Java: The Complete Reference

Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This "cross-pollination" from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language

__String___

- ✔ string is a sequence of characters. Java implements strings as objects of type String that make string handling convenient.
- ✔ When you create a String object, you are creating a string that cannot be changed.
- ✔ That is, once a String object has been created, you cannot change the characters that comprise that string.
- ✔ 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.
- ✔ This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.
- ✔ For those cases in which a modifiable string is desired, Java provides two options: StringBuffer and StringBuilder.
- ✔ Both hold strings that can be modified after they are created. TheString ,StringBuffer, andStringBuilder classes are defined in java.lang. All are declared final, which means that none of these classes may be sub-classed.
- ✔ 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 s = new String();**
- ⇨ This will create an instance of String with no characters in it.

✔ **char chars[] = { 'a', 'b', 'c' };**
String s = new String(chars);
- ⇨ This constructor initializes s with the string **"abc".**

✔ **char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };**
String s = new String(chars, 2, 3);
- ⇨ This initializes s with the characters **cde.**

✔
```
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);
}
}
```
The output from this program is as follows:
```
Java
Java
```
- ⇨ s1and s2 contain the same string.

✔ String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

✔ The first thing to understand about strings is that every string you create is actually an object of type String. Even string constants are actually String objects.

✔ For example, in the statement

System.out.println("This is a String, too");
the string "This is a String, too" is a String constant. The second thing to understand about strings is that objects of type String are immutable;
once a String object is created, its contents cannot be altered

✔ While this may seem like a serious restriction, it is still not.
✔ If you need to change a string, you can always create a new one that contains the modifications.

- ✔ Java defines a peer class of String, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java

- ✔ Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

  ```
  String myString = "this is a test";
  ```

  Once you have created a String object, you can use it anywhere that a string is allowed. For example, this statement displays myString

  ```
  System.out.println(myString);
  ```

- ✔ You can test two strings for equality by using **equals( ).** You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( ).**

- ✔ Byte[] specifies the array of bytes.

- ✔ **class AsciiString {**

  ```
  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);
  }
  }
  ```

  This program generates the following output:

  ```
  ABCDEFCDE
  CDE
  ```

- ✔ The byte-to-character conversion is done by using the default character encoding of the platform.

- ✔ 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. For example, the following code fragment creates two equivalent strings:

  ```
  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. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the length( ) method on the string “abc”.

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

As expected, it prints “3”.

✔ The + operator, 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);
```

This displays the string “He is 9 years old.

✔ You can concatenate strings with other types of data.

✔ In this case, age is an int rather than another String, but the output produced is the same as before. This is because 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.

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

This fragment displays four: 22 rather than the four: 4

✔ Operator precedence causes the concatenation of “four” with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time.

✔ To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2)
```

Now s contains the string “four: 4”

- ✔ 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.

- ✔ valueOf( )returns a string for simple types and For objects, valueOf( ) calls the toString( ) method on the object.

- ✔ the toString( ) method is the means by which you can determine the string representation for objects of classes that you create. Every class implements toString( ) because it is defined by Object.

String toString( )

- ✔ To implement toString( ), simply return a String object that appropriately describes an object of your class.

- ✔ By overriding toString( ) for classes that you create, you allow them to be used in print( ) and println( ) statements and in concatenation expressions.

```
// Overriding 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);
```

```
}
}
```

✔ The output of this program is shown here:

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

✔ As you can see, Box's toString( ) method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

✔ The characters that comprise a string within a String object cannot be indexed as if they were a character array.

✔ Following are some ways in which characters can be extracted from a String object.

✔ 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.

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

assigns the value "b" to ch.

✔ 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 to obtain the substring that lies between sourceStart and sourceEnd.

✔ 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.

✔ To impletment **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);
} }
```

Here is the output of this program:

Demo

✔ 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( )

✔ 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.

✔ 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()

✔ STRING COMPARISON: 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. The comparison is case-sensitive.

✔ To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. It has this general form:

boolean equalsIgnoreCase(String str)

✔ **// 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)); } }

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

✔ The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.

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.

✔ 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")
"Foobar".startsWith("Foo")

are both true.

➪ **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.

✔ It is important to understand that the equals( ) method and the == operator perform two different operations.
✔ As just explained, the equals( ) method compares the characters inside a String object.
✔ But, The == operator compares two object references to see whether they refer to the same instance.

```
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));
} }
```

The variable s1 refers to the String instance created by “Hello”.
The object referred to by s2 is created with s1 as an initializer.
Thus, the contents of the two String objects are identical, but they are distinct objects.
This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

Hello equals Hello -> true
Hello == Hello -> false

✔ 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.

```
static String arr[] = { "Now", "is", "the", "time", "for", "all", "good", "men", "to",
"come", "to", "aid", "of", "their", "country" };
```

✔ On comparing and sorting each string literal in the list, The output of this program becomes the order of words as shown below:

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

✔ **Now** is first, because it has an uppercase, and thus a lower ascii value. To ignore cases, use compareToIgnoreCase( ), as shown here:

int compareToIgnoreCase(String str)

SEARCHING STRINGS :-

✔ indexOf( ) searches for the first occurrence of a character or substring.

✔ lastIndexOf( ) searches for the last occurrence of a character or substring.

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

int indexOf(char ch)

✔ To search for the last occurrence of a character, use

int lastIndexOf(char ch)

✔ To search for the first or last occurrence of a substring, use

int indexOf(String str)

or

int lastIndexOf(String str)

Also,

int indexOf(String str, int startIndex)

Or,

int indexOf(char ch, int startIndex)

✔ For indexOf( ), the search runs from startIndex to the end of the string.

✔ For lastIndexOf( ), the search runs from startIndex to zero.

✔ For example,

String s = "The quick brown fox jumped out of the window";

s.lastIndexOf("the", 15)

//goes from right to left from startindex to zero so first occurence is actually the last one from rt. To left. **output: 0**

s.IndexOf("the", 15)

// goes from left to right from startindex to end so first occurrence is the required one. **output: 33**

s.IndexOf("the")

// goes from left to right from zero to end for first occurrence. **Output: 0**

s.lastIndexOf("the")

// goes from left to right from zero to end for last occurrence. **output:33**

STRING MODIFICATION

✔ 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 String methods, which constructs a new copy of the string with your modifications

✔ You can extract a substring from a string object using substring( ).

String substring(int startIndex)

String substring(int startIndex, int endIndex)

✔ The string returned contains all the characters from the beginning index, up to, but not including, the ending index

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

```
String concat(String str)
```

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

✔ **Replace()** replaces all occurrences of one character in the invoking string with another character.

```
String replace(char original, char replacement)
```

✔ Here, original specifies the character to be replaced by the character specified by replacement.

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into s

✔ **To replace** one character sequence with another –

```
String replace(CharSequence original, CharSequence replacement)
```

✔ Trim() is used to remove any leading or trailing whitespace that may have inadvertently been entered by the user. It is quite useful when you process user commands, to remove unnecessary spaces, but not in between the string literal.

```
String s = " Hello World ".trim();
```

This puts the string “Hello World” into s

```
// create a BufferedReader using System.in
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 String str;
```

```
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do {
str = br.readLine();
str = str.trim();            // remove whitespace
```

✔ **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.

✔ If in case Any **object** that you pass to valueOf( ), it will return the result of a call to the **object's toString( ) method.**

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(char[] c)    // creates a string containing all characters
public static String valueOf(int i)
public static String valueOf(long l)
public static String valueOf(float f)
public static String valueOf(double d)
public static String valueOf(Object o)
```

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

✔ Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring to be returned.

```
public static void main(String args[])
int value=30;
String s1=String.valueOf(value);
            //For others, e.g., Boolean.valueOf(value) or Integer.valueOf(value)
System.out.println(s1+10);              //concatenating string with 10
}}
        Output: 3010
```

✔ 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

```
String toLowerCase( )
String toUpperCase( )
String s = "This is a test.";
String upper = s.toUpperCase();
String lower = s.toLowerCase();
```

boolean contains(CharSequence str)

// Returns true if the invoking object contains the string specified by str; else false

boolean matches(string regExp)

// Returns true if the invoking string matches the regular expression pattern passed in regExp, else false.

String replaceFirst(String regExp, String newStr)
String replaceAll(String regExp, String newStr

// returns a string in which the first or all substrings that matches the regular expression specified by regExp is replaced by newStr.

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.

CharSequence subSequence(int startIndex, int stopIndex)

Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex. Like substring() for String, subsequence() is for CharSequence.

STRINGBUFFER

✔ String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.

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.
✔ The fourth constructor creates an object that contains the character sequence contained in chars.

✔ 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( )

str1.length()

- ✔ 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.

void ensureCapacity(int capacity)

str1.ensureCapacity(40);

Here, capacity specifies the size of the buffer

- ✔ To set the length of the buffer within a StringBuffer object, use setLength( ).

void setLength(int len)

- ✔ When you increase the size of the buffer, null characters are added to the end of the existing buffer. If you call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost.
- ✔ 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)

- ✔ To copy a substring of a StringBuffer into an array, use the getChars( ) method

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.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.

- ✔ The **append( )** method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append(Object obj)

- ✔ First, valueOf( ) is called for each parameter to obtain its string representation. The result is then appended to the current StringBuffer object. Finally, 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(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s);
 }
 }
```

The output of this example is shown here: a = 42!

- ✔ A concatenation or concat() invokes append( ) on a StringBuffer object. After the concatenation has been performed, the compiler inserts a call to toString( ) to turn the modifiable StringBuffer back into a constant String

- ✔ The **insert( )** method inserts one string into another,

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, unlike append() which only appends at the end of the stringbuffer object.

- ✔ 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.

```
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 the characters within a StringBuffer object using **reverse( ),** This method returns the reversed object on which it was called.

```
// Using reverse() to reverse a StringBuffer
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

✔ You can delete characters within a StringBuffer by using the methods **delete( ) and deleteCharAt( ).**

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

```
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);
```

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

✔ 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)

```
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.**

✔ You can obtain a portion of a StringBuffer by calling substring( ).

String substring(int startIndex)
String substring(int startIndex, int endIndex)
// start till |endIndex-1|

int indexOf(String str)
int indexOf(String str, int startindex)
int lastIndexOf(String str
int lastIndexOf(String str, int startIndex)

// returns the first or last occurrence of the string

void trimToSize( )

Reduces the size of the character buffer for the invoking object to exactly fit the current contents.

✔ 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.

--- NEXT CHAPTER ON NEXT PAGE ---

Exploring Java.lang Package

Commonly Used Classes and Interfaces

✔ Float Object Methods:

static int compare(float *num1*, float *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Float *f*)	Compares the numerical value of the invoking object with that of *f*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *FloatObj*)	Returns **true** if the invoking **Float** object is equivalent to *FloatObj*. Otherwise, it returns **false**.

float floatValue( )	Returns the value of the invoking object as a **float**.

boolean isNaN( )	Returns **true** if the invoking object contains a value that is not a number. Otherwise, it returns **false**.
static boolean isNaN(float *num*)	Returns **true** if *num* specifies a value that is not a numbe Otherwise, it returns **false**.
long longValue( )	Returns the value of the invoking object as a **long**.

int intValue( )	Returns the value of the invoking object as an **int**.

String toString( )	Returns the string equivalent of the invoking object.
static String toString(float *num*)	Returns the string equivalent of the value specified by *num*.
static Float valueOf(float *num*)	Returns a **Float** object containing the value passed in *num*.

Methods Defined by Double

byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(double *num1*, double *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Double *d*)	Compares the numerical value of the invoking object with that of *d*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.

- isInfinite( ) returns true if the value being tested is infinitely large or small in magnitude.
- isNaN( ) returns true if the value being tested is not a number.

```
public static void main(String args[]) {
 Double d1 = new Double(1/0.);
 Double d2 = new Double(0/0.);
d1.isInfinite();
 returns true
d1.isNaN();
returns true
```

Methods Defined by Integer

Method	Description
static int numberOfTrailingZeros(int *num*)	Returns the number of low-order zero bits that precede the first low-order set bit in *num*. If *num* is zero, 32 is returned.
static int parseInt(String *str*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using radix 10.
static int parseInt(String *str*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using the specified *radix*.
static int reverse(int *num*)	Reverses the order of the bits in *num* and returns the result.
static int reverseBytes(int *num*)	Reverses the order of the bytes in *num* and returns the result.
static int rotateLeft(int *num*, int *n*)	Returns the result of rotating *num* left *n* positions.
static int rotateRight(int *num*, int *n*)	Returns the result of rotating *num* right *n* positions.
static int signum(int *num*)	Returns –1 if *num* is negative, 0 if it is zero, and 1 if it is positive.
short shortValue( )	Returns the value of the invoking object as a **short**.
static String toBinaryString(int *num*)	Returns a string that contains the binary equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(int *num*)	Returns a string that contains the decimal equivalent of *num*.
static String toString(int *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static Integer valueOf(int *num*)	Returns an **Integer** object containing the value passed in *num*.
static Integer valueOf(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static Integer valueOf(String *str*, int *radix*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str* using the specified *radix*.

TABLE 16-5 The Methods Defined by **Integer** *(continued)*

- ✔ The Byte, Short, Integer, and Long classes provide the parseByte( ), parseShort( ), parseInt( ), and parseLong( ) methods, respectively.
- ✔ These methods return the byte, short, int, or long equivalent of the numeric string with which they are called.
- ✔ If a list of integers is entered by the user Which is read by readline() , then we need to use parseInt( ) to convert these strings into their int equivalents.

/* This program sums a list of numbers entered by the user.

It converts the string representation of each number into an int using parseInt(). */

```
import java.io.*;
class ParseDemo {
public static void main(String args[]) throws IOException {
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str; int i; int sum=0;
 System.out.println("Enter numbers, 0 to quit.");
 do {
 str = br.readLine();
try {
i = Integer.parseInt(str);
 }
 catch(NumberFormatException e) {
System.out.println("Invalid format");
 i = 0;
}
sum += i;
System.out.println("Current sum is: " + sum);
 } while(i != 0);
 }
 }
```

✔ Conversly, To convert a whole number into a decimal string, use the versions of toString( ) defined in the Byte, Short, Integer, or Long classes.

Methods Defined by Character

✔ Character is a simple wrapper around a char, To obtain the char value contained in a Character object, call charValue( )

✔ char charValue( ) It returns the character

Method	Description
static boolean isDefined(char *ch*)	Returns **true** if *ch* is defined by Unicode. Otherwise, it returns **false**.
static boolean isDigit(char *ch*)	Returns **true** if *ch* is a digit. Otherwise, it returns **false**.
static boolean isIdentifierIgnorable(char *ch*)	Returns **true** if *ch* should be ignored in an identifier. Otherwise, it returns **false**.
static boolean isISOControl(char *ch*)	Returns **true** if *ch* is an ISO control character. Otherwise, it returns **false**.
static boolean isJavaIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Java identifier (other than the first character). Otherwise, it returns **false**.
static boolean isJavaIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Java identifier. Otherwise, it returns **false**.
static boolean isLetter(char *ch*)	Returns **true** if *ch* is a letter. Otherwise, it returns **false**.
static boolean isLetterOrDigit(char *ch*)	Returns **true** if *ch* is a letter or a digit. Otherwise, it returns **false**.
static boolean isLowerCase(char *ch*)	Returns **true** if *ch* is a lowercase letter. Otherwise, it returns **false**.
static boolean isUpperCase(char *ch*)	Returns **true** if *ch* is an uppercase letter. Otherwise, it returns **false**.
static boolean isWhitespace(char *ch*)	Returns **true** if *ch* is whitespace. Otherwise, it returns **false**.
static char toLowerCase(char *ch*)	Returns lowercase equivalent of *ch*.
static char toTitleCase(char *ch*)	Returns titlecase equivalent of *ch*.
static char toUpperCase(char *ch*)	Returns uppercase equivalent of *ch*.

✔ **TABLE 16-7** Various **Character** Methods

✔

✔ Boolean is a very thin wrapper around boolean values, which is useful mostly when you want to pass a boolean variable by reference

✔ Boolean 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

Method	Description
boolean booleanValue( )	Returns **boolean** equivalent.
int compareTo(Boolean *b*)	Returns zero if the invoking object and *b* contain the same value. Returns a positive value if the invoking object is **true** and *b* is **false**. Otherwise, returns a negative value.
boolean equals(Object *boolObj*)	Returns **true** if the invoking object is equivalent to *boolObj*. Otherwise, it returns **false**.

- ✔ **TABLE 16-9** The Methods Defined by **Boolean**

Method	Description
static boolean getBoolean(String *propertyName*)	Returns **true** if the system property specified by *propertyName* is **true**. Otherwise, it returns **false**.
int hashCode( )	Returns the hash code for the invoking object.
static boolean parseBoolean(String *str*)	Returns **true** if *str* contains the string "true". Case is not significant. Otherwise, returns **false**.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(boolean *boolVal*)	Returns the string equivalent of *boolVal*.
static Boolean valueOf(boolean *boolVal*)	Returns the **Boolean** equivalent of *boolVal*.
static Boolean valueOf(String *boolString*)	Returns **true** if *boolString* contains the string "true" (in uppercase or lowercase). Otherwise, it returns **false**.

- ✔ **TABLE 16-9** The Methods Defined by **Boolean** *(continued)*
- ✔ Although Java provides automatic garbage collection, sometimes you will want to know how large the object heap is and how much of it is left
- ✔ You can use this information, for example, to check your code for efficiency or to approximate how many more objects of a certain type can be instantiated.
- ✔ To obtain these values, use the totalMemory( ) and freeMemory( ) methods.
- ✔ Java's garbage collector runs periodically to recycle unused objects.
- ✔ But, You can run the garbage collector on demand by calling the gc( ) method.

- call gc( ) and then call freeMemory( ) to get a baseline memory usage or , execute your code and call freeMemory( ) again to see how much memory it is allocating.

```
public static void main(String args[]) {
 Runtime r = Runtime.getRuntime();
long mem1, mem2;
 Integer someints[] = new Integer[1000];
System.out.println("Total memory is: " + r.totalMemory());
System.out.println("Initial free memory: " + r.freeMemory());
r.gc(); // requesting garbage collection
mem1 = r.freeMemory();
System.out.println("Free memory after garbage collection: " + mem1);
// allocate integers
mem2 = r.freeMemory(); // free memory after allocation
System.out.println("Memory used by allocation: " + (mem1-mem2));
```

- Several forms of the exec( ) method allow you to name the program you want to run as well as its input parameters.

- The exec( ) method returns a Process object, which can then be used to control how your Java program interacts with this new running process.

- The Process object returned by exec( ) can be manipulated by Process' methods after the new program starts running.

- You can kill the subprocess with the destroy( ) method.

- The waitFor( ) method causes your program to wait until the subprocess finishes. The exitValue( ) method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur

```
// Demonstrate exec().
class ExecDemo {
 public static void main(String args[]) {
Runtime r = Runtime.getRuntime();
 Process p = null; try {
 p = r.exec("notepad");
 } catch (Exception e) {
 System.out.println("Error executing notepad.");
```

```
}
System.out.println("Notepad returned " + p.exitValue());
}
}
```

- ✔ ProcessBuilder provides another way to start and manage processes or programs

- ✔ all processes are represented by the Process class, and a process can be started by Runtime.exec( ).

- ✔ To create a process using ProcessBuilder, simply create an instance of ProcessBuilder, specifying the name of the program and any needed arguments.

- ✔ To begin execution of the program, call start( ) on that instance

```
class PBDemo {
public static void main(String args[]) {
try {
ProcessBuilder proc = new ProcessBuilder("notepad.exe", "testfile"); proc.start();
}
 catch (Exception e) {
System.out.println("Error executing notepad.");
}
 }
 }
```

- ✔ The Methods Defined by System –

static void gc( )	Initiates garbage collection.
static void setErr(PrintStream *eStream*)	Sets the standard **err** stream to *eStream*.
static void setIn(InputStream *iStream*)	Sets the standard **in** stream to *iStream*.
static void setOut(PrintStream *oStream*)	Sets the standard **out** stream to *oStream*.

static void runFinalization( )	Initiates calls to the **finalize( )** methods of unused but not yet recycled objects.

✔ You can obtain the values of various environment variables by calling the System.getProperty( ) method. For example, the following program displays the path to the current user directory:

```
class ShowUserDir {
public static void main(String args[]) {
System.out.println(System.getProperty("user.dir"));
}
 }
```

✔ Math Class, defines two double constants: E (approximately 2.72) and PI (approximately 3.14).

Some General purpose methods are as given below:-

Exponential Functions

Math defines the following exponential methods:

Method	Description
static double cbrt(double *arg*)	Returns the cube root of *arg*.
static double exp(double *arg*)	Returns e to the *arg*.
static double expm1(double *arg*)	Returns e to the *arg*–1
static double log(double *arg*)	Returns the natural logarithm of *arg*.
static double log10(double *arg*)	Returns the base 10 logarithm for *arg*.
static double log1p(double *arg*)	Returns the natural logarithm for *arg* + 1.
static double pow(double *y*, double *x*)	Returns *y* raised to the *x*; for example, pow(2.0, 3.0) returns 8.0.
static double scalb(double *arg*, int *factor*)	Returns $val \times 2^{factor}$. (Added by Java SE 6.)
static float scalb(float *arg*, int *factor*)	Returns $val \times 2^{factor}$. (Added by Java SE 6.)
static double sqrt(double *arg*)	Returns the square root of *arg*.

ROUNDING METHODS DEFINED BY MATH

Method	Description
static int abs(int *arg*)	Returns the absolute value of *arg*.

static double ceil(double *arg*)	Returns the smallest whole number greater than or equal to *arg*.
static double floor(double *arg*)	Returns the largest whole number less than or equal to *arg*.
static int max(int *x*, int *y*)	Returns the maximum of *x* and *y*.

static float min(float *x*, float *y*)	Returns the minimum of *x* and *y*.
static double min(double *x*, double *y*)	Returns the minimum of *x* and *y*.
static double nextAfter(double *arg*, double *toward*)	Beginning with the value of *arg*, returns the next value in the direction of *toward*. If *arg* == *toward*, then *toward* is returned. (Added by Java SE 6.)

static double nextUp(double *arg*)	Returns the next value in the positive direction from *arg*. (Added by Java SE 6.)

static double rint(double *arg*)	Returns the integer nearest in value to *arg*.
static int round(float *arg*)	Returns *arg* rounded up to the nearest **int**.

static double random( )	Returns a pseudorandom number between 0 and 1.
static float signum(double *arg*)	Determines the sign of a value. It returns 0 if *arg* is 0, 1 if *arg* is greater than 0, and –1 if *arg* is less than 0.

✔ The Runtime class encapsulates the run-time environment. You cannot instantiate a Runtime object. However, you can get a reference to the current Runtime object by calling the static method Runtime.getRuntime( ).

long freeMemory( )	Returns the approximate number of bytes of free memory available to the Java run-time system.
void gc( )	Initiates garbage collection.
static Runtime getRuntime( )	Returns the current **Runtime** object.
void halt(int *code*)	Immediately terminates the Java Virtual Machine. No termination threads or finalizers are run. The value of *code* is returned to the invoking process.

✔

void runFinalization( )	Initiates calls to the **finalize( )** methods of unused but not yet recycled objects.
long totalMemory( )	Returns the total number of bytes of memory available to the program.

✔

✔ **The Runnable interface** must be implemented by any class that will initiate a separate thread of execution. Runnable only defines one abstract method, called run( ), which is the entry point to the thread. It is defined like this: void run( ) Threads that you create must implement this method.

- Thread creates a new thread of execution, its commonly used forms of constructors are –

  Thread( )

  Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)

- threadOb is an instance of a class that implements the Runnable interface and defines where execution of the thread will begin. The name of the thread is specified by threadName.
- groupOb specifies the thread group to which the new thread will belong. When no thread group is specified, the new thread belongs to the same group as the parent thread

Method	Description
static int activeCount( )	Returns the number of threads in the group to which the thread belongs.
final void checkAccess( )	Causes the security manager to verify that the current thread can access and/or change the thread on which **checkAccess( )** is called.
static Thread currentThread( )	Returns a **Thread** object that encapsulates the thread that calls this method.

-

long getID( )	Returns the ID of the invoking thread.
final String getName( )	Returns the thread's name.
final int getPriority( )	Returns the thread's priority setting.
StackTraceElement[ ] getStackTrace( )	Returns an array containing the stack trace for the invoking thread.
Thread.State getState( )	Returns the invoking thread's state.
final ThreadGroup getThreadGroup( )	Returns the **ThreadGroup** object of which the invoking thread is a member.

-

void interrupt( )	Interrupts the thread.
static boolean interrupted( )	Returns **true** if the currently executing thread has been scheduled for interruption. Otherwise, it returns **false**.

-

Method	Description
final boolean isAlive( )	Returns **true** if the thread is still active. Otherwise, it returns **false**.

-

✔

void run( )	Begins execution of a thread.

✔

final void setName(String *threadName*)	Sets the name of the thread to that specified by *threadName*.
final void setPriority(int *priority*)	Sets the priority of the thread to that specified by *priority*.

✔

void start( )	Starts execution of the thread.
String toString( )	Returns the string equivalent of a thread.
static void yield( )	The calling thread yields the CPU to another thread.

✔ ThreadGroup creates a group of threads. Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads.

✔

final void destroy( )	Destroys the thread group (and any child groups) on which it is called.
int enumerate(Thread *group*[ ])	The threads that comprise the invoking thread group are put into the *group* array.

final int getMaxPriority( )	Returns the maximum priority setting for the group.
final String getName( )	Returns the name of the group.
final ThreadGroup getParent( )	Returns **null** if the invoking **ThreadGroup** object has no parent. Otherwise, it returns the parent of the invoking object.
final void interrupt( )	Invokes the **interrupt( )** method of all threads in the group.

// Start the thread

// Demonstrate thread groups.

class NewThread extends Thread {

boolean suspendFlag; NewThread(String threadname, ThreadGroup tgOb) {

super(tgOb, threadname);

System.out.println("New thread: " + this);

suspendFlag = false; start();

// entry point for thread.

```
public void run() { try
 { for(int i = 5; i > 0; i--) {
 System.out.println(getName() + ": " + i);
 Thread.sleep(1000);
synchronized(this) {
while(suspendFlag) {
wait();
} } } }
 catch (Exception e) {
System.out.println("Exception in " + getName()); }
System.out.println(getName() + " exiting."); }
```

- ✔ The Throwable class supports Java's exception-handling system and is the class from which all exception classes are derived.
- ✔ The StackTraceElement class describes a single stack frame, which is an individual element of a stack trace when an exception occurs. Each stack frame represents an execution point, which includes such things as the name of the class, the name of the method, the name of the file, and the source-code line number.
- ✔ An array of StackTraceElements is returned by the getStackTrace( ) method of the Throwable class.
- ✔ The CharSequence interface defines methods that grant read-only access to a sequence of characters. This interface is implemented by String, StringBuffer, and StringBuilder.
- ✔ Objects of classes that implement Comparable can be ordered. In other words, classes that implement Comparable contain objects that can be compared in some meaningful manner. interface Comparable Here, T represents the type of objects being compared.

- ✔ This interface is implemented by several of the classes , viz. Byte, Character, Double, Float, Long, Short, String, and Integer classes define a compareTo( ) method declared by CharSequence.
- ✔ Objects of a class that implements Appendable interface can have a character or character sequences appended to it.
- ✔ Iterable must be implemented by any class whose objects will be used by the for-each version of the for loop. interface Iterable Here, T is the type of the object being iterated

EXCEPTION HANDLING

- ✔ an exception is a run-time error in the code sequence.
- ✔ exception handling avoids the problem of manually checking codes and handling errors and, in the process, brings run-time error management into the object oriented world.
- ✔ Basics :- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- ✔ exception handling is managed via five keywords: try, catch, throw, throws, and finally.
- ✔ Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.
- ✔ Your code can catch this exception (using catch) and handle it
- ✔ System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw

✔ Any exception that is thrown out of a method must be specified as such by a throws clause

✔ Any code that absolutely must be executed after a try block completes is put in a finally block.

exception-handling block

```
try {
// block of code to monitor for errors
 }
catch (ExceptionType1 exOb) {
 // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
 }
finally {
// block of code to be executed after try block ends
      }
```

✔ All exception types are subclasses of the built-in class Throwable.

✔ Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. Exception. This class is used for exceptional conditions that user programs should catch.

✔ RuntimeException a subclass of Exception branch is automatically defined for the programs that you write and include things such as division by zero and invalid array indexing

✔ The other branch is topped by Error, which defines exceptions that are not expected to be caught by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. For example, Stack overflow

class DivideByZeroExcep

```
{
public static void main(String args[]) {
int d = 0; int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes execution of DivideByZero Class to stop.

✔ Because once an exception has been thrown, it must be caught by an exception handler and dealt with. In absence of an exception handler, Java's default hander returns the entire call stacktrace desrcribing the trpe of error and terminates the program.

✔ Handling an exception yourself, instead of letting the default hander do it has two benefits

✔ First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

✔ To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

```
class Exc2 {
public static void main(String args[]) {
int d, a;
 try { // monitor a block of code. d = 0;
 a = 42 / d;
System.out.println("This will not be printed.");
 }
catch (ArithmeticException e)
{ // catch divide-by-zero error
 System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
 } }
```

This program generates the following output:

```
 Division by zero.
After catch statement.
```

✔ Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

✔ catch clauses should be able to resolve the exceptional condition and then continue on as if the error had never happened.

✔ Sometimes more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

✔ When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

✔ After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

```
public static void main(String args[]) {
try {
int a = args.length; System.out.println("a = " + a);
 int b = 42 / a;
 int c[] = { 1 };
 c[42] = 99;
}
catch(ArithmeticException e)
 {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
```

```
}
```

✔ It it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

OUTPUT:

```
C:\>java MultiCatch TestArg a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

✔ It is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

✔ The try statement can be nested.

✔ it is possible for your program to throw an exception explicitly, using the throw statement.

✔ The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exception.

✔ The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
static void demoproc()
{ try
 {
 throw new NullPointerException("demo");
// creating exception object using new
```

```
}
 catch(NullPointerException e)
{
 System.out.println("Caught inside demoproc.");
 throw e;
// rethrow the exception
}}
public static void main(String args[])
{
try
{
demoproc();
 }
catch(NullPointerException e) {
System.out.println("Recaught: " + e);
 }}}
```

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ).

The demoproc( ) method then sets up another exceptionhandling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. Here is the resulting output: Caught inside demoproc. Recaught: java.lang.NullPointerException: demo

✔ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception otherwise a compile time error will return. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions separated by comma that a method might throw except Error and RuntimeException objects.

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException { System.out.println("Inside
throwOne.");
throw new IllegalAccessException("demo");
 }
public static void main(String args[]) {
try { throwOne(); }
 catch (IllegalAccessException e)
{ System.out.println("Caught " + e); } } }
```

Here is the output generated by running this example program: inside throwOne caught java.lang.IllegalAccessException: demo

- ✔ finally creates a block of code that will be executed after a try/catch block has completed and is written before the code following the try/catch block.
- ✔ The finally block will execute whether or not an exception is thrown
- ✔ It is done to ensure that important lines of code are not bypassed during a try-catch exection. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- ✔ This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- ✔ The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

```
static void procA() {
try{
 System.out.println("inside procA");
 throw new RuntimeException("demo");
 }
 finally {
 System.out.println("procA's finally");
} }
```

- ✔ exceptions that belong to RuntimeException or its subclasses need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. These are checked at runtime.

- ✔ must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. They are checked at compile time.

```
int a=50/0;//ArithmeticException
String s=null;
System.out.println(s.length());//NullPointerException
```

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

✔

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

✔ Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

```
class ChainExcDemo {
static void demoproc() {
 // create an exception NullPointerException
e = new NullPointerException("top layer");
 // add a cause
 e.initCause(new ArithmeticException("cause")); throw e; }

public static void main(String args[]) {
try { demoproc();
} catch(NullPointerException e)
{ // display top level exception
System.out.println("Caught: " + e);
// display cause exception
System.out.println("Original cause: " + e.getCause()); } } }
```

NEXT: CONSTRUCTORS

✔It can be tedious to initialize all of the variables in a class each time an instance is created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor

✔A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

✔Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

- ✔ Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

  `Box mybox1 = new Box();  // create, allocate and initialize box object using Box CTOR`

- ✔ When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of Box that did not define a constructor. The default constructor automatically initializes all instance variables to zero.

✔ Parameterized CTOR :-

```
class Box {
 double width;
double height;
 double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
{ width = w;
 height = h;
 depth = d;
 }

public static void main(String args[]) {
 // declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15); }
```

✔ Sometimes a method will need to refer to the object that invoked it. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

✔ // One use of this - As you know, it is illegal in Java to declare two local variables with the same name inside in the same scope.

✔ you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name: // Use this to resolve name-space collisions. Box(double width, double height, double depth) { this.width = width; this.height = height; this.depth = depth; }

✔ objects are destroyed and their memory released for later reallocation automatically using garbage collection.

✔ when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed

✔ Garbage collection only occurs sporadically during the execution of your program so we do not have to think about it

- ✔ When an object is destroyed, it performs some action. if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- ✔ For this purpose, we need finalization, By using finalization, you can define specific actions beforehand that will occur when an object is just about to be reclaimed by the garbage collector.

- ✔ Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object. Thus, entailing no need of a constructor.
- ✔ The finalize( ) method has this general form:

```
protected void finalize( )
{ // finalization code here }
```

- ✔ Note: your program should provide other means of releasing system resources, etc. if needed by the program, because finalize( ) is only called just prior to garbage collection.

MORE ON BASIC CONCEPTS

- ✔ class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it

- ✔ there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine.

- ✔ Therefore, changes made to the parameter of the subroutine have no effect on the argument.

- ✔ The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine

- ✔ when you pass a primitive type to a method, it is passed by value. But complex types, String and object are passed by reference.

- ✔ recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number.

```
// this is a recursive method
int fact(int n) {
int result; if(n==1) return 1;
result = fact(n-1) * n;
return result;  }
```

- ✔ The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives
- ✔ Methods declared as static have several restrictions: • They can only call other static methods. • They must only access static data. • They cannot refer to this or super in any way
- ✔ It is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static

- ✔ A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared a final variable is essentially a constant.
- ✔ You also use the **final keyword** in a **method** declaration to indicate that the **method** cannot be overridden by subclasses. You can use it on some or all other class's methods.

```
class A {
 final void meth() {
 System.out.println("This is a final method.");
 }
 }
```

- ✔ when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden

```
// Method overriding.
class A { int i, j;
 A(int a, int b)
 { i = a;
 j = b;
 }
 // display i and j
void show() { System.out.println("i and j: " + i + " " + j);
 }
 }
class B extends A
 { int k; B(int a, int b, int c)
{
super(a, b);
 k = c;
}
// display k – this overrides show() in A
void show()
 {
```

```
System.out.println("k: " + k); } }
class Override
 {
 public static void main(String args[]) {
 B subOb = new B(1, 2, 3);
 subOb.show();
 // this calls show() in B
} }
```

The output produced by this program is shown here: k: 3

- ✔ Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism
- ✔ Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- ✔ There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
- ✔ you cannot declare abstract constructors, or abstract static methods

```
// A Simple demonstration of abstract.
 abstract class A { abstract void callme();
class B extends A
{,….}
 public static void main(String args[]) {
 B b = new B(); b.callme();
```

- ✔ To prevent a class from being inherited, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too
- ✔ it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations
- ✔ final class A { // ... } // The following class is illegal.
- ✔ class B extends A { // ERROR! Can't subclass A // ... }

- ✔ Object is a superclass of all other classes

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

✔ The interface, itself, does not actually define any implementation. Although they are similar to abstract classes, interfaces have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

```
// Define an integer stack interface.
interface IntStack {
void push(int item);              // store an item
int pop();                        // retrieve an item }

// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
 private int stck[];
private int tos;
}
 // Push an item onto the stack public void push(int item) {
if(tos==stck.length-1)
// use length member
 System.out.println("Stack is full.");
else stck[++tos] = item; }
```

✔ One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.

✔ When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

- ✔ Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

The synchronization is mainly used to

- ✔ To prevent thread interference. (so threads don't interfere in each other's task)
- ✔ To prevent consistency problem. (so threads do not cause inconsistent state of data)
- ✔ Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- ✔ If you declare any method as synchronized, and When a thread invokes that synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
synchronized void printTable(int n){//synchronized method
final Table obj = new Table();//only one object

Thread t1=new Thread(){
public void run(){
obj.printTable(5)
```

- ✔ Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

```
synchronized(this)
{//synchronized block ……..…….}
```

- ✔ Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.
- ✔ Wait() Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. Notify() and NotifyAll() thread wakes the threads waitig for the resource to resurrect and acquire the resource.

✔ Only one thread at a time can run in a single process. The thread scheduler mainly uses preemptive (higher priority first) or time slicing (predefined slices of time) scheduling to schedule the threads

Distribution of the priority of a thread is between 1 and 10.

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
m1.setPriority(Thread.MIN_PRIORITY);
```

✔ The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

```
Thread.sleep(500);
```

✔ provides servies to the user thread and die when there are none left. It is a low priorirty thread.

```
public void setDaemon(boolean status)
```

To mark the current thread as daemon thread.

✔ **Thread pool** represents a group of worker threads that are waiting for the job and reuse many times. It saves time because there is no need to create new thread. Pick one and use and drop back in the pool.

✔ ThreadGroup is a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

✔ Shutdown hook save the state when JVM shuts down normally or abruptly.

✔ The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
System.gc();  // explicitly calling garbage collection
```

✔ java.util.regex package provide classes and interface for regular expressions like matcher class that is used to perform match operations.

boolean matches()	test whether the regular expression matches the pattern.
boolean find()	finds the next expression that matches the pattern.
boolean find(int start)	finds the next expression that matches the pattern from the given start number.
String group()	returns the matched subsequence.
int start()	returns the starting index of the matched subsequence.
int end()	returns the ending index of the matched subsequence.
int groupCount()	returns the total number of the matched subsequence.

✔ Create a regular expression that accepts alpha numeric characters only. Its length must be 6 characters long only.*/

```
import java.util.regex.*;
class RegexExample6{
public static void main(String args[]){
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));  //true
```

✔ The . (dot) represents a single character.

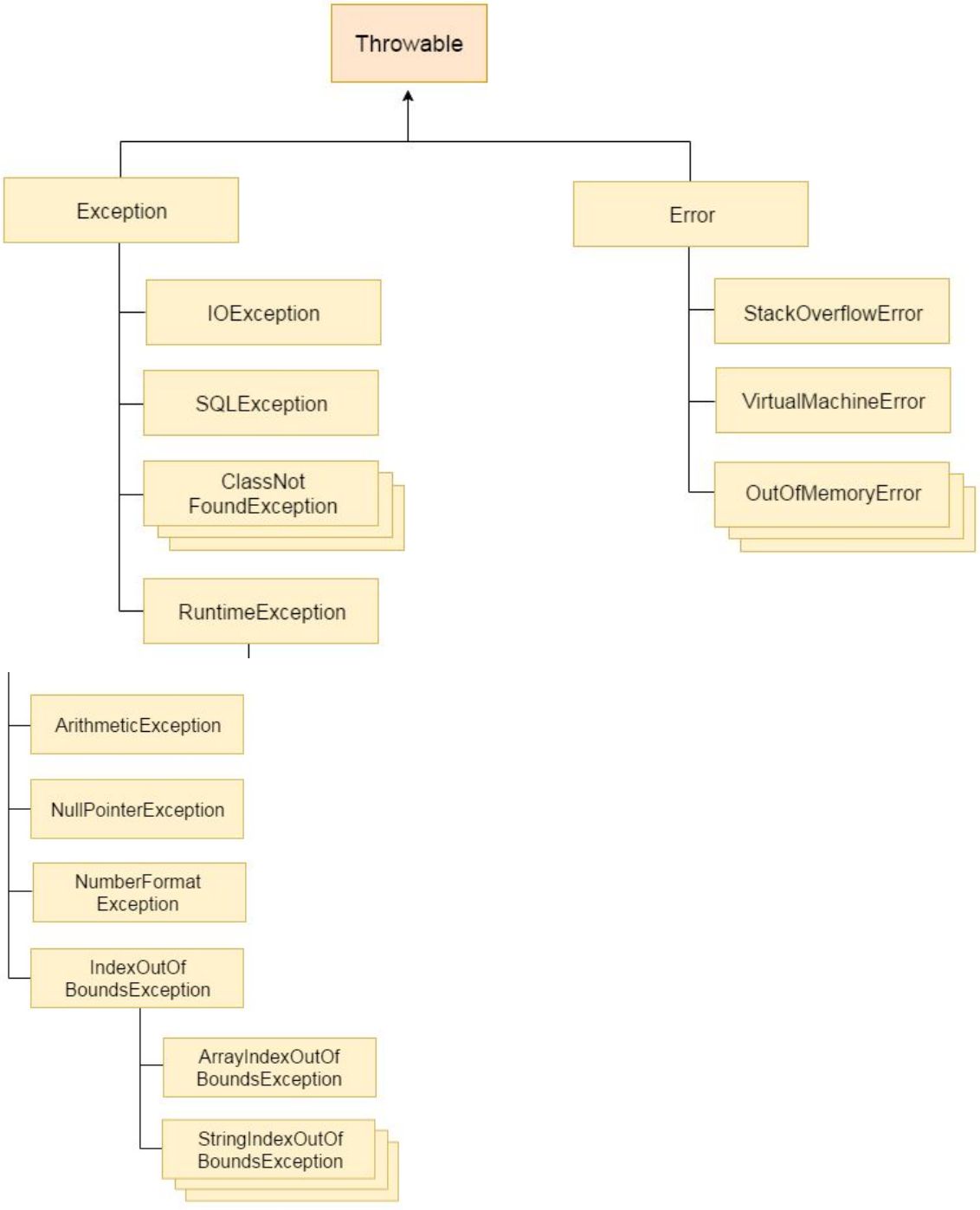
```
System.out.println(Pattern.matches("..s", "mas"));
//true (3rd char is s)
```

[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)

[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)

X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

MULTITHREADED PROGRAMMING

✔ A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

✔ A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently

✔ In a thread-based multitasking environment, a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

✔ In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.

✔ Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. A thread can be running. Or, It can be ready to run as soon as it gets CPU time. A running thread can be suspended or, A suspended thread can then be resumed Or, A thread can be blocked when waiting for a resource or another can be terminated.

✔ a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

✔ A thread can voluntarily relinquish control using sleep or yield or block OR A thread can be preempted by a higher-priority thread. threads of equal priority are time-sliced automatically in round-robin fashion.

✔ if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. Java does this by monitor which acts like a control section, where only one thread can enter at a time. This is implemented using synchronized keywords in methods, or block.

✔ To create a new thread, your program will either extend Thread or implement the Runnable interface.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

✔ The main thread is created automatically when your program is started. t is the thread from which other "child" threads will be spawned. Often, it must be the last thread to finish execution because it performs various shutdown actions

//Code Extract

```
// Controlling the main Thread.
 class CurrentThreadDemo
 {
 public static void main(String args[])
```

```
{
Thread t = Thread.currentThread();
 System.out.println("Current thread: " + t);
// change the name of the thread
 t.setName("My Thread");
 System.out.println("After name change: " + t);
```

✔ In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished: • You can implement the Runnable interface. • You can extend the Thread class, itself

✔ One way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

✔ The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

```
// Create a second thread by extending Thread
class NewThread extends Thread
{
 NewThread()
{ // Create a new, second thread
 super("Demo Thread");
 System.out.println("Child thread: " + this);
 start();
 // Start the thread
 }
// This is the entry point for thread.
 public void run() {……)
```

✔ Two ways exist to determine whether a thread has finished. The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

✔ the method that you will more commonly use to wait for a thread to finish is called join( ). This method waits until the thread on which it is called terminates.

✔ A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

✔ All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

✔ all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

✔ you can't add synchronized to the appropriate methods within the class if the class is created by someone else or the class does not support multithreading. In such a case, to synchronize an object of this class, You simply put calls to the methods defined by this class inside a synchronized block.

```
public void run() { synchronized(target) {
// synchronized block target.call(msg);
}
```

✔ To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods

are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context

✔ wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ). • notify( ) wakes up a thread that called wait( ) on the same object. • notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access

✔ occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

✔ suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked. The resume( ) method is also deprecated. It does not cause problems, but cannot be used without the suspend( ) method as its counterpart. The stop( ) method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

✔ Now, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the run( ) method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate.

✔ The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.

✔ Multithreading can makes program efficient but If you create too many threads, more CPU time will be spent changing contexts than executing your program, which will eventually result in degraded performance.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    suspendFlag = false;
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }


  void mysuspend() {
    suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}

class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.myresume();
```

How to use BufferdReader & Scanner for reading from standard input stream (keyboard)

```
// A tiny editor.
import java.io.*;

class TinyEdit {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    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;
    }
```

Method	Description
String next( )	Returns the next token of any type from the input source.

```
public static void main(String args[]) {
  Scanner conin = new Scanner(System.in);

  int count = 0;
  double sum = 0.0;

  System.out.println("Enter numbers to average.");

  // Read and sum numbers.
  while(conin.hasNext()) {
    if(conin.hasNextDouble()) {
      sum += conin.nextDouble();
      count++;
    }
    else {
      String str = conin.next();
      if(str.equals("done")) break;
      else {
        System.out.println("Data format error.");
        return;
      }
    }
  }
```