S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.

Department of Computer Science & Engineering

Course: Programming in Java(18CS653)

Module 5: Enumerations, Type Wrappers, I/O, Applets, and Other Topics

Prof. Prasanna Patil
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Contents

- Enumerations: Enumerations, Type Wrappers
- I/O, Applets, and Other Topics: I/O Basics, Reading Console Input, Writing Console Output, The PrintWriter Class, Reading and Writing Files, Applet Fundamentals, The transient and volatile Modifiers, Using instanceof, strictfp, Native Methods, Using assert, Static Import, Invoking Overloaded Constructors Through this( )
- String Handling: The String Constructors, String Length, Special String Operations, Character Extraction, String Comparison, Searching Strings, Modifying a String, Data Conversion Using valueOf( ), Changing the Case of Characters Within a String , Additional String Methods, StringBuffer, StringBuilder.

String Handling

Introduction

- 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.

The String Constructors

- The String class supports several constructors.
- To create an empty String, you call the default constructor. For example,

  String s = new String();
- will create an instance of String with no characters in it.
- Frequently, you will want to create strings that have initial values.
- The **String class** provides a variety of constructors to handle this.
- To create a **String initialized by an array** of characters, use the constructor shown here:

  String(char *chars[ ])*
- Here is an example:

  char chars[] = { 'a', 'b', 'c' };

  String s = new String(chars);
- This constructor initializes **s with the string "abc".**

- You can specify a subrange of a character array as an initializer using the following constructor:

  String(char chars[ ], int startIndex, int numChars)

- Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.
- Here is an example:

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

- This initializes s with the characters cde.
- You can construct a String object that contains the same character sequence as another String object using this constructor:

  String(String strObj)

- Here, strObj is a String object.

```
// Construct one String from another.
class MakeString {
  public static void main(String args[]) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String();
    String s2 = new String(c);
    String s3 = new String(c,1,3);
    String s4 = new String(s2);
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
    System.out.println(s4);
  }
}
```

Output:
The output from this program is as follows:

Java
ava
Java

- 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 subrange.
- In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

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

Output:

ABCDEF
CDE

- You can construct a String from a StringBuffer by using the constructor shown here:

  String(StringBuffer strBufObj)
- J2SE 5 added two constructors to String.
- The first supports the extended Unicode character set and is shown here:

  String(int codePoints[ ], int startIndex, int numChars)
- Here, codePoints is an array that contains Unicode code points.
- The resulting string is constructed from the range that begins at startIndex and runs for numChars.
- The second new constructor supports the new StringBuilder class.
- It is shown here:

  String(StringBuilder strBuildObj)
- This constructs a String from the StringBuilder passed in strBuildObj.

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( )
```

- The following fragment prints “3”, since there are three characters in the string s:

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

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.
- These 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.

- Special String Operations :
  - String Literals
  - String Concatenation
  - String Concatenation with Other Data Types
  - String Conversion and toString( )

String Literals

- For each string literal in your program, Java automatically constructs a String object.
- Thus, you can use a string literal to initialize a String object.

```
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 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.
- For example, the following fragment concatenates three strings:

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

- 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 concatenate them. Here is an example:

```
// 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 Concatenation with Other Data Types

- You can concatenate strings with other types of data.
- For example,

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

- The int value in age is automatically converted into its string representation within a String object.
- This string is then concatenated as before.
- 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, however.
- You might get surprising results.
- Consider the following:

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

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.
- For objects, valueOf( ) calls the toString( ) method on the object.
- The toString( ) method has this general form:

  String toString( )
- To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

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

Output :

- Dimensions are 10.0 by 14.0 by 12.0
- Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

- The String class provides a number of ways in which characters can be extracted from a String object.
  - charAt( )
  - getChars( )
  - getBytes( )
  - toCharArray( )

charAt( )

- To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method.
- It has this general form:

```
char charAt(int where)
```

- Here, where is the index of the character that you want to obtain.
- The value of where must be nonnegative and specify a location within the string.
- charAt( ) returns the character at the specified location.
- For example,

```
char ch;
ch = "abc".charAt(1);
```

- assigns the value “b” to ch

getChars( )

- If you need to extract more than one character at a time, you can use the getChars( ) method.
- It has this general form:

  void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
- Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.
- Thus, the substring containsthe characters from sourceStart through sourceEnd -1.
- The array that will receive the characters is specified by target.
- The index within target at which the substring will be copied is passed in targetStart.
- Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

```
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);
}
}
```

Here is the output of this program:

```
demo
```

- **getBytes( )**
- There is an alternative to getChars( ) that stores the characters in an array of bytes.
- This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:

  byte[ ] getBytes( )
- Other forms of getBytes( ) are also available.
- getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.
- For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

- If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ).
- It returns an array of characters for the entire string.
- It has this general form:

  char[ ] toCharArray( )
- This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

```
class ChEx
{
public static void main(String args[])
{
String s = "This is a demo of the getChars method.";
char[] b = s.toCharArray();
System.out.println(b);
}
}
```

Output:

This is a demo of the getChars method.

String Comparison

- The String class includes several methods that compare strings or substrings within strings.
  - **equals( ) and equalsIgnoreCase( )**
  - **regionMatches( )**
  - **startsWith( ) and endsWith( )**
  - **equals( ) Versus ==**
  - **compareTo( )**

- **equals( ) and equalsIgnoreCase( )**
- To compare two strings for equality, use equals( ). It has this general form:

  boolean equals(Object str)
- Here, str is the String object being compared with the invoking String object.
- It returns true if the strings contain the same characters in the same order, and false otherwise.
- The comparison is case-sensitive.
- To perform a comparison that ignores case differences, call equalsIgnoreCase( ).
- When it compares two strings, it considers A-Z to be the same as a-z.
- It has this general form:

  boolean equalsIgnoreCase(String str)
- Here, str is the String object being compared with the invoking String object.
- It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

```
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));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

- **regionMatches( )**
- The regionMatches( ) method compares a specific region inside a string with another specific region in another string.
- There is an overloaded form that allows you to ignore case in such comparisons.
- Here are the general forms for these two methods:

  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.

```
class ChEx {
public static void main(String args[]) {
String s1 = "Hello welcome";
String s2 = "Good-bye";
String s3 = "HELLO";
System.out.println(s1 + " regionmatch " + s2 + " -> " +
    s1.regionMatches(0,s2,0,5));
System.out.println(s1 + " regionmatch ignore case" + s3 +
    " -> " + s1.regionMatches(true,0,s3,0,5));
}
}
```

Output:

Hello welcome regionmatch Good-bye -> false

Hello welcome regionmatch ignore case HELLO -> true

- **startsWith( ) and endsWith( )**
- String defines two routines that are, more or less, specialized forms of regionMatches( ).
- The startsWith( ) method determines whether a given String begins with a specified string.
- Conversely, endsWith( ) determines whether the String in question ends with a specified string. They have the following general forms:

  boolean startsWith(String str)

  boolean endsWith(String str)
- Here, str is the String being tested. If the string matches, true is returned. Otherwise, false is returned.
- For example,

  "Foobar".endsWith("bar") and "Foobar".startsWith("Foo")
- are both true.
- A second form of startsWith( ), shown here, lets you specify a starting point:

  boolean startsWith(String str, int startIndex)
- Here, startIndex specifies the index into the invoking string at which point the search will begin.
- For example,
- "Foobar".startsWith("bar", 3) returns true.

- **equals( ) Versus ==**
- It is important to understand that the equals( ) method and the == operator perform two different operations.
- 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.
- The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

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

Output:

Hello equals Hello -> true

Hello == Hello -> false

- **compareTo( )**
- 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.
- The String method compareTo( ) serves this purpose.
- It has this general form:

  int compareTo(String str)
- Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted, as shown here:

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.

```
// A bubble sort for Strings.
class SortString {
static String arr[] = {
"Now", "is", "the", "time", "for", "all", "good", "men", "to", "come", "too",
    "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]);
}
}
}
```

- The output of this program is the list of words:

Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
too

- If you want to ignore case differences when comparing two strings, use compareToIgnoreCase( ), as shown here:

  int compareToIgnoreCase(String *str)*

- This method returns the same results as compareTo( ), except that case differences are ignored.

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.

- To search for the first occurrence of a character, use

  int indexOf(int ch)
- To search for the last occurrence of a character, use

  int lastIndexOf(int ch)
- Here, ch is the character being sought.
- To search for the first or last occurrence of a substring, use

  int indexOf(String str)

  int lastIndexOf(String str)
- Here, str specifies the substring.

- You can specify a starting point for the search using these forms:

  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.

```
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));
}
}
```

- **Output :**

Now is the time for all good men to come to the aid of their country.

indexOf(t) = 7

lastIndexOf(t) = 65

indexOf(the) = 7

lastIndexOf(the) = 55

indexOf(t, 10) = 11

lastIndexOf(t, 60) = 55

indexOf(the, 10) = 44

lastIndexOf(the, 60) = 55

Modifying a String

- 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( )
  - concat()
  - replace()
  - trim()

substring( )

- You can extract a substring using substring( ). It has two forms. The first is

  String substring(int startIndex)
- Here, startIndex specifies the index at which the substring will begin.
- This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.
- The second form of substring( ) allows you to specify both the beginning and ending index of the substring:

  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.

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

Output:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was,
    too.
```

concat( )

- You can concatenate two strings using concat( ), shown here:

  String concat(String str)
- This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +.
- For example,

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

- puts the string “onetwo” into s2.
- It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

- The replace( ) method has two forms.
- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

  String replace(char original, char replacement)
- Here, original specifies the character to be replaced by the character specified by replacement.
- The resulting string is returned. For example,

  String s = "Hello".replace('l', 'w');
- puts the string “Hewwo” into s.
- 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.

trim( )

- The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.
- It has this general form:

  String trim( )
- Here is an example:

  String s = " Hello World ".trim();
- This puts the string “Hello World” into s.
- The trim( ) method is quite useful when you process user commands.

```
class StringCon {
public static void main(String args[]) {
String s1 = "  Hello World  ";
String s2 = "two";
String s3 = s1.trim();
System.out.println("Resultant string is="+s3);
}
}
```

Output:

Resultant string is=Hello World

Data Conversion Using valueOf( )

- The valueOf( ) method converts data from its internal format into a human-readable form.
- It is a static method that is overloaded within String for all of Java's built-in types so that each type can be converted properly into a string. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument.
- Here are a few of its forms:
  - static String valueOf(double *num)*
  - static String valueOf(long *num)*
  - static String valueOf(Object *ob)*
  - static String valueOf(char *chars[ ])*

- valueOf( ) is called when a string representation of some other type of data is needed—for example, during concatenation operations.
- You can call this method directly with any data type and get a reasonable String representation.
- All of the simple types are converted to their common String representation.
- Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method.

Changing the Case of Characters Within a String

- The method toLowerCase( ) converts all the characters in a string from uppercase to lowercase.
- The toUpperCase( ) method converts all the characters in a string from lowercase to uppercase.
- Nonalphabetical characters, such as digits, are unaffected.
- Here are the general forms of these methods:

  String toLowerCase( )

  String toUpperCase( )
- Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String.

```
class ChangeCase {
public static void main(String args[])
{
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

Additional String Methods

Method	Description
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **String**.

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false**, otherwise. Added by J2SE 5.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**. Added by J2SE 5.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.

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 preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- StringBuffer defines these four constructors:
  StringBuffer( )
  StringBuffer(int *size)*
  StringBuffer(String *str)*
  StringBuffer(CharSequence *chars)*
- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer.
- The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.
- StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.
- The fourth constructor creates an object that contains the character sequence contained in *chars.*

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.
- They have the following general forms:

  int length( )

  int capacity( )
- Here is an example:

```
// 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());
}
}
```

Output

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

ensureCapacity( )

- If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
- This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
- It has this general form:

  void ensureCapacity(int capacity)
- Here, capacity specifies the size of the buffer.

setLength( )

- To set the length of the buffer within a StringBuffer object, use setLength( ).
- Its general form is shown here:

  void setLength(int len)

- Here, len specifies the length of the buffer. This value must be nonnegative.
- When you increase the size of the buffer, null characters are added to the end of the existing buffer.

charAt( ) and setCharAt( )

- The value of a single character can be obtained from a StringBuffer via the charAt( ) method.
- You can set the value of a character within a StringBuffer using setCharAt( ).
- Their general forms are shown here:

  char charAt(int where)

  void setCharAt(int where, char ch)
- For charAt( ), where specifies the index of the character being obtained.
- For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.
- For both methods, where must be nonnegative and must not specify a location beyond the end of the buffer.

```
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1));
}
}
```

Output:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

- To copy a substring of a StringBuffer into an array, use the getChars( ) method.
- It has this general form:
- void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
- Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.
- This means that the substring contains the characters from sourceStart through sourceEnd – 1.
- The array that will receive the characters is specified by target.
- The index within target at which the substring will be copied is passed in targetStart.
- Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

append( )

- The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.
- It has several overloaded versions. Here are a few of its forms:

  StringBuffer append(String str)

  StringBuffer append(int num)

  StringBuffer append(Object obj)
- String.valueOf( ) is called for each parameter to obtain its string representation.
- The result is appended to the current StringBuffer object.
- The buffer itself is returned by each version of append( ).

```
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer();
s = sb.append("a = "). append(a). append("!").toString();
System.out.println(s);
}
}
```

The output of this example is shown here:

```
a = 42!
```

insert( )

- The insert( ) method inserts one string into another.
- It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with.
- This string is then inserted into the invoking StringBuffer object.
- These are a few of its forms:

  StringBuffer insert(int index, String str)

  StringBuffer insert(int index, char ch)

  StringBuffer insert(int index, Object obj)
- Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

```
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);
}
}
```

The output of this example is shown here:

I like Java!

reverse( )

- You can reverse the characters within a StringBuffer object using reverse( ), shown here:

  StringBuffer reverse( )

- This method returns the reversed object on which it was called.

```
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

- Characters can be deleted within a StringBuffer by using the methods delete( ) and deleteCharAt( ).
- These methods are shown here:

  StringBuffer delete(int startIndex, int endIndex)

  StringBuffer deleteCharAt(int loc)
- The delete( ) method deletes a sequence of characters from the invoking object.
- Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.
- Thus, the substring deleted runs from startIndex to endIndex –1.
- The resulting StringBuffer object is returned.
- The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

```
class deleteDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.delete(4, 7);
System.out.println("After delete: " + sb);
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);
}
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( )

- You can replace one set of characters with another set inside a StringBuffer object by calling replace( ).
- Its signature is shown here:

  StringBuffer replace(int startIndex, int endIndex, String str)
- The substring being replaced is specified by the indexes startIndex and endIndex.
- Thus, the substring at startIndex through endIndex – 1 is replaced.
- The replacement string is passed in str.
- The resulting StringBuffer object is returned.

```
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
}
}
```

Here is the output:

After replace: This was a test.

substring( )

- You can obtain a portion of a StringBuffer by calling substring( ).
- It has the following two forms:

  String substring(int startIndex)

  String substring(int startIndex, int endIndex)
- The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.
- The second form returns the substring that starts at startIndex and runs through endIndex–1.

```
class StringCon {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
String s1 = sb.substring(5);
System.out.println("substring 1: " + s1);
String s2 = sb.substring(5);
System.out.println("substring 2: " + s2);
}
}
```

Output :

substring 1: is a test.

substring 2: is

Additional StringBuffer Methods

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.

Method	Description
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **StringBuffer**.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

```
class IndexOfDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("one two one");
int i;
i = sb.indexOf("one");
System.out.println("First index: " + i);
i = sb.lastIndexOf("one");
System.out.println("Last index: " + i);
}
}
```

The output is shown here:

```
First index: 0
Last index: 8
```

StringBuilder

- J2SE 5 adds a new string class to Java's already powerful string handling capabilities.
- This new class is 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.

Enumerations and Type Wrappers

Enumeration Fundamentals

- An enumeration is a list of named constants.
- In Java, an enumeration defines a class type.
- An enumeration is created using the enum keyword.
- For example, here is a simple enumeration that lists various apple varieties:
- // An enumeration of apple varieties.

```
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

- The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- Each is implicitly declared as a public, static final member of Apple.
- Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case.
- Thus, in the language of Java, these constants are called self-typed, in which “self” refers to the enclosing enumeration.
- Once you have defined an enumeration, you can create a variable of that type.
- For example, this declares ap as a variable of enumeration type Apple:

```
Apple ap;
```

- Because ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration.
- For example, this assigns ap the value RedDel:

```
ap = Apple.RedDel;
```

- Notice that the symbol RedDel is preceded by Apple.

- Two enumeration constants can be compared for equality by using the = = relational operator.
- For example, this statement compares the value in ap with the GoldenDel constant:

```
if(ap == Apple.GoldenDel) // ...
```

- An enumeration value can also be used to control a switch statement.
- Of course, all of the case statements must use constants from the same enum as that used by the switch expression.
- For example, this switch is perfectly valid:

```
switch(ap) {
case Jonathan:
// ...
case Winesap:
// ...
```

- When an enumeration constant is displayed, such as in a println( ) statement, its name is output.
- For example, given this statement:

```
System.out.println(Apple.Winesap);
```

- the name Winesap is displayed.

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo {
public static void main(String args[])
{
Apple ap;
ap = Apple.RedDel;
// Output an enum value.
System.out.println("Value of ap: " + ap);
System.out.println();
ap = Apple.GoldenDel;
// Compare two enum values.
if(ap == Apple.GoldenDel)
System.out.println("ap contains GoldenDel.\n");
// Use an enum to control a switch statement.
switch(ap) {
case Jonathan:
System.out.println("Jonathan is red.");
break;
case GoldenDel:
System.out.println("Golden Delicious is yellow.");
break;
case RedDel:
System.out.println("Red Delicious is red.");
break;
case Winesap:
System.out.println("Winesap is red.");
break;
case Cortland:
System.out.println("Cortland is red.");
break;
}
}
}
```

The output from the program is shown here:

```
Value of ap: RedDel
ap contains GoldenDel.
Golden Delicious is yellow.
```

The values( ) and valueOf( ) Methods

- All enumerations automatically contain two predefined methods: values( ) and valueOf( ).
- Their general forms are shown here:

  public static enum-type[ ] values( )

  public static enum-type valueOf(String str)
- The values( ) method returns an array that contains a list of the enumeration constants.
- The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.
- In both cases, enum-type is the type of the enumeration.

```
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2 {
public static void main(String args[])
{
Apple ap;
System.out.println("Here are all Apple constants:");
// use values()
Apple allapples[] = Apple.values();
for(Apple a : allapples)
System.out.println(a);
System.out.println();
// use valueOf()
ap = Apple.valueOf("Winesap");
System.out.println("ap contains " + ap);
}
}
```

Output :
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland
ap contains Winesap

Java Enumerations Are Class Types

- A Java enumeration is a class type.
- Although you don't instantiate an enum using new, it otherwise has much the same capabilities as other classes.
- The fact that enum defines a class gives powers to the Java enumeration that enumerations in other languages simply do not have.
- For example, you can give them constructors, add instance variables and methods, and even implement interfaces.
- Each enumeration constant is an object of its enumeration type.
- Thus, when you define a constructor for an enum, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
enum Apple {
Jonathan(10), GoldenDel(9),
    RedDel(12), Winesap(15),
    Cortland(8);
private int price;
Apple(int p) { price = p; }
int getPrice() { return price; }
}
class EnumDemo3 {
public static void main(String
    args[])
{
Apple ap;
System.out.println("Winesap
    costs " +
Apple.Winesap.getPrice() +
" cents.\n");
System.out.println("All apple
    prices:");
for(Apple a : Apple.values())
System.out.println(a + " costs " +
    a.getPrice() +
" cents.");
}
}
```

Output :

- Winesap costs 15 cents.
- All apple prices:
- Jonathan costs 10 cents.
- GoldenDel costs 9 cents.
- RedDel costs 12 cents.
- Winesap costs 15 cents.
- Cortland costs 8 cents.

Enumerations Inherit Enum

- Although you can't inherit a superclass when declaring an enum, all enumerations automatically inherit one: java.lang.Enum.
- This class defines several methods that are available for use by all enumerations.
- You can obtain a value that indicates an enumeration constant's position in the list of constants.
- This is called its ordinal value, and it is retrieved by calling the ordinal( ) method, shown here:

  final int ordinal( )
- It returns the ordinal value of the invoking constant.
- Ordinal values begin at zero.

- You can compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method.
- It has this general form:

  final int compareTo(enum-type e)
- Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant.
- You can compare for equality an enumeration constant with any other object by using equals( ), which overrides the equals( ) method defined by Object.

```
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap,
    Cortland
}
class EnumDemo4 {
public static void main(String args[])
{
Apple ap, ap2, ap3;
// Obtain all ordinal values using
    ordinal().
System.out.println("Here are all apple
    constants" +
" and their ordinal values: ");
for(Apple a : Apple.values())
System.out.println(a + " " + a.ordinal());
ap = Apple.RedDel;
ap2 = Apple.GoldenDel;
ap3 = Apple.RedDel;
System.out.println();
// Demonstrate compareTo() and
    equals()
if(ap.compareTo(ap2) < 0)
System.out.println(ap + " comes before "
    + ap2);
if(ap.compareTo(ap2) > 0)
System.out.println(ap2 + " comes before
    " + ap);
if(ap.compareTo(ap3) == 0)
System.out.println(ap + " equals " + ap3);
System.out.println();
if(ap.equals(ap2))
System.out.println("Error!");
if(ap.equals(ap3))
System.out.println(ap + " equals " + ap3);
if(ap == ap3)
System.out.println(ap + " == " + ap3);
}
}
```

- Output :

Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4
GoldenDel comes before RedDel
RedDel equals RedDel
RedDel equals RedDel
RedDel == RedDel

```
enum Answers {
NO, YES, MAYBE, LATER, SOON, NEVER
}
class Question {
Random rand = new Random();
Answers ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 15)
return Answers.MAYBE; // 15%
else if (prob < 30)
return Answers.NO; // 15%
else if (prob < 60)
return Answers.YES; // 30%
else if (prob < 75)
return Answers.LATER; // 15%
else if (prob < 98)
return Answers.SOON; // 13%
else
return Answers.NEVER; // 2%
}
}
class AskMe {
static void answer(Answers result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Type Wrappers

- Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language.
- Primitive types, rather than objects, are used for these quantities for the sake of performance.
- Using objects for these values would add an unacceptable overhead to even the simplest of calculations.
- you can’t pass a primitive type by reference to a method.

- Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.
- To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.
- The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character, and Boolean.**
- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character

- Character is a wrapper around a char.
- The constructor for Character is

  Character(char ch)
- Here, ch specifies the character that will be wrapped by the Character object being created.
- To obtain the char value contained in a Character object, call charValue( ), shown here:

  char charValue( )
- It returns the encapsulated character.

Boolean

- Boolean is a wrapper around boolean values.
- It defines these constructors:

  Boolean(boolean boolValue)

  Boolean(String boolString)
- In the first version, boolValue must be either true or false.
- In the second version, if boolString contains the string “true” (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.
- To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

  boolean booleanValue( )
- It returns the boolean equivalent of the invoking object.

The Numeric Type Wrappers

- The most commonly used type wrappers are those that represent numeric values.
- These are Byte, Short, Integer, Long, Float, and Double.
- All of the numeric type wrappers inherit the abstract class Number.
- Number declares methods that return the value of an object in each of the different number formats.
- These methods are shown here:

```
byte  byteValue( )
double  doubleValue( )
float   floatValue( )
int   intValue( )
long   longValue( )
short   shortValue( )
```

- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.
- For example, here are the constructors defined for Integer:

  Integer(int num)

  Integer(String str)
- If str does not contain a valid numeric value, then a NumberFormatException is thrown.
- All of the type wrappers override toString( ).
- It returns the human-readable form of the value contained within the wrapper.
- This allows you to output the value by passing a type wrapper object to println( ), for example, without having to convert it into its primitive type.

```
// Demonstrate a type wrapper.
class Wrap {
public static void main(String args[]) {
Integer iOb = new Integer(100);
int i = iOb.intValue();
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

- This program wraps the integer value 100 inside an Integer object called iOb.
- The program then obtains this value by calling intValue( ) and stores the result in i.

- The process of encapsulating a value within an object is called boxing.
- Thus, in the program, this line boxes the value 100 into an Integer:

```
Integer iOb = new Integer(100);
```

- The process of extracting a value from a type wrapper is called unboxing.
- For example, the program unboxes the value in iOb with this statement:

```
int i = iOb.intValue();
```

I/O, Applets, and Other Topics

I/O Basics

- Java does provide strong, flexible support for I/O as it relates to files and networks.
- Java's I/O system is cohesive and consistent.

Streams

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

Byte Streams and Character Streams

- Java defines two types of streams: byte and character.
  - Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.
  - Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes: InputStream and OutputStream.
- Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers.
- The byte stream classes are shown in Table 13-1.
- The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.
- Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data.
- Both methods are declared as abstract inside InputStream and OutputStream.
- They are overridden by derived stream classes.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

TABLE 13-1 The Byte Stream Classes

The Character Stream Classes

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes, Reader and Writer.
- These abstract classes handle Unicode character streams.
- Java has several concrete subclasses of each of these.
- The character stream classes are shown in Table 13-2.
- The abstract classes Reader and Writer define several key methods that the other stream classes implement.
- Two of the most important methods are read( ) and write(), which read and write characters of data, respectively.
- These methods are overridden by derived stream classes.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer

TABLE 13-2 The Character Stream I/O Classes

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

TABLE 13-2 The Character Stream I/O Classes *(continued)*

The Predefined Streams

- All Java programs automatically import the java.lang package.
- This package defines a class called System, which encapsulates several aspects of the run-time environment.
- For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system.
- System also contains three predefined stream variables: in, out, and err.
- These fields are declared as public, static, and final within System.
- This means that they can be used by any other part of your program and without reference to a specific System object.

- System.out refers to the standard output stream.
- By default, this is the console.
- System.in refers to standard input, which is the keyboard by default.
- System.err refers to the standard error stream, which also is the console by default.
- However, these streams may be redirected to any compatible I/O device.
- System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.
- These are byte streams, even though they typically are used to read and write characters from and to the console.

Reading Console Input

- In Java, console input is accomplished by reading from System.in.
- To obtain a characterbased 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)
- 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
- Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value.
- It returns –1 when the end of the stream is encountered.
- It can throw an IOException.
- The following program demonstrates read( ) by reading characters from the console until the user types a "q.”

```
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');
}
}
```

Output :

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

Reading Strings

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

  String readLine( ) throws IOException
- It returns a String object.
- The following program demonstrates BufferedReader and the readLine( ) method;
- The program reads and displays lines of text until you enter the word “stop”:

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

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

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

Writing Console Output

- Console output is most easily accomplished with print( ) and println( ).
- These methods are defined by the class PrintStream (which is the type of object referenced by System.out).
- Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ).
- Thus, write( ) can be used to write to the console.
- The simplest form of write( ) defined by PrintStream is shown here:

  void write(int byteval)

- This method writes to the stream the byte specified by byteval.
- Although byteval is declared as an integer, only the low-order eight bits are written.

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

The PrintWriter Class

- For real-world programs, the recommended method of writing to the console when using Java is through a PrintWriter stream.
- PrintWriter is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your program.
- PrintWriter defines several constructors. The one we will use is shown here:

  PrintWriter(OutputStream outputStream, boolean flushOnNewline)
- Here, outputStream is an object of type OutputStream, and flushOnNewline controls whether Java flushes the output stream every time a println( ) method is called. If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d);
}
}
```

Output :
This is a string
-7
4.5E-7

Reading and Writing Files

- Java provides a number of classes and methods that allow you to read and write files.
- In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.
- Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files.
- To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.
- While both classes support additional, overridden constructors, the following are the forms that we will be using:

  FileInputStream(String fileName) throws FileNotFoundException

  FileOutputStream(String fileName) throws FileNotFoundException

- Here, fileName specifies the name of the file that you want to open.
- When you create an input stream, if the file does not exist, then FileNotFoundException is thrown.
- For output streams, if the file cannot be created, then FileNotFoundException is thrown.
- When an output file is opened, any preexisting file by the same name is destroyed.
- When you are done with a file, you should close it by calling close().
- It is defined by both FileInputStream and FileOutputStream, as shown here:

  void close( ) throws IOException
- To read from a file, you can use a version of read( ) that is defined within FileInputStream.
- The one that we will use is shown here:

  int read( ) throws IOException
- Each time that it is called, it reads a single byte from the file and returns the byte as an integer value.
- read( ) returns –1 when the end of the file is encountered. It can throw an IOException.

```
/* Display a text file.
To use this program, specify the
    name
of the file that you want to see.
For example, to see a file called
    TEST.TXT,
use the following command line.
java ShowFile TEST.TXT
*/
import java.io.*;
class ShowFile {
public static void main(String args[])
throws IOException
{
int i;
FileInputStream fin;
try {
fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("File Not Found");
return;
}
    catch(ArrayIndexOutOfBoundsExc
    eption e) {
System.out.println("Usage: ShowFile
    File");
return;
}
// read characters until EOF is
    encountered
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
} while(i != -1);
fin.close();
}
}
```

- To write to a file, you can use the write( ) method defined by FileOutputStream.
- Its simplest form is shown here:

  void write(int byteval) throws IOException
- This method writes the byte specified by byteval to the file.
- Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an IOException is thrown. The next example uses write( ) to copy a text file:

```
/* Copy a text file.
To use this program, specify the name
of the source file and the destination file.
For example, to copy a file called FIRST.TXT
to a file called SECOND.TXT, use the following
command line.
java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
public static void main(String args[])
throws IOException
{
int i;
FileInputStream fin;
FileOutputStream fout;
try {
// open input file
try {
fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("Input File Not Found");
return;
}
// open output file
try {
fout = new FileOutputStream(args[1]);
}
catch(FileNotFoundException e) {
System.out.println("Error Opening Output File");
return;
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Usage: CopyFile From To");
return;
}
// Copy File
try {
do {
i = fin.read();
if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
System.out.println("File Error");
}
fin.close();
fout.close();
}
}
```

Applets

Two Types of Applets

- The first are those based directly on the Applet class. These applets use the Abstract Window Toolkit (AWT) to provide the Graphical User Interface(GUI).
- The second type of applets are those based on the Swing class JApplet. Swing applets use the Swing classes to provide the GUI.

- All applets are subclasses (either directly or indirectly) of Applet.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.
- Execution of an applet does not begin at main( ).
- Instead, execution of an applet is started and controlled with an entirely different mechanism.
- To use an applet, it is specified in an HTML file. One way to do this is by using the APPLET tag.
- The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file.

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

An Applet Skeleton

- All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods, init( ), start( ), stop( ), and destroy(), apply to all applets and are defined by Applet.
- AWT-based applets will also override the paint( ) method, which is defined by the AWT Component class.
- This method is called when the applet's output must be redisplayed.
- These five methods can be assembled into the skeleton shown here:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```


Applet Viewer: AppletSkel
Applet
Applet started.

Simple Applet Display Methods

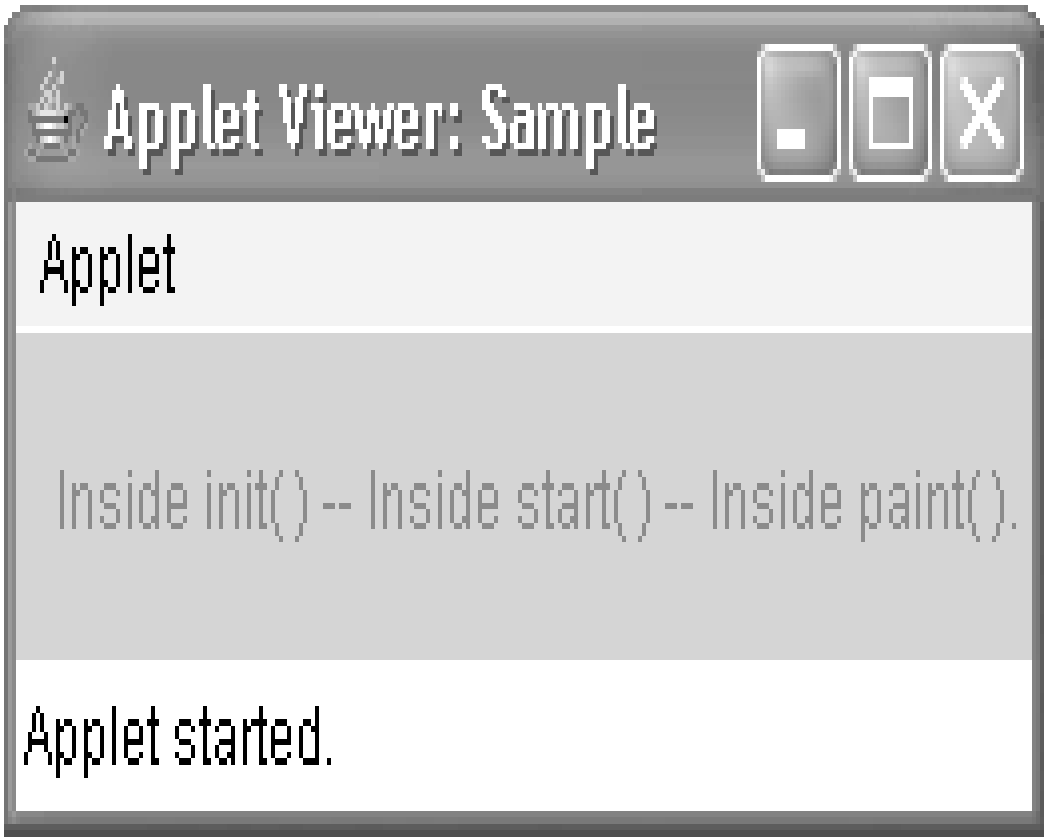
- Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init( ), start( ), and paint( )** methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
String msg;

// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}

// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}

// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```


Applet Viewer: Sample
Applet
Inside init( ) -- Inside start( ) -- Inside paint( ).
Applet started.

The transient and volatile Modifiers

- Java defines two interesting type modifiers: transient and volatile.
- These modifiers are used to handle somewhat specialized situations.
- When an instance variable is declared as transient, then its value need not persist when an object is stored. For example:

```
class T {
transient int a; // will not persist
int b; // will persist
}
```

- Here, if an object of type T is written to a persistent storage area, the contents of a would not be saved, but the contents of b would.

- The volatile modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program.
- One of these situations involves multithreaded programs.
- In a multithreaded program, sometimes two or more threads share the same variable.
- For efficiency considerations, each thread can keep its own, private copy of such a shared variable.
- The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered.
- While this approach works fine, it may be inefficient at times.
- In some cases, all that really matters is that the master copy of a variable always reflects its current state.
- To ensure this, simply specify the variable as volatile, which tells the compiler that it must always use the master copy of a volatile variable.

Using instanceof

- Sometimes, knowing the type of an object during run time is useful.
- For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects.
- In this situation, it might be useful for the processing thread to know the type of each object when it receives it.
- Another situation in which knowledge of an object's type at run time is important involves casting.
- In Java, an invalid cast causes a run-time error.
- Many invalid casts can be caught at compile time.
- However, casts involving class hierarchies can produce invalid casts that can be detected only at run time.

- The instanceof operator has this general form:

  objref instanceof type
- Here, objref is a reference to an instance of a class, and type is a class type.
- If objref is of the specified type or can be cast into the specified type, then the instanceof operator evaluates to true.
- Otherwise, its result is false.
- Thus, instanceof is the means by which your program can obtain run-time type information about an object.

```
class A {
int i, j;
}
class B {
int i, j;
}
class C extends A {
int k;
}
class D extends A {
int k;
}
class InstanceOf {
public static void main(String args[]) {
A a = new A();
B b = new B();
C c = new C();
D d = new D();
if(a instanceof A)
System.out.println("a is instance of A");
if(b instanceof B)
System.out.println("b is instance of B");
if(c instanceof C)
System.out.println("c is instance of C");
if(c instanceof A)
System.out.println("c can be cast to A");
if(a instanceof C)
System.out.println("a can be cast to C");
System.out.println();
// compare types of derived types
A ob;
ob = d; // A reference to d
System.out.println("ob now refers to d");
if(ob instanceof D)
System.out.println("ob is instance of D");
System.out.println();
ob = c; // A reference to c
System.out.println("ob now refers to c");
if(ob instanceof D)
System.out.println("ob can be cast to D");
else
System.out.println("ob cannot be cast to D");
if(ob instanceof A)
System.out.println("ob can be cast to A");
System.out.println();
// all objects can be cast to Object
if(a instanceof Object)
System.out.println("a may be cast to Object");
if(b instanceof Object)
System.out.println("b may be cast to Object");
if(c instanceof Object)
System.out.println("c may be cast to Object");
if(d instanceof Object)
System.out.println("d may be cast to Object");
}
}
```

strictfp

- A relatively new keyword is strictfp.
- With the creation of Java 2, the floating-point computation model was relaxed slightly.
- Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation.
- This prevents overflow or underflow in some cases.
- By modifying a class or a method with strictfp, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java.
- When a class is modified by strictfp, all the methods in the class are also modified by strictfp automatically.
- For example, the following fragment tells Java to use the original floating-point model for calculations in all methods defined within MyClass:

  strictfp class MyClass { //...