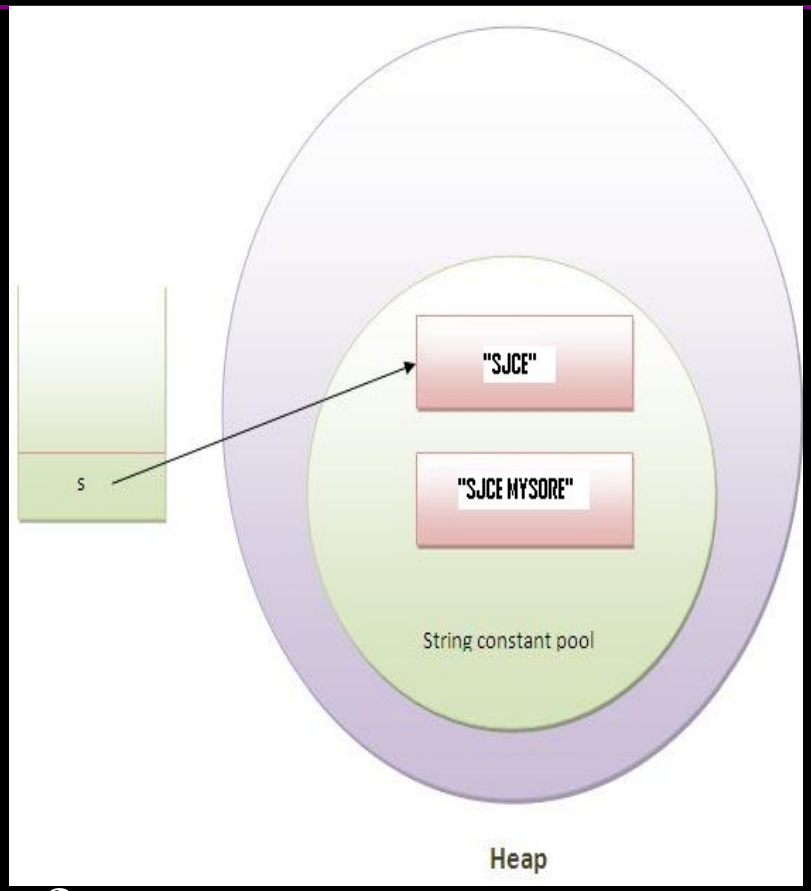
String Handling

Overview: ...

```
class Simple{
    public static void main(String args[]){
        String s = "SJCE";
        s.concat("  MYSORE");

        // s = s.concat("  MYSORE");

        System.out.println(s);
    }
}
```



Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "SJCE". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

String Handling

The String Constructors:

- The **String** class supports several constructors.

- *To create an empty* **String**, you call the default constructor.
  Eg: String s = new String(); // instance with no characters

- To create strings that have initial values by an array of characters, **String(char chars[ ])**
  Eg:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

  •Specify a sub-range of a character array as an

String Handling

The String Constructors: ...

- specify a sub-range...

Eg: char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);

- To construct a **String** object that contains the same character sequence as another **String** object using, **String(String strObj)**

```
// Construct one String from another.
class MakeString {
public static void main(String args[]) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);
    System.out.println(s1);
    System.out.println(s2);
}
```

String Handling

The String Constructors: ...

- Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
- Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

  String(byte asciiChars[ ])

  String(byte asciiChars[ ], int startIndex, int numChars)
- Here, *asciiChars* specifies the array of bytes. The second form allows you to specify a sub-range.
- In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

String Handling

The String Constructors: ...

- 8-bit ASCII strings....

```
// Construct string from subset of char array.
class SubStringCons {
      public static void main(String args[]) {
            byte ascii[] = {65, 66, 67, 68, 69, 70 };
            String s1 = new String(ascii);
            System.out.println(s1);
            String s2 = new String(ascii, 2, 3);
            System.out.println(s2);
      }
}
```

NOTE *:The contents of the array are copied whenever you create a* ***String*** *object from an array. If you modify the contents of the array after you have created the string, the* ***String*** *will be unchanged.*

You can construct a **String** from a **StringBuffer** by using the constructor, **String(StringBuffer *strBufObj)***

String Handling

The String Constructors: ...

- To create an empty **String**, you call the default constructor, **String();**
  String s = new String();
- To create strings that have initial values by an array of characters,
  String(char chars[ ])
- To specify a sub-range of a character array as an initializer using,
  String(char chars[ ], int startIndex, int numChars)
- To construct a String object that contains the same character sequence as another String object using,
  String(String strObj)
- The String class provides constructors that initialize a string when given a byte array. Their forms are shown here: **String(byte asciiChars[ ])**
  String(byte asciiChars[ ], int startIndex, int numChars)
- To construct a **String** from a **StringBuffer** by using the constructor,
  String(StringBuffer *strBufObj)*

String Handling

String Length:

- The length of a string is the number of characters that it contains.
- To obtain this value, call the **length( )** method, shown here:

int length( )

Eg:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length()); // What is the Size?
```

String Handling

Special String Operations:

- Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- Operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation.
- There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Handling

Special String Operations: ...

String Literals: Explicitly create a **String** instance using a string literal. Thus, you can use a string literal to initialize a **String** object.

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object.

```
System.out.println("abc".length());
```

String Handling

Special String Operations: ...

String Concatenation: In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations.

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

String Handling

Special String Operations: ...

String Concatenation: ...

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to c

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

String Handling

Special String Operations: ...

String Concatenation with other Data Types:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

- The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of **String.**
- Be careful when you mix other types of operations with string concatenation expressions.

```
String s = "four: " + 2 + 2;
System.out.println(s); // Output: ...
```

String s = "four: " + (2 + 2); // Output: ...

String Handling

Special String Operations: ...

String Conversion and toString( ):

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**.
- **valueOf( )** is overloaded for all the simple types and for type **Object**.
- *For the simple types,* ***valueOf( )*** *returns a string that contains the human-readable equivalent of the value with which it is called. For objects,* ***valueOf( )*** *calls the* ***toString( )*** *method on the object.*
- Every class implements **toString( )** because it is defined by **Object.**
- For most important classes that you create, you will want to

String Handling

Special String Operations: ...

String Conversion and toString( ): ....

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  public String toString() {
    return "Dimensions are " + width + " by " +
           depth + " by " + height + ".";
  }
}
```

```
class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

Box's toString( ) method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( ).**

String Handling

Character Extraction:

- The **String** class provides a number of ways in which characters can be extracted from a **String** object.

- **charAt( ):** To extract a single character from a **String,** you can refer directly to an individual character.

char charAt(int *where)*

Eg: char ch; ch = "abc".charAt(1); // b is extracted

- **getChars( ):** To extract more than one character at a time.

void getChars(int *sourceStart, int sourceEnd,*
char target[ ], int targetStart)

String Handling

Character Extraction: ...

- **getChars( ): ...**

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

- **getBytes( ) :** Alternative to **getChars( )** that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform.

byte[ ] getBytes( )

- most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. **Eg:** Most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

String Handling

Character Extraction: ...

- **toCharArray( )**: Converts all the characters in a **String** object into a character array, it returns an array of characters for the entire string. **char[ ] toCharArray( )**
- This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

String Handling

String Comparison:

- The **String** class includes several methods that compare strings or substrings within strings.

- **equals( ) and equalsIgnoreCase( ):** To compare two strings for equality.

  - **equals( )** //case-sensitive
    boolean equals(Object str)
  - **equalsIgnoreCase( )** //ignores case differences
    boolean equalsIgnoreCase(String str)

String Handling

String Comparison: ...

- equals( ) and equalsIgnoreCase( ): ...

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

String Handling

String Comparison: ...

- **regionMatches( ):**

boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

- For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars.
- In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

String Handling

String Comparison: ...

- **startsWith( ) and endsWith( ):** Specialized forms of **regionMatches( ).** These methods determines whether a given String begins/ends with a specified string.

  boolean startsWith(String str)

  boolean endsWith(String str)

  boolean startsWith(String str, int startIndex)

  Eg:

  "Foobar".endsWith("bar")

  "Foobar".startsWith("Foo")

  "Foobar".startsWith("bar", 3)

String Handling

String Comparison: ...

equals( ) Versus ==( )

- The ***equals( )*** method compares the characters inside a String object.
- The == operator compares two object references to see whether they refer to the same instance.

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

String Handling

String Comparison: ...

- **compareTo( ):** It is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to, or greater than the next. A string* is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.

int compareTo(String str)

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

String Handling

String Comparison: ...

- **compareTo( ): *An Exmaple***

```
// A bubble sort for Strings.
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]);
    }
  }
}
```

String Handling

Searching Strings:

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
  - **indexOf( )** Searches for the first occurrence of a character or substring.
  - **lastIndexOf( )** Searches for the last occurrence of a character or substring.
- These two methods are overloaded in several different ways.
- *In all cases, the methods return the index at which the character or substring was found, or –1 on failure.*

String Handling

Searching Strings: ...

- To search for the first/last occurrence of a character, use
  int indexOf(int *ch)*
  int lastIndexOf(int *ch)*
- To search for the first/last occurrence of a substring, use
  int indexOf(String *str)*
  int lastIndexOf(String *str)*
- You can specify a starting point for the search using
  int indexOf(int *ch, int startIndex)*
  int lastIndexOf(int *ch, int startIndex)*
  int indexOf(String *str, int startIndex)*
  int lastIndexOf(String *str, int startIndex)*
- Here, *startIndex specifies the index at which point the search begins. For **indexOf( )**, the* search runs from *startIndex to the end of the string. For **lastIndexOf( ),** the search runs from startIndex to zero.*

String Handling

Searching Strings: ...*An Example*

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

String Handling

Modifying Strings:

- Because **String** objects are immutable, whenever you want to modify a **String,** you must either copy it into a **StringBuffer** or **StringBuilder,** or use one of the following **String** methods, Which will construct a new copy of the string with your modifications complete.
- **substring( )**: To extract a substring use, **substring( ).**
- It has two forms:

  String substring(int *startIndex)*

  String substring(int *startIndex, int endIndex)*
- Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

String Handling

Modifying Strings: ... The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);

  }
}
```

String Handling

Modifying Strings: ...

- **concat( )**: To concatenate two stings use, **String substring( ),** performs the same function as **+.**

Eg:

```
String s1 = "one";
String s2 = s1.concat("two");   // String s2 = s1 + "two";
```

- **replace( )**: To replaces all occurrences of one character in the invoking string with another character.

String replace(char *original, char replacement)*

Eg: String str = "NEW".replace('E', 'O');

- The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original, CharSequence replacement)*

- This form was added by J2SE 5.

String Handling

Modifying Strings: ...

- **trim( ):** returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

String trim( )

Eg: String s = " Hello World ".trim();

String Handling

StringBuffer:

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- **String** represents fixed-length, immutable character sequences. *In contrast,*
- **StringBuffer** represents growable and writeable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for

String Handling

StringBuffer: ... StringBuffer Constructors

StringBuffer defines these four constructors:

- StringBuffer( ): Reserves room for 16 characters without reallocation.
- StringBuffer(int *size):* Accepts an integer argument that explicitly sets the size of the buffer.
- StringBuffer(String *str):* Accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
- StringBuffer(CharSequence *chars):* Creates an object that contains the character sequence contained in *chars*.
- **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take

String Handling

StringBuffer: ...

- **length( ) and capacity( ):** The current length of a **StringBuffer** can be found via the *length( )* method, while the total allocated capacity can be found through the *capacity( )* method. **int length( )** **int capacity( )**

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string **"Hello"** when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

String Handling

StringBuffer: ... ***Some useful methods***

- **charAt( ) and setCharAt( )**
  char charAt(int *where)*
  void setCharAt(int *where, char ch)*
- **getChars( )**
  void getChars(int sourceStart, int sourceEnd,
  char target[ ], int targetStart)
- **append( )**
  StringBuffer append(String str)
  StringBuffer append(int num)
  StringBuffer append(Object obj)

String Handling

StringBuffer: ... ***Some useful methods***

- **insert( )**
  StringBuffer insert(int index, String str)
  StringBuffer insert(int index, char ch)
  StringBuffer insert(int index, Object obj)
- **reverse( ):** StringBuffer reverse( )
- **delete( ) and deleteCharAt( )**
  StringBuffer delete(int startIndex, int endIndex)
  StringBuffer deleteCharAt(int loc)
  - **replace( ):**
    StringBuffer replace(int startIndex, int endIndex, String str)
- **replace( ):** String substring(int startIndex)
  String substring(int startIndex, int endIndex)

String Handling

StringBuilder:

- J2SE 5 adds a new string class to Java's already powerful string handling capabilities, called **StringBuilder.**
- It is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe.
- The advantage of **StringBuilder** is faster performance.
- However, in cases in which you are using multithreading, you must use **StringBuffer** rather than **StringBuilder.**

String Handling

Concept of Mutable and Immutable String:

- Java's String is designed to be ***immutable*** *i.e*, once a String is constructed, its contents cannot be modified.

- The Strings within objects of type **String** are unchangeable means the content of the **String** instance cannot be changes after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

String Handling

Concept of Mutable and Immutable String:...

- The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters. (*Mutable*)

- The ***String, StringBuffer*** and ***StringBuilder*** classes are defined in ***java.lang***

String Handling

Command -line arguments:

- Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments to* ***main( ).***
- It is the information that directly follows the program's name on the command line when it is executed.
- To access the command-line arguments inside a Java program is quite easy - *they are stored as strings in a* ***String*** *array passed to the args parameter of* ***main( ).*** *The first command-line argument is stored at* ***args[0]****, the second at* ***args[1]****, and so on.*
- ***REMEMBER*** *All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.*

String Handling

Command -line arguments: ...*Example*

```
// Display all command-line arguments.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " + args[i]);
    }
}
```

javac CommandLine.java
java CommandLine this is a test 100 -1

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class***

- In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

- Here, *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*)

String Handling

Basics of I/O operations: Keyboard input using *BufferedReader class...*

- Because **System.in** refers to an object of type **InputStream**, it can be used for *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

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class...***

int read( ) throws IOException...*Example*

```
// 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');
        }
}
```

String Handling

Basics of I/O operations: Keyboard input using ***BufferedReader class...***

Declaration: **public class BufferedReader extends Reader**

Class constructors

BufferedReader(Reader in) : Create a new BufferedReader that will read from the specified subordinate stream with a default buffer size of 8192 chars.

BufferedReader(Reader in, int size) : Create a new BufferedReader that will read from the specified subordinate stream with a buffer size that is specified by the caller.

String Handling

Basics of I/O operations: Keyboard input using *BufferedReader class...*

Method Summary	
void	close() This method closes the underlying stream and frees any associated resources.
void	mark(int readLimit) Mark a position in the input to which the stream can be "reset" by calling the reset() method.
boolean	markSupported() Returns true to indicate that this class supports mark/reset functionality.
int	read() Reads an char from the input stream and returns it as an int in the range of 0-65535.
int	read(char[] buf, int offset, int count) This method read chars from a stream and stores them into a caller supplied buffer.
String	readLine() This method reads a single line of text from the input stream, returning it as a String.
boolean	ready() This method determines whether or not a stream is ready to be read.
void	reset() Reset the stream to the point where the mark() method was called.
long	skip(long count) This method skips the specified number of chars in the stream.

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class:***

- It is the complement of **Formatter,** reads formatted input and converts it into its binary form.
- Read all types of numeric values, strings, and other types of data, whether it comes from a disk file, the keyboard, or another source.
- **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable interface** or **ReadableByteChannel.**

String Handling

Basics of I/O operations:

Keyboard input using *Scanner class*:...

The Scanner Constructors

- **Scanner** defines the constructors, it can be created for a **String,** an **InputStream,** a **File,** or any object that implements the **Readable** or **ReadableByteChannel** interfaces.
- **Eg:** The following sequence creates a **Scanner** that reads the file Test.txt:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

String Handling

Basics of I/O operations:

Keyboard input using *Scanner class*:...

The Scanner Constructors...

Method	Description
Scanner(File *from*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(File *from*, String *charset*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(InputStream *from*)	Creates a **Scanner** that uses the stream specified by *from* as a source for input.
Scanner(InputStream *from*, String *charset*)	Creates a **Scanner** that uses the stream specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(Readable *from*)	Creates a **Scanner** that uses the **Readable** object specified by *from* as a source for input.
Scanner (ReadableByteChannel *from*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* as a source for input.
Scanner(ReadableByteChannel *from*, String *charset*)	Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* with the encoding specified by *charset* as a source for input.
Scanner(String *from*)	Creates a **Scanner** that uses the string specified by *from* as a source for input.

String Handling

Basics of I/O operations:

Keyboard input using ***Scanner class*:...**

Scanning Basics

- Once you have created a **Scanner,** it is a simple matter to use it to read formatted input. It reads *tokens from the underlying source that you specified when the* **Scanner** was created.
- As it relates to **Scanner**, a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default.

String Handling

Basics of I/O operations:

Keyboard input using *Scanner class*:...

Scanning Basics...

In general, to use **Scanner,** follow this procedure:

.Determine if a specific type of input is available by calling one of **Scanner's hasNext*X*** *methods, where **X** is the type of data desired.*

.If input is available, read it by calling one of **Scanner's next*X*** *methods*.

.Repeat the process until input is exhausted.

The following sequence shows how to read a list of integers from the keyboard.

```
Scanner conin = new Scanner(System.in);
int i;
// Read a list of integers.
while(conin.hasNextInt()) {
        i = conin.nextInt();
        // ...
}
```

String Handling

Basics of I/O operations:

Keyboard input using *Scanner class*:...

Scanning Basics... The Scanner **hasNext** Methods

Method	Description
boolean hasNext( )	Returns **true** if another token of any type is available to be read. Returns **false** otherwise.
boolean hasNext(Pattern *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNext(String *pattern*)	Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise.
boolean hasNextBigDecimal( )	Returns **true** if a value that can be stored in a **BigDecimal** object is available to be read. Returns **false** otherwise.
boolean hasNextBigInteger( )	Returns **true** if a value that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextBigInteger(int *radix*)	Returns **true** if a value in the specified radix that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise.
boolean hasNextBoolean( )	Returns **true** if a **boolean** value is available to be read. Returns **false** otherwise.
boolean hasNextByte( )	Returns **true** if a **byte** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextByte(int *radix*)	Returns **true** if a **byte** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextDouble( )	Returns **true** if a **double** value is available to be read. Returns **false** otherwise.
boolean hasNextFloat( )	Returns **true** if a **float** value is available to be read. Returns **false** otherwise.
boolean hasNextInt( )	Returns **true** if an **int** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextInt(int *radix*)	Returns **true** if an **int** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextLine( )	Returns **true** if a line of input is available.
boolean hasNextLong( )	Returns **true** if a **long** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextLong(int *radix*)	Returns **true** if a **long** value in the specified radix is available to be read. Returns **false** otherwise.
boolean hasNextShort( )	Returns **true** if a **short** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.)
boolean hasNextShort(int *radix*)	Returns **true** if a **short** value in the specified radix is available to be read. Returns **false** otherwise.

String Handling

Basics of I/O operations:

Keyboard input using

Scanner class:...

Scanning Basics...

The Scanner **next** Methods

Method	Description
String next( )	Returns the next token of any type from the input source.
String next(Pattern *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
String next(String *pattern*)	Returns the next token that matches the pattern passed in *pattern* from the input source.
BigDecimal nextBigDecimal( )	Returns the next token as a **BigDecimal** object.
BigInteger nextBigInteger( )	Returns the next token as a **BigInteger** object. The default radix is used. (Unless changed, the default radix is 10.)
BigInteger nextBigInteger(int *radix*)	Returns the next token (using the specified radix) as a **BigInteger** object.
boolean nextBoolean( )	Returns the next token as a **boolean** value.
byte nextByte( )	Returns the next token as a **byte** value. The default radix is used. (Unless changed, the default radix is 10.)
byte nextByte(int *radix*)	Returns the next token (using the specified radix) as a **byte** value.
double nextDouble( )	Returns the next token as a **double** value.
float nextFloat( )	Returns the next token as a **float** value.
int nextInt( )	Returns the next token as an **int** value. The default radix is used. (Unless changed, the default radix is 10.)
int nextInt(int *radix*)	Returns the next token (using the specified radix) as an **int** value.
String nextLine( )	Returns the next line of input as a string.
long nextLong( )	Returns the next token as a **long** value. The default radix is used. (Unless changed, the default radix is 10.)
long nextLong(int *radix*)	Returns the next token (using the specified radix) as a **long** value.
short nextShort( )	Returns the next token as a **short** value. The default radix is used. (Unless changed, the default radix is 10.)
short nextShort(int *radix*)	Returns the next token (using the specified radix) as a **short** value.