CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 1)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ☑ **String Handling**
    - ☑String Constructors
    - ☑String Length
    - ☑Special String Operations

String Handling

- String is a **class** in Java.
- Java implements strings as **objects** of type **String.**
  - The String type is used to **declare string variables**
- Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- A quoted string constant(E.g. “hello”) can be assigned to a **String** variable**.**
- A variable of *type String* can be assigned to another variable of *type String*.

String Constructors

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

String()

  - For example,

String s = new String();

--This will create an instance of **String with no characters in it.**

String Constructors
(initialize array of characters)

- To create a **String initialized by an array of characters**, use the constructor

String(char *chars[ ])

- Example:

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

This constructor initializes **s** with the string **"abc".**

String Constructors

(initialize with a subrange of character array)

- To initialize a string with a subrange of a character array(substring) the following constructor is used:

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

  - Here, *startIndex specifies the staindex at which the subrange begins, and*
  - *numChars specifies* the number of characters to use.

E.g.

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
                 0   1   2   3   4   5
String s = new String(chars, 2, 3);
```

- This initializes s with the characters starting from index 2 and number of letters =3. i.e. s will contain **cde.**

String Constructors
(initialize using another string)

- We can construct a **String** object that contains the same character sequence as another String object using this constructor:

String(String *strObj)*

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

OUTPUT
Java
Java

String Constructors (initialize using byte array)

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

String Constructors
(initialize using byte array) contd.


```
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

String Constructors (contd.)

- We construct a String from a **StringBuffer** by using the constructor :

```
String(StringBuffer strBufObj)
```

- J2SE 5 added two constructors to String.

➢ The first supports the ***extended Unicode character set*** :

```
String(int codePoints[ ], int startIndex, int numChars)
```

– Here, *codePoints is an array that contains Unicode code points*

➢ The second new constructor supports the new **StringBuilder class:-**

```
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 in the string

E.g. length of the string "hello" is 5

- The method **length**() is used to find the length of the string.

int **length**( )

```
class Stringlen
{
    public static void main(String args[])
    {
    String s="Hello";
    System.out.println("Length="+s.length());
    }
}
```

OUTPUT
Length=5

Special String Operations

- These operations include
  - the *automatic creation of new String instances(object)* from **string literals**
  - **concatenation** of multiple String objects by use of the + operator, and
  - the **conversion** of other data types **to a string** representation.

String Literals

- Java automatically constructs a **String** object **for each string literal** in our program,.
  - So we can use a string literal to initialize a **String object**

```
String s2 = "abc";
```

is same as

```
char chars[] = { 'a', 'b', 'c' };
String s2 = new String(chars);
```

- We can use a string literal at any place where we use a **String object.**
- String literals can call the length( ) method on the string

E.g.

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

String Concatenation

- String concatenation is used to join two strings
- Method 1:The + operator can be used between strings to combine them. This is called concatenation.

  ❑Operator + can be chained to concatenate many strings

```
String age = "9";

String s = "He is " + age + " years old.";

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

- This fragment displays the string **He is 9 years old.**
- Instead of letting long strings wrap around within our source code, we can break them into smaller pieces, using the **+ to concatenate them**

String Concatenation with Other Data Types

- We can concatenate strings with other types of data.
- If one of the **operand of the + is an instance of String** then compiler will convert other operand to its string equivalent.

```
String s = "four: " + 2 + 2;

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

  - This fragment displays
    four: 22
  - Operator precedence causes the concatenation of "four" with 2. So 2is converted into string and "four: " concatenates with string equivalent of 2.
  - Then this result is then concatenated with the string equivalent of 2.
- Parentheses can be used for grouping integers and + to perform addition.

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

  - Here parentheses is first computed. So (2+2) is 4 then string "four: " is concatenated with that. So **s** contains the string **"four: 4"**

String Concatenation(contd.)

- Method 2:We can use **concat()** method to concatenate two strings.

  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 :

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

String Conversion and toString( )

- When Java **converts** data into its string representation during concatenation, it calls one of the overloaded versions of the string conversion method **valueOf( )** by class **String.**
- **valueOf( )** is overloaded for all the simple types and for type Object
  - For the simple types, **valueOf( ) returns a string that contains the human-readable equivalent of the value** with which it is called.
  - For objects, **valueOf( ) calls the toString( )** method on the Object.

String Conversion and toString( )

The valueOf() returns the string representation of the corresponding argument. Different overloaded form of valueOf() in String class.

- **valueOf(boolean b)** – Returns the string representation of boolean argument.
- **valueOf(char c)** – char argument.
- **valueOf(char[] data)** char array argument.
- **valueOf(char[] data, int offset, int count)** – specific subarray of the char array argument.
- **valueOf(double d)** – double argument.
- **valueOf(float f)** – float argument.
- **valueOf(int i)** – int argument.
- **valueOf(long l)** – long argument.
- **valueOf(Object obj)** – Object argument. (calls toString() method of the class Object(parent class of all classes n Java)

toString( )(contd.)

- The **toString**( ) method has this general form:

```
String toString( )
```

- When we try to print an object of a class, it will call method valueOf(object) which calls toString( ) function :-
  - if toString( ) is present (overridden) in the class, then it is called.
  - If there is no toString( ) function in the class, when we try to print an object of that class, it prints **clasname@the memory location of the object**(the hexidecimal address of where that **object** is stored in memory.)

Without using toString()

```
class Box {
double width;
double height;
double depth;
Box(double w, double d, double h,) {
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, 14,12);
String s = "Box b: "+ b;

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

OUTPUT
Box@106d69c
Box b: **Box**@1db9742

Here when we print the object **b** ,since there is no toString( ) function in the class it will call toString() in class Object and prints **clasname@the memory location of the object**
(Here it prints **Box**@106d69c)

Using toString()

```
class Box {
double width;
double height;
double depth;
Box(double w, double d, double h,)
{
width = w;
depth = d;
height = h;
}
public String toString()
{
return "Dimensions are " + width + " by " + depth + " by " + height + ".";
}
}
```

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

System.out.println(b);
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

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

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 2)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ☑ **String Handling**
    - ☑ Character Extraction
    - ☑String Comparison
    - ☑Searching Strings
    - ☑Modifying Strings
    - ☑Using valueOf( )
    - ☑Comparison of String Buffer and String.

Character Extraction

- The String class provides the following methods through which characters can be extracted from a String object.

  charAt( )

  getChars( )

  getBytes( )

  toCharArray( )

Character Extraction(contd.)

charAt( )

- Used to **extract a single character** from a String
- we can refer directly to an individual character via the charAt( ) method.
- General form:

char **charAt**(int *where)*

Here *where* is the index of the character that you want to obtain.

e.g.

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

This assigns the value "**b**" to variable **ch.**

	abc
index	**012**

Character Extraction(contd.)

getChars( )

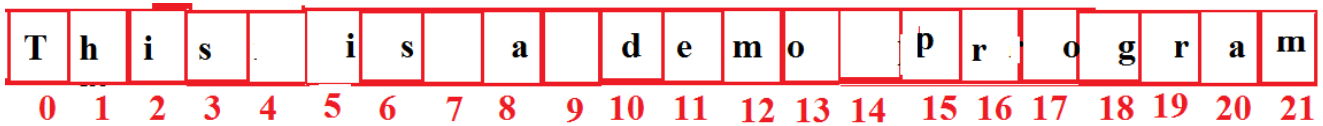
- Used to **extract more than one character** at a time.
- General form:

```
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 up to which character need to be extracted.
  - (the extracted substring contains the characters from *sourceStart* through *sourceEnd–1.)*
- This extracted substring is stored at ***target*** *array at location targetStart*.

Example program- getChars( )

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



This program will extract characters in string **s** from index 10 to 14-1(13) and store in character array **buf** and prints it.

OUTPUT
demo

Character Extraction(contd.)

getBytes( )

- Used to **extract the characters in an array of bytes.**
  - it uses the default character-to-byte conversions.
  - General form

```
byte[ ] getBytes( )
```

  - Most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

Character Extraction(contd.)

toCharArray( )

- Used **to convert all the characters in a String object into a character array.**
  - It returns an array of characters for the entire string.
- General form:

```
char[ ] toCharArray( )
```

Example program -toCharArray( )

```
public class CharArrayEg{
   public static void main(String args[]){
      String str = new String("Welcome to OOP");
      char[] a= str.toCharArray();
      System.out.print("Content of a is:");
      for(char c: a){
         System.out.print(c);
      }
   }
}
```

OUTPUT
Content of a is: Welcome to OOP

String Comparison

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

String Comparison(contd.)

equals( )

- To compare two strings for equality, use **equals( )**
- General form:

boolean **equals**(Object *str*)

  - Here, String object *str* is 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.

String Comparison(contd.)

equalsIgnoreCase( )

- This perform a comparison that **ignores case differences**(not case sensitive)
- When it compares two strings, it considers A-Z to be the same as a-z.
- General form:

boolean **equalsIgnoreCase**(String *str)*

String Comparison(contd.)

```
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 + " is " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " is " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " is " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " is " +s1.equalsIgnoreCase(s4));
}
}
```

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

String Comparison(contd.)

regionMatches( )

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

General forms:

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

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

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

String Comparison(contd.)

- **startsWith( ) and endsWith( )**
- 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.

General forms:

boolean **startsWith**(String *str)*

boolean **endsWith**(String *str)*

System.out.println(“Football".endsWith("ball"));

This prints **true**. (because *bar* comes at the end of string *Football*

System.out.println(" Football ".startsWith("Foo"));

This prints **true**. (because *Foo* comes at the beginning of string *Football*

String Comparison(contd.)

- A second form of startsWith( ), 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,

System.println("Football".startsWith("ball", 4));

  - This prints **true**.

equals( ) Versus ==

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

```
class EqualsNotEqualTo
{
public static void main(String args[])
{
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " is " + s1.equals(s2));
System.out.println(s1 + " = = " + s2 + " is " + (s1 ==s2));
}
}
```

OUTPUT

Hello equals Hello is true

Hello = = Hello is false

compareTo( )

- A string is **less than** another if it **comes before** the other in dictionary order.
  - E.g. “ant”<“bat” (ant comes beforr bat in dictionary)
- A string is **greater than** another if it **comes after** the other in dictionary order.
  - E.g. “bat”>“ant” (bat comes after ant in dictionary)
- **compareTo( )** method in String is used for comparing two strings. General form:

```
int compareTo(String str)
```

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

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

The uppercase letter has a lower value in the ASCII character set than lowercase letters.

Example program - compareTo( )


```
class CompareToEg
{
public static void main(String args[]) {
    String s1="ant";
    String s2="bat";
    if(s1.compareTo(s2) < 0)
    {System.out.println(s1 + " comes before "+s2);
    }
    else if(s1.compareTo(s2) > 0)
    {System.out.println(s1 + " comes after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);
}
}
```

OUTPUT
ant comes before bat

Bubblesort to sort strings


```
class SortString {
static String arr[] = {"This", "is", "best", "time", "for", "all"};
public static void main(String args[]) {
for(int i = 0; i < arr.length; i++)
{
    for(int j = i + 1; j < arr.length; j++)
     {
          if(arr[j].compareTo(arr[i]) < 0)
          {
          String temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
          }
    }
System.out.println(arr[i]);
}
}
}
```

OUTPUT

This
all
best
for
is
time

compareToIgnoreCase( )

- **compareToIgnoreCase( )** method is not case sensitive.

int **compareToIgnoreCase**(String *str)

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

- .

Using **compareToIgnoreCase**

```
class CompareToIgnoreEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareToIgnoreCase(s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToIgnoreCase(s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);

}
}
```

OUTPUT

ant is before Hat

Using **compareTo**

```
class CompareToEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareTo (s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToI (s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is  same as "+s2);
    }
}
```

OUTPUT

ant is after Hat

Searching Strings

- The **String** class provides two methods 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**. If the character or substring is **not found** then these method returns -1.

Searching Strings(contd.)

- 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 searched.*

- 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*.

Searching Strings(contd.)

- We can specify a **starting point for the search** using :

  int indexOf(char *ch, int startIndex)*

  int lastIndexOf(char *ch, int startIndex)*

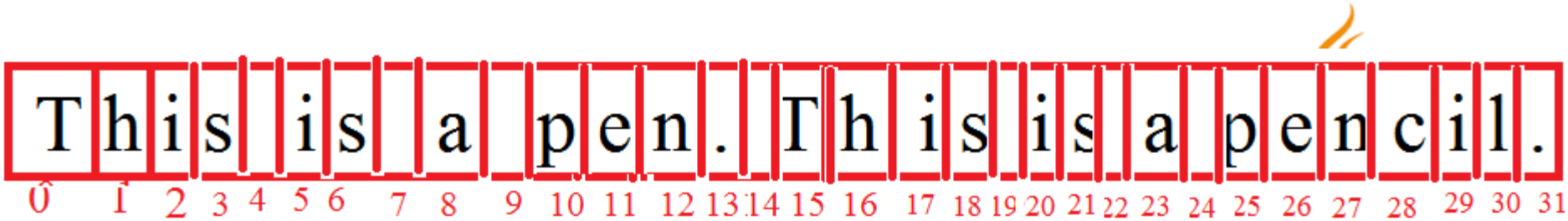
  int indexOf(String *str, int startIndex)*

  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.*

Searching Strings(contd.)


```
class indexOfDemo {
public static void main(String args[]) {
String s = “This is a pen. This is a pencil.";
System.out.println(s);
System.out.println("indexOf(i) = " +s.indexOf('i'));
System.out.println("lastIndexOf(i) = " +s.lastIndexOf('i'));
System.out.println("indexOf(This) = " +s.indexOf("This"));
System.out.println("lastIndexOf(This) = " +s.lastIndexOf("This"));
System.out.println("indexOf(i, 10) = " +s.indexOf('i', 10));
System.out.println("lastIndexOf(i, 23) = " + s.lastIndexOf('i', 23));
System.out.println("indexOf(This, 10) = " + s.indexOf("This", 10));
System.out.println("lastIndexOf(This, 13) = " + s.lastIndexOf("This", 13));
}
}
```



This is a pen. This is a pencil.

indexOf(i) = 2

lastIndexOf(i) = 29

indexOf(This) = 0

lastIndexOf(This) = 15

indexOf(i, 10) = 17

lastIndexOf(i, 23) = 20

indexOf(This, 10) = 15

lastIndexOf(This, 13) = 0

Modifying a String

- **String objects are immutable(cannot change a string.)**
- To modify a String, we must either
  - copy it into a StringBuffer or StringBuilder, or
  - use one of the following String methods:

    substring( )

    concat( )

    replace()

    trim( )

Modifying a String(contd.)

substring( ).

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

```
String substring(int startIndex)
```

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

```
String substring(int startIndex, int endIndex)
```

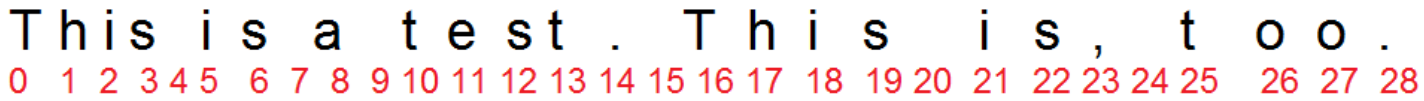
    - Here, *startIndex* specifies the beginning index, and endIndex specifies the stopping point.
    - The string returned contains all the characters from the beginning index, up to, but **not including, the ending index.**

Modifying a String(contd.)

```
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 {
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.
```



Modifying a String(contd.) concat()

concat( )

- We can use **concat()** method to concatenate two strings.

  String concat(String *str)*

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

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

  - Here s1 is the invoking string that call the function concat() .
  - s1 contains "one" and is concatenated with argument string value "two" and form the string "onetwo". This result is stored in the String object **s2**

Modifying a String(contd.) replace( )

replace( ) -The **replace( ) method has two forms.**

1. The first form **replaces all occurrences of one character in** the invoking string with another character.

   String replace(char *original, char replacement)*

   – Here, *original* specifies the character that will be replaced by the character specified by *replacement.*

   – The resulting string is returned.

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

   – Here letter l is replaced by w. So "Hewwo" is put into String object s.

2. The second form of **replace( ) replaces one character sequence with another.**

String replace(CharSequence *original, CharSequence replacement)*

This form was added by J2SE 5.

Modifying a String(contd.) replace( )

replace( ) -The **replace( ) method has two forms.**

1. The first form **replaces all occurrences of one character in** the invoking string with another character.

   String replace(char *original, char replacement)*

   – Here, *original* specifies the character that will be replaced by the character specified by *replacement.*

   – The resulting string is returned.

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

   – Here letter l is replaced by w. So "Hewwo" is put into String object s.

2. The second form of **replace( ) replaces one character sequence with another.**

String replace(CharSequence *original, CharSequence replacement)*

**This form was added by J2SE 5.

Modifying a String(contd.) trim( )

trim( )

- The **trim( )** method returns a copy of the invoking string after **removing any leading and trailing whitespace**
- General form:

String trim( )

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

  - This puts the string "Hello World" into **s.**
- The **trim( )** method is quite useful when we process user commands.

- **E.g.** Write a program that prompts the user to **enter the name** of a state(Assam,Goa etc) and then **displays that state's capital**. Use **trim( )** to *remove any leading or trailing whitespace* that may have inadvertently been entered by the user.

```
import java.io.*;
class UseTrim {
public static void main(String args[]) throws
                                        IOException
{
BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
do {
System.out.println("Enter the State: ");
str = br.readLine();
str = str.trim();
if(str.equals("Assam"))
      System.out.println("Capital is  Dispur");
else if(str.equals("Goa"))
      System.out.println("Capital is Panaji");
else if(str.equals("Bihar"))
      System.out.println("Capital is Patna.");
else
System.out.println("Capital is not entered");
      } while(!str.equals("stop"));
  }
}
```

Data Conversion Using valueOf( )

- The valueOf( ) method **converts** data from its internal format **into a human-readable form.**
- It is a **static** method**.**
- **valueOf( )** is overloaded for all the simple types and for type Object
  - For the simple types, **valueOf( ) returns a string that contains the human-readable equivalent of the value** with which it is called.
  - For objects, **valueOf( ) calls the toString( )** method on the Object.

valueOf()(contd.)

The valueOf() **returns the string representation of the corresponding argument.** Different overloaded form of valueOf() in String class.

- **valueOf(boolean b)** – Returns the string representation of boolean argument.
- **valueOf(char c)** – char argument.
- **valueOf(char[] data)** char array argument.
- **valueOf(char[] data, int offset, int count)** – specific subarray of the char array argument.
- **valueOf(double d)** – double argument.
- **valueOf(float f)** – float argument.
- **valueOf(int i)** – int argument.
- **valueOf(long l)** – long argument.
- **valueOf(Object obj)** – Object argument. (calls toString() method of the class Object(parent class of all classes n Java)

valueOf()(contd.)

- **valueOf( )** is called when **a string representation of some other type of data is needed**
  - example, during concatenation operations
- Any object that we pass to **valueOf( )** will return the result of a call to the object's **toString( )** method**.**
- For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type.
- For arrays of **char,** however, a String object is created that contains the characters in the char array

Changing the Case of Characters Within a String

- String toLowerCase( )
- String toUpperCase( )

```
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);
}
}
```

OUTPUT

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

Comparison of String Buffer and String.

- **StringBuffer** is a **peer class of String** that provides much of the functionality of strings.
- **String** represents **fixed-length, immutable character** sequences.
- **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.

String

- **String** is immutable.
- **String** represents **fixed-length, immutable character** sequences.
- Concatenation using String is slow.
- String class can override equals() method.

StringBuffer

- **StringBuffer** is **mutable.**
- **StringBuffer** represents **growable** and **writeable** character sequences
- Concatenation using StringBuffer is fast.
- StringBuffer class doesnot override equals() method.

String str = "Hello World";

str = "Hi World!";

- Here an object is created using string literal “Hello World”.
- In second statement when we assigned the new string literal “Hi World!” to str, the **object itself didn’t change** instead a **new object got created in memory** using string literal “Hi World!” and the reference to it is assigned to str.

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.

StringBuffer(contd.)

length( ) and capacity( )

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

int length( )
int 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

Here capacity is 21 because room for 16 additional characters is automatically added to value Hello

StringBuffer(contd.)

ensureCapacity( )

- **ensureCapacity**( ) is used to set the size of the buffer.
- This is useful if we know in advance that we will be appending a large number of small strings to a **StringBuffer.**

```
void ensureCapacity(int capacity)
```

- Here, *capacity specifies the size of the buffer.*

StringBuffer(contd.)

setLength( )

- Used to set the length of the buffer within a **StringBuffer object.**

```
void setLength(int len)
```

Here *len* specifies the length of the buffer. This value must be nonnegative.

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

StringBuffer(contd.)

- **charAt( )** and **setCharAt( )**
- The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method.
- We can set the value of a character within a **StringBuffer** using **setCharAt( ).**

char charAt(int *where)*

void setCharAt(int *where, char ch)*

StringBuffer(contd.)


```
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
```

StringBuffer(contd.)

- **getChars( )**
- Used to copy a substring of a **StringBuffer.**

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

StringBuffer(contd.)

append( )

- 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)*

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

StringBuffer(contd.)


```
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);
}
}
```

Output

a = 42!

StringBuffer(contd.)

insert( )

- The **insert( )** method inserts one string into another.
- It calls **String.valueOf( ).**
- This string is then inserted into the invoking **StringBuffer object.**

StringBuffer insert(int *index, String str)*
StringBuffer insert(int *index, char ch)*
StringBuffer insert(int *index, Object obj)*

StringBuffer(contd.)


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

OUTPUT

I like Java!

StringBuffer(contd.)

reverse( )

- We can reverse the characters within a **StringBuffer** object using **reverse( ):**

StringBuffer **reverse**( )

```
class ReverseDemo {

public static void main(String args[]) {

StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

OUTPUT

abcdef

fedcba

StringBuffer(contd.)

delete( ) and **deleteCharAt( )**

- We can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**

StringBuffer delete(int *startIndex, int endIndex)*

StringBuffer deleteCharAt(int *loc)*

- **delete()** deletes from *startIndex to endIndex–1*.
- The **deleteCharAt( )** method deletes the character at the index specified by *loc*

StringBuffer(contd.)


```
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.

StringBuffer(contd.)

replace( )

- We can replace **one set of characters** with another set inside a **StringBuffer** object by calling **replace( ).**

StringBuffer **replace**(int *startIndex, int endIndex, String str)*

  - The substring at *startIndex through endIndex–1 is replaced.*

```
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);
}
}
```

OUTPUT

After replace: This was a test

StringBuffer(contd.)

substring( )

- We can obtain a portion of a **StringBuffer** by calling **substring( ).**

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*.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 3)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ❑ **Collections framework**
    - ❑ Collections overview
    - ❑ Collections Interfaces- Collection Interface

Collections Framework

- The **java.util** package contains one of Java's most powerful subsystems: The ***Collections Framework.***

- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology(best possible technlogy) for **managing groups of objects**.

  - The **Collection** in **Java** is a **framework** that provides an architecture to **store and manipulate the group of objects**.

  - **Java Collection framework** provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)

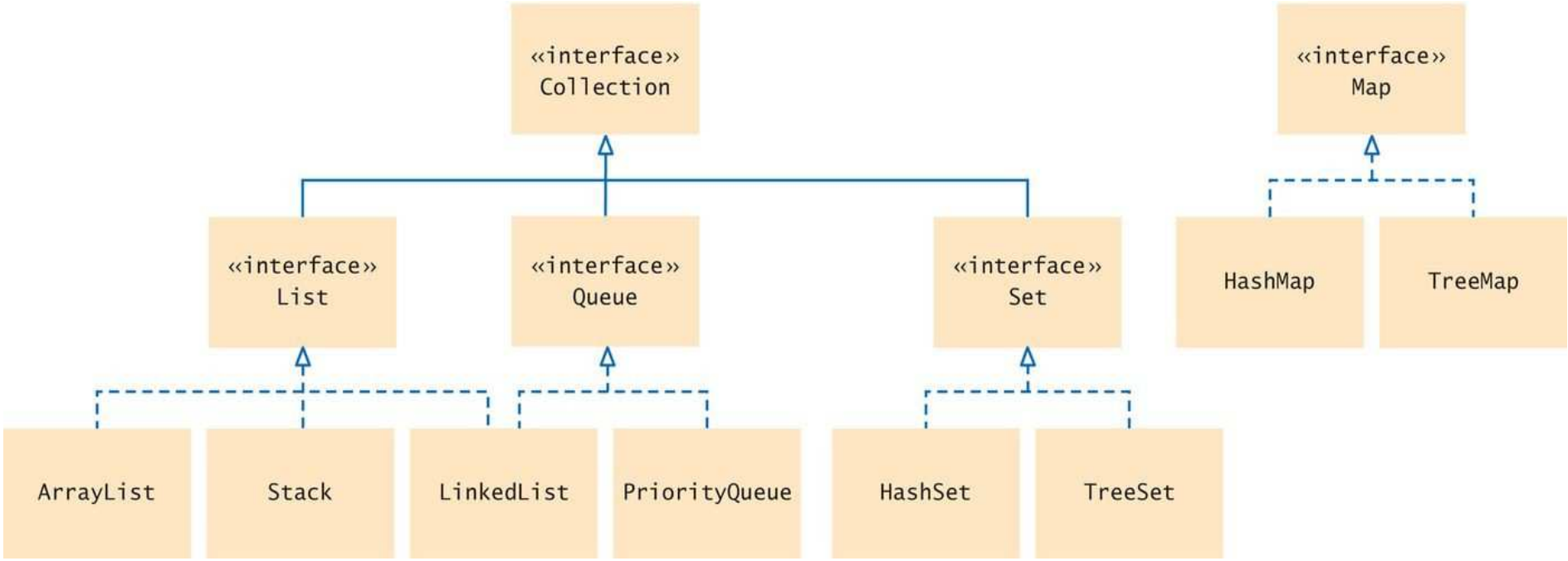


Figure 1 Interfaces and Classes in the Java Collections Framework

Collections Overview

- The Java **Collections** Framework **standardizes** **the way in which groups of objects are handled** by our programs.
- The entire **Collections** Framework is **built upon a set of standard interfaces.**
- Mechanisms were added that allow the **integration of standard arrays** into the **Collections** Framework.

Collections Overview(contd.)

- The **Collections** Framework was designed to meet several goals.
  - First, the framework had to be **high-performance**.
    - The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
  - Second, the framework had to **allow different types of collections to work in a similar manner** and with a **high degree of interoperability**.
  - Third, **extending and/or adapting** a collection had to be **easy**.

Collections Overview(contd.)

- **Algorithms** are an important part of the **collection** mechanism.
  - *Algorithms* operate on collections and are defined as **static methods** within the **Collections class.**
  - The algorithms provide a standard means of **manipulating collections.**
- **Java Collections Framework** provides **algorithm implementations that are commonly used** such as sorting, searching etc.
  - void sort(List list)
  - int binarySearch(List list, Object value)

Collections Overview(contd.)

- Another item closely associated with the Collections Framework is the **Iterator interface.**
  - An iterator offers a general-purpose, **standardized way of accessing the elements** within a collection, **one at a time**.
  - An iterator provides a means of ***enumerating the contents*** *of a collection.*
  - Because each collection implements **Iterator,** the elements of any collection class can be accessed through the methods defined by **Iterator**

Collections Overview(contd.)

- The framework defines several **map** interfaces and classes.
  - *Maps* store key/value pairs.
    - A **map** cannot contain duplicate keys.
    - Although maps are part of the Collections Framework, they are not "collections" in the strict use of the term

Recent Changes to Collections

- Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use.
  - The changes were caused by the addition of
    - **generics**
    - **autoboxing/unboxing**, and
    - **for-each** style **for loop.**

Recent Changes to Collections

- **Generics** add the one feature : **type safety**.
  - With generics, it is possible to **explicitly state the type of data** being stored, and run-time type mismatch errors can be avoided.
- **Autoboxing/unboxing** facilitates the **storing of primitive types** in collections.
  - IN THE PAST, if we wanted ***to store a primitive value, such as an int, in a collection*, we had to manually box it into its type wrapper**.
  - When *the value was* ***retrieved***, it needed to be **manually unboxed** (by using an explicit cast) into its proper primitive type.
  - Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types.

The Collection Framework

- The Collections Framework defines several interfaces.

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

Collection interface

- Collection interface helps to work with group of objects
- The **Collection interface** is at the **top** of collections hierarchy.
- **Collection interface** is the foundation upon which the Collections Framework is built
  - because it must be implemented by any class that defines a collection.
- **Collection** is a generic interface that has this declaration:

```
interface Collection<E>
```

  - Here, **E** specifies the **type of objects** that the collection will hold.
- Collection **extends** the **Iterable interface.**
  - This means that all collections can be cycled through by use of the for-each style **for loop.**

Collection interface(contd.)

Collection declares the **core methods** that all collections will have.

Several of these methods can throw an **UnsupportedOperationException** if a collection cannot be modified.

A **ClassCastException** is generated when one object is **incompatible** with another.

A **NullPointerException** is thrown if an attempt is made to **store a null object** and null elements are not allowed in the collection.

An **IllegalArgumentException** is thrown if an **invalid argument** is used.

An **IllegalStateException** is thrown if an attempt is made to **add an element to a fixed-length collection that is full**.

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

The Methods Defined by Collection

Collection interface(contd.)

- Objects are added to a collection by calling **add( ).**
  - **add( )** takes an argument of type **E,** which means that objects added to a collection must be compatible with the type of data expected by the collection.
- To add the entire contents of one collection to another by calling **addAll( ).**
- To remove an object call **remove( ).**
- To remove a group of objects, call **removeAll( ).**
- To remove all elements except those of a specified group by call **retainAll( ).**
- To empty a collection, call **clear( ).**

Collection interface(contd.)

- We can check whether a collection contains a specific object by calling **contains( ).**
- To check whether one collection contains all the members of another, call **containsAll( ).**
- To determine whether a collection is empty call **isEmpty().**
- The number of elements currently held in a collection can be determined by calling **size( ).**
- The **toArray**( ) methods return an array that contains the elements stored in the invoking collection.
  - **Object[ ] toArray**( ) returns an array of **Object.**
  - **<T> T[ ] toArray(T array[ ])** returns an array of elements that have the same type as the array specified as a parameter.

Collection interface(contd.)

- Two collections can be compared whether they are equal or not by calling **equals( ).**
- The precise meaning of "equality" may differ from collection to collection.
  - **equals**( ) can be implemented to ***compare the values of elements*** stored in the collection.
  - **equals**( ) can be implemented to ***compare references*** to those elements.
- The method **iterator**( ) returns an iterator to a collection.
  - Iterators help to loop through the collections.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 4)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ❑ **Collections framework**
    - ❑ List Interface
    - ❑ Collections Class
    - ❑ ArrayList Class

List Interface

- The **List** interface **extends** *Collection* interface.
- **List** declares the behavior of a collection that stores a sequence of elements.
  - In Java, the **List** interface is an **ordered collection** that allows us to **store and access elements sequentially.**
- Elements can be inserted or accessed by **their position** in the list, using zero-based index.
- A list may contain **duplicate elements**.
- **List is a generic interface** that has this declaration:

```
interface List<E>
```

List Interface(contd.)

- List supports methods defined by **Collection**,
- List defines its own methods also.
- Some methods throw exceptions.
- **Exceptions** that are thrown by List methods are:

UnsupportedOperationException	• if the list cannot be modified
ClassCastException	• when one object is incompatible with another
IndexOutOfBoundsException	• if an invalid index is used
NullPointerException	• thrown if an attempt is made to store a null object and null elements are not allowed in the list.
IllegalArgumentException	• if an invalid argument is used.

Methods in List interface

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Methods in List interface (contd.)

- List has many methods:-
- **add(int, E)** and **addAll(int, Collection)**
    - These methods **insert elements at the specified index.**
- The meaning of **add(E)** and **addAll(Collection)** defined by Collection are changed by List. In **List they add elements to the end of the list.**
- To **obtain the object stored at a specific location**, call **get( )** with the index of the object.
- To **assign a value to an element in the list**, call **set( )**, specifying the index of the object to be changed.
- To **find the index of an object**, use **indexOf( )** or **lastIndexOf()**.
- A **sublist of a list can be obtained** by calling **subList() ,** specifying the beginning and ending indexes of the sublist.

The Collection Classes

- The collection classes **implement collection interfaces**.
- Some of the collection classes provide **full implementations** that can be used as-is.
- Some of the collection classes are **abstract**, providing **skeletal implementations** that are used as starting points for creating concrete collections.
- Collection classes are **not synchronized.**
  - Two or more threads can access the methods of collection class at any time.

The standard collection classes are

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. (Added by Java SE 6.)
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

ArrayList Class

- The **ArrayList class extends AbstractList** and implements the **List** interface.
- ArrayList is a generic class that has declaration:

```
class ArrayList<E>
```

  - Here, E specifies the type of objects that the list will hold.
- **ArrayList** supports **dynamic arrays** that can grow as needed.
  - This is needed because in some cases we may not know how large an array we need precisely until run time.

ArrayList Class(contd.)

- An **ArrayList** is a **variable-length array** of object references.
  - So **ArrayList** can dynamically increase or decrease in size.
- Array lists are created with an initial size.
  - When this size is exceeded, the collection is **automatically enlarged.**
  - When objects are removed, the array can be **shrunk**.

ArrayList Class(contd.)

- **ArrayList** has following **constructors**:

ArrayList( )

- This constructor builds an **empty** array list.

ArrayList(Collection<? extends E> *c)*

- This constructor builds an array list that is **initialized with the elements of the collection c**.

ArrayList(int *capacity)*

- This constructor builds an array list that has the specified initial capacity.
- The capacity is the **size** of the underlying array that is used to store the elements.
- The capacity **grows automatically** as elements are added to an array list.

ArrayList Class(contd.)


```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size=" +al.size());
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size now=" +al.size());
System.out.println("Contents : " + al);
al.remove("F");
al.remove(2);
System.out.println("Size=" +al.size());
System.out.println("Contents=" +al);
}
}
```

```
Initial size=0
Size now=7
Contents : [C, A2, A, E, B, D, F]
Size=5
Contents=[C, A2, E, B, D]
```

ArrayList Class(contd.)

- The contents of a collection are displayed using the default conversion provided by **toString( )**, which was inherited from **AbstractCollection**.
- We can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( ).**

void **ensureCapacity**(int *cap*)

- If we want to **reduce** the size of the array that of**ArrayL ist** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**:

void **trimToSize( )**

Obtaining an Array from an ArrayList

- To **convert a collection into an array**, **toArray( ),** which is defined by **Collection** can be called.
  - This is needed
    - To obtain **faster processing times** for certain operations
    - To **pass an array to a method** that is not overloaded to accept a collection
    - To **integrate collection-based code with legacy code** that does not understand collections
- Two versions of **toArray( )** are:

  Object[ ] toArray( )

  <T> T[ ] toArray(T *array[ ])*

ArrayList Class(contd.)


```
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
ArrayList<Integer> al = new ArrayList<Integer>();

al.add(1);
al.add(2);
al.add(3);
al.add(4);

System.out.println("Contents of al: " + al);
Integer arr[] = new Integer[al.size()];
arr = al.toArray(arr);
int sum = 0;
for(int i : arr) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 5)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ❑ **Collections framework**
    - ❑ Accessing Collections via an Iterator

Accessing Collections via an Iterator

- To **cycle through the elements in a collection** (e.g. display each element, sum of elements etc.), we can use *iterator, which is an object* that **implements** either
  - **Iterator** or
  - **ListIterator**

Accessing Collections via an Iterator(contd.)

- **Iterator** enables you to
  - cycle through a collection
  - obtaining or removing elements.
- **ListIterator extends Iterator** to allow
  - **bidirectional traversal** of a list,
  - the **modification** of elements

Accessing Collections via an Iterator(contd.)

- **Iterator** and **ListIterator** are **generic interfaces** which are declared as :

```
interface Iterator<E>
```

```
interface ListIterator<E>
```

  - Here, **E** specifies the type of objects being iterated

Accessing Collections via an Iterator(contd.)

The Methods Defined by **Iterator**

boolean **hasNext( )**

- Returns **true** if there are **more elements**. Otherwise, returns false.

E **next( )**

- Returns the **next element**.
- Throws **NoSuchElementException** if there is not a next element.

void **remove( )**

- Removes the **current element**.
- **Throws IllegalStateException** if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Accessing Collections via an Iterator(contd.)

Method Defined by **ListIterator**	
void add(E obj)	• **Inserts obj into the list in front** of the element that will be returned by the next call to next( ).
boolean hasNext( )	• Returns **true** if there is a **next element**. Otherwise, returns false.
boolean hasPrevious( )	• Returns **true** if there is **a previous element**. Otherwise, returns false.
E next( )	• **Returns the next element.** NoSuchElementException is thrown if there is not a next element.
int nextIndex( )	• Returns the **index of the next element**. If there is not a next element, returns the size of the list.
E previous( )	• Returns the **previous element**. **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	• Returns the **index of the previous element**. If there is not a previous element, returns −1.
void remove( )	• Removes the **current element** from the list. An **IllegalStateException** is thrown if remove( ) is called before next( ) or previous( ) is invoked.
void set(E obj)	• **Assigns obj to the current element.** This is the element last returned by a call to either next( ) or previous( ).

Exceptions in methods

- Exceptions in the Methods Defined by **Iterator**
  - **NoSuchElementException**
  - **IllegalStateException**
- Exceptions in the Methods Defined by **ListIterator**
  - **NoSuchElementException**
  - **IllegalStateException**
  - **UnsupportedOperationException**

Using an Iterator

- Each of the **collection classes** provides an **iterator( ) method that returns an iterator to the start** of the collection.
  - By using this iterator object, we can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection,
  - 1. **Obtain an iterator** to the start of the collection by calling the collection's **iterator( )** method.
  - 2. Set up a loop that makes a call to **hasNext( ).**
    - Have the loop iterate as long as hasNext( ) returns true.
  - 3. Within the loop**, obtain each element** by calling **next( ).**

```
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext())
{
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
ListIterator<String> litr = al.listIterator();
while(litr.hasNext())
 { String element = litr.next();
litr.set(element + "+");  }
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " "); }
System.out.println();
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
} System.out.println();  }   }
```

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

The For-Each Alternative to Iterators

- The **for** loop is substantially **shorter** and **simpler** to use than the iterator based approach.
- Bur **for** loop can only be used to cycle through a collection in the forward direction, and we **can't modify** the contents of the collection.
- If we don't want to modify the contents of a collection or obtaining elements in reverse order, then the **for-each version** of the for loop is often a more **convenient alternative** to cycling through a collection than is using an iterator.

for each loop used to find sum of elements in collection


```
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
ArrayList<Integer> vals = new ArrayList<Integer>();
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
System.out.print("Original contents of vals: ");
for(int v : vals)
    System.out.print(v + " ");
System.out.println();

int sum = 0;
for(int v : vals)
    sum += v;
System.out.println("Sum of values: " + sum);
}
}
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 6)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Event handling:**
  - ❑ **Event Handling Mechanisms**
  - ❑ **Delegation Event Model**

Event Handling

- There are several types of events,
  - Events can be generated by
    - the mouse (click, move, drag mouse etc.)
    - the keyboard (type, press, release etc.)
    - different GUI controls, such as a
      - push button
      - scroll bar
      - check box
- **Event Handling** is the mechanism that **controls the event** and **decides what should happen if an event occurs**.

Event Handling(contd.)

- Events are supported by a number of packages, including

  java.util

  java.awt

  java.awt.event

- Event handling is an integral part in the **creation** of **applets** and other types of **GUI-based programs**.
- Applets are **event-driven programs** that use a graphical user interface(GUI) to interact with the user.
- Any program that uses a graphical user interface is event driven.
  - Thus, we cannot write these types of programs without a solid command of event handling.

Event Handling Mechanisms

- The **two ways** in which events are handled changed significantly between the (Two event handling mechanisms)
  - **original version of Java (1.0) event handling** and
  - **modern versions of Java (beginning with version 1.1) event handling**
- The 1.0 method of event handling is still supported, but it is not recommended for new programs.
  - Many of the methods that support the old 1.0 event model have been deprecated.
- The **modern approach** is the way that **events should be handled by all new programs**

The Delegation Event Model

- The **modern approach** to handling events is based on the ***delegation event model***
  - It defines standard and consistent mechanisms to **generate and process events.**
- Concept of ***delegation event model*** **:**
  - A **source** **generates an event** and **sends** it to one or more **listeners**.
  - In this scheme, the listener simply **waits until it receives an event**.
  - Once an event is received, the listener processes the event and then returns.

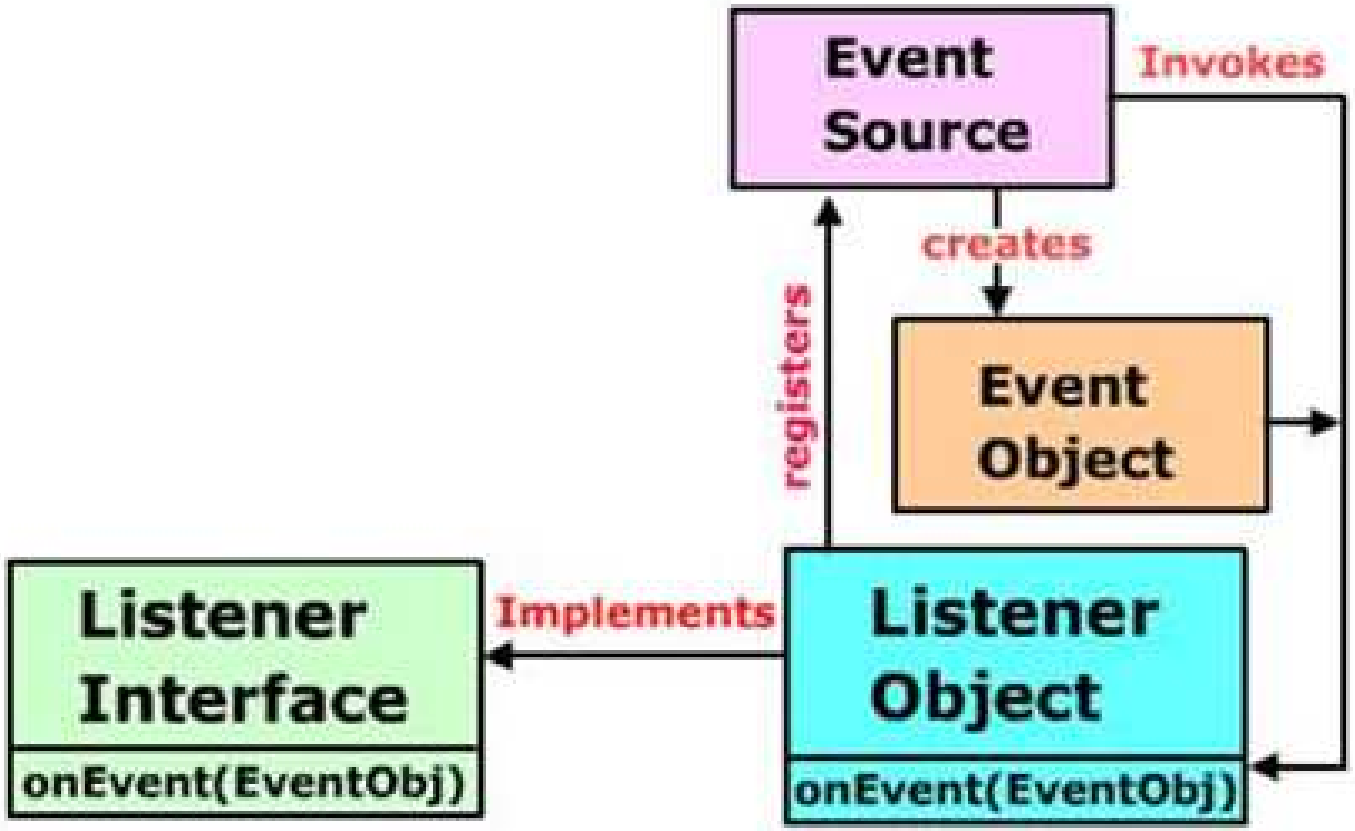
The Delegation Event Model(contd.)

- The **advantage** of **Delegation Event Model** is that the **application logic** that **processes events** is cleanly separated from the **user interface logic** that **generates those events.**
- A user interface element is able **to "delegate"(entrust) the processing of an event to a separate piece of code.**
- In the delegation event model, **listeners must register with a source in order to receive an event notification**.
  - **Benefit:** Notifications are sent only to listeners that want to receive them.

The Delegation Event Model(contd.)

- In previous models, an event was *propagated up the containment hierarchy* **until it was handled** by a component.
  - This required components to receive events that they did not process, and it wasted valuable time.
- **The delegation event model eliminates this overhead**

The Delegation Event Model(contd.)



The Delegation Event Model -Event

- In the delegation model, an **event** is an **object** that describes a **state change in a source.**
- Events can be caused with or without user interaction.

The Delegation Event Model -Event

- Some events are caused by interactions with a user interface such as:
  - pressing a button,
  - entering a character via the keyboard,
  - selecting an item in a list,
  - clicking the mouse.

The Delegation Event Model –Event (contd.)

- Events may also occur that are not directly caused by interactions with a user interface.
  - Example: an **event** may be generated
    - when a timer expires,
    - a counter exceeds a value,
    - a software or hardware failure occurs,
    - an operation is completed

The Delegation Event Model – Event Sources

- A **event source** is an **object** that **generates an event.**
  - Event occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- A **source** must **register listeners,** then only listeners can receive notifications about a specific type of event.
- Each type of event has its own registration method.

The Delegation Event Model – Event Sources (contd.)

General form of listener registration is:

public void ***add****TypeListener*(*TypeListener* el)

- Type is the name of the event, and el is a reference to the event listener.
- For example, the method that **registers a keyboard event listener** is called **addKeyListener( ).**
- The method that **registers a mouse motion listener** is called **addMouseMotionListener( ).**

The Delegation Event Model – Event Sources (contd.)

- When an event occurs, **all registered listeners are notified** and receive a copy of the event object. This is known as ***multicasting*** the event.
  - In all cases, notifications are sent only to listeners that register to receive them.
- Some sources may allow **only one listener to register.** The general form of such a method is this:

```
public void addTypeListener(TypeListener el) throws
java.util.TooManyListenersException
```

  - When such an event occurs, that single registered listener is notified. This is known as **unicasting** *the event.*

The Delegation Event Model – Event Sources (contd.)

- A source must also provide a method that allows a **listener to unregister** an interest in a specific type of event. The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

  - Here, *Type is the name of the event, and el is a reference to the event listener.*

  – *For example,* to remove a keyboard listener, call **removeKeyListener( ).**

- The **methods that add or remove listeners** are provided by the **source** that generates events.

  – For example, the **Component class** provides methods to add and remove keyboard and mouse event listeners.

The Delegation Event Model – Event Listeners

- A **listener** is an object that is **notified when an event occurs.**
- It has two major requirements.
  - First, it must have been **registered with one or more sources** to receive notifications about specific types of events.
  - Second, it **must implement methods** to receive and process these notifications.

The Delegation Event Model – Event Listeners (contd.)

- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event.**
  - For example, the **MouseMotionListener interface** defines two methods to **receive notifications when the mouse is dragged or moved.**
    - Any object may receive and process one or both of these events if it provides an implementation of this interface.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 7)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

☑ Event handling:

- ❑ Event Classes

Event Classes

- The classes that represent events(Event classes) are at the core of Java's event handling mechanism.
- The most widely used events are those defined by the AWT and those defined by Swing.
- At the **root of the Java event class hierarchy** is **EventObject,** which is in **java.util.**
  - **EventObject is the** superclass for all events.
  - Its one constructor is :

    EventObject(Object *src)*
  - Here, *src is the object that generates this event.*

Event Classes(contd.)

- **EventObject** contains two methods:

  getSource( )

  toString( ).

- The getSource( ) method returns the source of the event.

  Its general form is :

  Object **getSource( )**

- **toString( )** returns the string equivalent of the event.

Event Classes(contd.)

- The class **AWTEvent**, defined within the **java.awt** package, is a subclass of EventObject.
  - It is the **superclass** (either directly or indirectly) of all AWT-based events used by the delegation event model.
  - Its **getID( )** method can be used to determine the type of the event.
  - The signature of **getID( )** method is

```
int getID( )
```

Event Classes(contd.)

- **EventObject is a superclass of all events.**
- **AWTEvent is a superclass of all AWT events that are handled by the delegation event model.**

Event Class (contd.)

ActionEvent	• Generated when a **button is pressed**, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	• Generated when a scroll bar is manipulated.
ComponentEvent	• Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	• Generated when a component is added to or removed from a container.
FocusEvent	• Generated when a component gains or loses keyboard focus.
InputEvent	• Abstract superclass for all component input event classes.
ItemEvent	• Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	• Generated when input is received from the keyboard.
MouseEvent	• Generated when the mouse is dragged, moved, clicked, pressed, or released;also generated when the mouse enters or exits a component.
MouseWheelEvent	• Generated when the mouse wheel is moved.
TextEvent	• Generated when the value of a text area or text field is changed.
WindowEvent	• Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event classes

ActionEvent	• Generated when a **button is pressed**, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	• Generated when a **scroll bar is manipulated**.
ComponentEvent	• Generated when a **component** is **hidden**, **moved**, **resized**, or becomes **visible**.
ContainerEvent	• Generated when a **component** is **added** to or **removed from** a **container**.
FocusEvent	• Generated when a **component gains or loses keyboard focus.**
InputEvent	• Abstract **superclass** for all component input event classes.

Event classes(contd.)

ItemEvent	• Generated when a **check box or list item is clicked;** also occurs when a **choice selection is made** or a **checkable menu item is selected or deselected**.
KeyEvent	• Generated when **input** is received from the **keyboard**.
MouseEvent	• Generated when the mouse is **dragged, moved, clicked, pressed, or released**;also generated when the mouse enters or exits a component.
MouseWheelEvent	• Generated when the **mouse wheel is moved**.
TextEvent	• Generated when the value of a **text area or text field is changed.**
WindowEvent	• Generated when a **window is activated, closed, deactivated, deiconified, iconified, opened, or quit**.

The ActionEvent Class

- An **ActionEvent** is generated when
  - a button is pressed,
  - a list item is double-clicked,
  - menu item is selected.
- The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event:
  - **ALT_MASK**
  - **CTRL_MASK**
  - **META_MASK**
  - **SHIFT_MASK**.
- Integer constant **ACTION_ PERFORMED,** can be used to identify action events.

The ActionEvent Class(contd.)

- **ActionEvent has these three constructors:**

ActionEvent(Object *src, int type, String cmd)*

ActionEvent(Object *src, int type, String cmd, int modifiers)*

ActionEvent(Object *src, int type, String cmd, long when,*

int modifiers)

  - Here, *src* is a reference to the object that generated this event.
  - The type of the event is specified by *type*, and its command string is *cmd*.
  - The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.
  - The *when* parameter specifies when the event occurred.

The ActionEvent Class(contd.)

- To obtain the **command name** for the invoking **ActionEvent** object **getActionCommand( )** method can used**:**

String **getActionCommand( )**

- For example, when a button is pressed, an action event is generated that has a command name equal to the **label on that button**.
- E.g.

The command name of button is **Submit.**

The ActionEvent Class(contd.)

- The **getModifiers( )** method
  - returns a value that indicates **which modifier keys** (ALT, CTRL, META, and/or SHIFT) **were pressed** when the event was generated. Its form is :

```
int getModifiers( )
```

- The method **getWhen( )**
  - returns the time at which the event took place. This is called the **event's *timestamp***. *The getWhen( ) method is :*

```
long getWhen( )
```

The AdjustmentEvent Class

- An AdjustmentEvent is generated by a **scroll bar**.
- There are five types of adjustment events.
- The AdjustmentEvent class defines **integer constants** that can be used to identify them.

BLOCK_DECREMENT	• The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	• The user clicked inside the scroll bar to increase its value.
TRACK	• The slider was dragged.
UNIT_DECREMENT	• The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	• The button at the end of the scroll bar was clicked to increase its value

The AdjustmentEvent Class(contd.)

- An integer constant, **ADJUSTMENT_VALUE_CHANGED,** that indicates that a change has occurred.
- One **AdjustmentEvent constructor:**

AdjustmentEvent(Adjustable *src, int id, int type, int data)*

  - Here, *src* is a reference to the object that generated this event.
  - The *id* specifies the event.
  - The type of the adjustment is specified by *type*, and its associated data is *data*.

The AdjustmentEvent Class(contd.)

- The **getAdjustable( )** method returns the object that generated the event. Its form is**;**

```
Adjustable getAdjustable( )
```

- The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent.** The general form is :

```
int getAdjustmentType( )
```

- The amount of the adjustment can be obtained from the **getValue( )** method is**:**

```
int getValue( )
```

  - For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

- A ComponentEvent is generated when the size, position, or visibility of a component is changed.
- There are four types of component events.
  - The ComponentEvent class defines integer constants for this.

COMPONENT_HIDDEN	• The component was hidden.
COMPONENT_MOVED	• The component was moved.
COMPONENT_RESIZED	• The component was resized.
COMPONENT_SHOWN	• The component became visible.

The ComponentEvent Class(contd.)

- **ComponentEvent has the constructor:**

```
ComponentEvent(Component src, int type)
```

  - Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

- **ComponentEvent** is the **superclass** either directly or indirectly of *ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.*

- The **getComponent( )** method returns the component that generated the event

```
Component getComponent( )
```

The ContainerEvent Class

- A ContainerEvent is generated when a component is added to or removed from a container.
- There are **two types of container events**.
- The ContainerEvent class defines int constants that can be used to identify them:
  - COMPONENT_ADDED
  - COMPONENT_REMOVED.

The ContainerEvent Class(contd.)

- **ContainerEvent** is a **subclass of ComponentEvent.**
- Constructor:

ContainerEvent(Component *src, int type, Component comp)*

  - Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.

The ContainerEvent Class(contd.)

- A reference to the container that generated this event by using the **getContainer( ) method.**

```
Container getContainer( )
```

- The **getChild( )** method returns a reference to the **component** that was added to or removed from the container.

```
Component getChild( )
```

The FocusEvent Class

- A FocusEvent is generated when a component gains or loses input focus.
- These events are identified by the integer constants
  - FOCUS_GAINED
  - FOCUS_LOST.
- **FocusEvent** is a **subclass** of **ComponentEvent** and has these constructors:

FocusEvent(Component *src, int type)*
FocusEvent(Component *src, int type, boolean temporaryFlag)*
FocusEvent(Component *src, int type, boolean temporaryFlag, Component other)*

The argument *temporaryFlag is set to* ***true if the focus event is temporary.***
Otherwise, it is set to **false.**
The other component involved in the focus change, called the ***opposite component****, is passed* in *other.*

The FocusEvent Class(contd.)

- A temporary focus event occurs as a result of another user interface operation.
  - For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.
- if a FOCUS_GAINED event occurred, *other* will refer to the component that lost focus.
- Conversely, if a FOCUS_LOST event occurred, *other* will refer to the component that gains focus.

The FocusEvent Class(contd.)

- To determine the other component call **getOppositeComponent( ):**

```
Component getOppositeComponent( )
```

  - The opposite component is returned.

- The **isTemporary( )** method indicates if this focus change is temporary.

```
boolean isTemporary( )
```

  - The method returns **true** **if the change is temporary.**
  - Otherwise, it returns **false.**

The InputEvent Class

- The **abstract class InputEvent** is a **subclass** of **ComponentEvent** and is the **superclass** for component input events.
  - Its **subclasses** are **KeyEvent** and **MouseEvent.**
- **InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed.
- **InputEvent** class defined the following eight values to **represent the modifiers**:

- ALT_MASK
- ALT_GRAPH_MASK
- BUTTON1_MASK
- BUTTON2_MASK
- BUTTON3_MASK
- CTRL_MASK
- META_MASK
- SHIFT_MASK

The InputEvent Class(contd.)

- The **extended modifier** values to avoid conflict between keybaord and mouse event modifiers are:

•ALT_DOWN_MASK

•ALT_GRAPH_DOWN_MASK

•BUTTON1_DOWN_MASK

•BUTTON2_DOWN_MASK

•BUTTON3_DOWN_MASK

•CTRL_MASK

•META_DOWN_MASK

•SHIFT_DOWN_MASK

The InputEvent Class(contd.)

- To test if a modifier was pressed at the time an event is generated, use the **isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ),** and **isShiftDown( ) methods.**

boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )

The InputEvent Class(contd.)

- To obtain a value that contains all of the original modifier flags call **getModifiers( ) method**

```
int getModifiers( )
```

- We can obtain the extended modifiers by calling **getModifiersEx( ), which is shown here:**

```
int getModifiersEx( )
```

The ItemEvent Class

- An **ItemEvent** is generated when
  - a **check box or a list item is clicked** or
  - when a **checkable menu item is selected or deselected.**
- There are two types of **item events**, which are identified by the following integer constants:

  DESELECTED The user deselected an item.

  SELECTED The user selected an item.

The ItemEvent Class(contd.)

- **ItemEvent** defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.
- ItemEvent has this constructor:

ItemEvent(ItemSelectable *src, int type, Object entry, int state)*

  - Here, *src* is a reference to the component that generated this event.
    - For example, this might be a list or choice element.
  - The type of the event is specified by *type*.
  - The specific item that generated the item event is passed in *entry*.
  - The current state of that item is in *state*

The ItemEvent Class(contd.)

- The getItem( ) method can be used to obtain a reference to the item that generated an event.

```
Object getItem( )
```

- The getItemSelectable( ) method can be used to obtain a reference to the ItemSelectable object that generated an event.

```
ItemSelectable getItemSelectable( )
```

  - Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

- The getStateChange( ) method returns the state change (that is, SELECTED or DESELECTED) for the event.

```
int getStateChange( )
```

The KeyEvent Class

- A **KeyEvent** is generated when keyboard input occurs.
- There are three types of key events, which are identified by these integer constants:

  KEY_PRESSED

  KEY_RELEASED

  KEY_TYPED.

  - The first two events are generated when any key is pressed or released.
  - The last event occurs only when a character is generated.
- Some keypresses does not result in characters.
  - For example, pressing SHIFT does not generate a character.

The KeyEvent Class(contd.)

- There are many other integer constants that are defined by **KeyEvent.**
  - For example, VK_0 through VK_9 define the ASCII equivalents of the numbers.
  - VK_A through VK_Z define the ASCII equivalents of the letters.

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- The VK constants specify **virtual key codes** and are independent of any modifiers, such as control, shift, or alt.

The KeyEvent Class(contd.)

- **KeyEvent** is a **subclass** of **InputEvent.**
- **Constructor:**

KeyEvent(Component *src, int type, long when,*
int modifiers, int code, char ch)

- Here, *src* is a reference to the component that generated the event.
- The type of the event is specified by *type*.
- The system time at which the key was pressed is passed in *when*.
- The *modifiers* argument indicates which modifiers were pressed when this key event occurred.
- The virtual key code, such as **VK_UP, VK_A**, and so forth, is passed in code.
- The character equivalent (if one exists) is passed in *ch*.
  - If no valid character exists, then ch contains CHAR_UNDEFINED.
  - For KEY_TYPED events, code will contain VK_UNDEFINED.

The KeyEvent Class(contd.)

- The KeyEvent class defines several methods,
  - **getKeyChar( ),** which returns the character that was entered,
  - **getKeyCode( ),** which returns the key code.

char **getKeyChar( )**
int **getKeyCode( )**

  - If no valid character is available, then getKeyChar( ) returns CHAR_UNDEFINED.
  - When a KEY_TYPED event occurs, getKeyCode( ) returns VK_UNDEFINED.

The MouseEvent Class

- There are eight types of mouse events.
- The **MouseEvent class** defines the following integer constants that can be used to identify them:
  - ✓ MOUSE_CLICKED The user clicked the mouse.
  - ✓ MOUSE_DRAGGED The user dragged the mouse.
  - ✓ MOUSE_ENTERED The mouse entered a component.
  - ✓ MOUSE_EXITED The mouse exited from a component.
  - ✓ MOUSE_MOVED The mouse moved.
  - ✓ MOUSE_PRESSED The mouse was pressed.
  - ✓ MOUSE_RELEASED The mouse was released.
  - ✓ MOUSE_WHEEL The mouse wheel was moved.

The MouseEvent Class(contd.)

- **MouseEvent** is a s**ubclass of InputEvent.** ***Constructor:***

MouseEvent(Component *src, int type, long when, int modifiers,* int *x, int y, int clicks, boolean triggersPopup)*

- Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*.
- The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred.
- The coordinates of the mouse are passed in *x* and *y*.
- The click count is passed in *clicks*.
- The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

The MouseEvent Class(contd.)

- Two commonly used methods in this class are
  - **getX( ) -**return the X coordinate
  - **getY( ) -** return the Y coordinate
    - (within the component when the event occurred.)

int **getX**( )
int **getY**( )

- **getPoint( ) method -** to obtain the **coordinates** of the mouse.

Point **getPoint**( )

  - returns a Point object that contains the X,Y coordinates in its integer members: x and y.

The MouseEvent Class(contd.)

- The **translatePoint( )** method **changes the location of the event**.

```
void translatePoint(int x, int y)
```

  - Here, the arguments *x and* y are ***added to the coordinates*** of the event.

- The **getClickCount( )** method obtains the **number of mouse clicks** for this event.

```
int getClickCount( )
```

- The **isPopupTrigger( )** method **tests if this event causes a pop-up menu to appear** on this platform.

```
boolean isPopupTrigger( )
```

The MouseEvent Class(contd.)

- **getButton( )** method- **returns a value that represents the button** that caused the event

```
int getButton( )
```

- The return value will be one of these constants defined by **MouseEvent:**

NOBUTTON - indicates that no button was pressed or released.

BUTTON1

BUTTON2

BUTTON3

The MouseEvent Class(contd.)

- Java SE 6 added three methods to **MouseEvent that obtain the coordinates of the mouse relative to the screen** rather than the component.

Point **getLocationOnScreen**( )

- The **getLocationOnScreen**( ) method returns a Point object that contains both the X and Y coordinate.

int **getXOnScreen( )** –

- return the X coordinate.

int **getYOnScreen( )**

- return the Y coordinate.

The MouseWheelEvent Class

- The **MouseWheelEvent** class encapsulates a mouse wheel event.
  - It is a **subclass of MouseEvent**.
- Not all mice have wheels.
- If a mouse has a wheel, it is located between the left and right buttons.
- Mouse wheels are used for scrolling.
- MouseWheelEvent defines these two integer constants:

WHEEL_BLOCK_SCROLL	A **page-up** or **page-down** scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

The MouseWheelEvent Class(contd.)

- Constructor defined by **MouseWheelEvent:**

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count)*

  - Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
  - The *scrollHow* value must be either **WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL.**
  - The number of units to scroll is passed in *amount*.
  - The *count* parameter indicates the number of rotational units that the wheel moved

The MouseWheelEvent Class(contd.)

- MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call getWheelRotation( ), :

  int **getWheelRotation**( )

  - If the value is **positive**, the wheel moved counterclockwise.
  - If the value is negative, the wheel moved clockwise.
- To obtain the type of scroll, call getScrollType( ), shown next:

  int **getScrollType**( )

  - It returns either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL.
- If the scroll type is WHEEL_UNIT_SCROLL, we can obtain the number of units to scroll by calling getScrollAmount( ).

  int **getScrollAmount**( )

The TextEvent Class

- Instances of TextEvent class describe text events.
- These are generated by **text fields** and **text areas** when ***characters are entered by a user or program***.
- **TextEvent** defines the integer constant

  TEXT_VALUE_CHANGED.
- The one constructor for this class is :

  TextEvent(Object *src, int type)*

  – Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*

The TextEvent Class(contd.)

- The **TextEvent object does not include the characters currently in the text component** that generated the event
  - Our program must use other methods associated with the text component to retrieve that information.
- Text event notification is like as a signal to a listener that it should retrieve information from a specific text component

The WindowEvent Class

- There are ten types of window events.
- The **WindowEvent class defines integer constants** that can be used to
  - **WINDOW_ACTIVATED** The window was activated.
  - **WINDOW_CLOSED** The window has been closed.
  - **WINDOW_CLOSING** The user requested that the window be closed.
  - **WINDOW_DEACTIVATED** The window was deactivated.
  - **WINDOW_DEICONIFIED** The window was deiconified.
  - **WINDOW_GAINED_FOCUS** The window gained input focus.
  - **WINDOW_ICONIFIED** The window was iconified.
  - **WINDOW_LOST_FOCUS** The window lost input focus.
  - **WINDOW_OPENED** The window was opened.
  - **WINDOW_STATE_CHANGED** The state of the window changed.

The WindowEvent Class(contd.)

- **WindowEvent** is a **subclass of ComponentEvent.**
- It defines several constructors.

WindowEvent(Window *src, int type)*

  - Here, *src is a reference to the object that generated this event. The type of the event is type.*

- The next three constructors offer more detailed control:

WindowEvent(Window *src, int type, Window other)*

WindowEvent(Window *src, int type, int fromState, int toState)*

WindowEvent(Window *src, int type, Window other, int fromState, int toState)*

  - Here, *other* specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

The WindowEvent Class(contd.)

- A commonly used method in this class is getWindow( ).
- getWindow( ) returns the Window object that generated the event.

```
Window getWindow( )
```

- WindowEvent also defines methods that
  - return the opposite window (when a focus or activation event has occurred),

```
Window getOppositeWindow( )
```

  - the previous window state,

```
int getOldState( )
```

  - and the current window state.

```
int getNewState( )
```

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 8)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

☑ **Event handling:**

- ❑ Sources of Events
- ❑ Event Listener Interfaces
- ❑ Using the Delegation Model

Sources of Events

- The components in user interface that can generate the events are the **sources of the event.**
- Event Source are:
  - ☑ Button
  - ☑ Check box
  - ☑ Choice
  - ☑ List
  - ☑ Menu Item
  - ☑ Scroll bar
  - ☑ Text components
  - ☑ Window

Event Source & Description

Button	• Generates **action events** when the button is **pressed.**
Check box	• Generates **item events** when the check box **is selected or deselected.**
Choice	• Generates **item events** when the choice is **changed**.
List	• Generates **action events** when an item is **double-clicked**; generates **item events** when an item is **selected or deselected.**
Menu Item	• Generates **action events** when a **menu item is selected**; generates **item events** when a **checkable menu item is selected or deselected.**
Scroll bar	• Generates **adjustment events** when the scroll bar is **manipulated**.
Text components	• Generates **text events** when the user **enters a character.**
Window	• Generates **window events** when a window is **activated, closed, deactivated, deiconified, iconified, opened, or quit.**

Sources of Events(contd.)

- Any class derived from **Component,** such as Applet, can generate events.
  - For example, we can receive key and mouse events from an applet.

Event Listener Interfaces

- Listeners are created by **implementing** one or more of the **interfaces** defined by the **java.awt.event** package,
- When an event occurs,
  - the event source invokes the appropriate method defined by the listener and provides an event object as its argument

Event class	Event Listener interface
ActionEvent	ActionListener
AdjustmentEvent	AdjustmentListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener
InputEvent	
ItemEvent	ItemListener
KeyEvent	KeyListener
MouseEvent	MouseListener
	MouseMotionListener
MouseWheelEvent	MouseWheelListener
TextEvent	TextListener
WindowEvent	WindowFocusListener
	WindowListener

Event listeners & Description

ActionListener	• Defines one method to receive **action events.**
AdjustmentListener	• Defines one method to receive **adjustment events.**
ComponentListener	• Defines four methods to recognize when a component is **hidden, moved, resized, or shown**.
ContainerListener	• Defines two methods to recognize when a **component** is **added to or removed** from a container.
FocusListener	• Defines two methods to recognize when a component **gains or loses keyboard focus**.
ItemListener	• Defines one method to recognize when the **state of an item changes.**

Event listeners & Description(contd.)

KeyListener	• Defines three methods to recognize when a **key is pressed, released, or typed.**
MouseListener	• Defines five methods to recognize when the **mouse is clicked, enters a component, exits a component, is pressed, or is released.**
MouseMotionListener	• Defines two methods to recognize when the mouse is **dragged or moved.**
MouseWheelListener	• Defines one method to recognize when the **mouse wheel is moved.**
TextListener	• Defines one method to recognize when **a text value changes.**
WindowFocusListener	• Defines two methods to recognize when a **window gains or loses input focus**.
WindowListener	• Defines seven methods to recognize when a window is **activated, closed, deactivated, deiconified, iconified, opened, or quit.**

ActionListener Interface

- ActionListener interface defines the **actionPerformed( ) method that is invoked when an action event** occurs.
  - **button is pressed**
  - **a list item is double-clicked**
  - **menu item is selected.**

void **actionPerformed**(**ActionEvent** *ae*)

AdjustmentListener Interface

- The **AdjustmentListener Interface d**efines the **adjustmentValueChanged( )** method that is invoked when an **adjustment event** occurs.
  - **scroll bar is manipulated**.

void **adjustmentValueChanged**(**AdjustmentEvent** *ae)*

ComponentListener Interface

- **The ComponentListener Interface** defines four methods that are invoked when a component is resized, moved, shown, or hidden.

void component**Resized**(**ComponentEvent** *ce)*

void component**Moved**(**ComponentEvent** *ce)*

void component**Shown**(**ComponentEvent** *ce)*

void component**Hidden**(**ComponentEvent***ce)*

ContainerListener Interface

- The **ContainerListener Interface** contains two methods.
- When a component is added to a container,
  - **componentAdded( ) is invoked.**
- When a component is removed from a container,
  - **componentRemoved( ) is invoked.**

void componentAdded(ContainerEvent *ce)*

void componentRemoved(ContainerEvent *ce)*

FocusListener Interface

- **The FocusListener Interface** defines two methods.
- When a component obtains keyboard focus,
  - **focusGained( )** is invoked.
- When a component loses keyboard focus,
  - **focusLost( )** is called.

void focus**Gained**(**FocusEvent** *fe*)

void focus**Lost**(**FocusEvent** *fe*)

ItemListener Interface

- The **ItemListener Interface** defines the **itemStateChanged( )** method that is invoked when the state of an item changes.

```
void itemStateChanged(ItemEvent ie)
```

KeyListener Interface

- The **KeyListener Interface** defines three methods.
- When a key is pressed , **keyPressed( )** method is invoked
- When a key is released, **keyReleased( )** method is invoked
- When a character has been entered **keyTyped( )** method is invoked
- For example, if a user presses and releases the letter **A** key in keyboard, three events are generated in sequence:
  - key pressed
  - key typed,
  - key released.
- If a user presses and releases the **HOME** key in keyboard, two key events are generated in sequence:
  - key pressed
  - key released

KeyListener Interface(contd.)

- The general forms of methods are :

void key**Pressed**(**KeyEvent** *ke*)
void key**Released**(**KeyEvent** *ke*)
void key**Typed**(**KeyEvent** *ke*)

MouseListener Interface

- **The MouseListener Interface** defines five methods.
  - If the mouse is pressed and released at the same point, **mouseClicked( ) is invoked.**
  - When the mouse enters a component, the **mouseEntered() method is called.**
  - When it leaves, **mouseExited( ) is called.**
  - When the mouse is pressed **the mousePressed( ) method is invoked**
  - When the mouse is released, **mouseReleased( ) methods is invoked**

void **mouseClicked**(**MouseEvent** *me)*

void **mouseEntered**(**MouseEvent** *me)*

void **mouseExited**(**MouseEvent** *me)*

void **mousePressed**(**MouseEvent** *me)*

void **mouseReleased**(**MouseEvent** *me)*

MouseMotionListener Interface

- The **MouseMotionListener Interface** defines two methods.

```
void mouseDragged(MouseEvent me)
```

```
void mouseMoved(MouseEvent me)
```

mouseDragged( )	• **called multiple times** as the mouse is **dragged**.
mouseMoved( )	• **called multiple times as the mouse** is **moved**.

MouseWheelListener Interface

- The **MouseWheelListener Interface** defines the **mouseWheelMoved( ) method** that is invoked **when the mouse** wheel is **moved**.

void mouse**WheelMoved**(**MouseWheelEvent** *mwe)*

TextListener Interface

- The **TextListener Interface** defines the **textChanged( )** method that is invoked **when a change occurs** in a **text area** or **text field.**

void **textChanged**(**TextEvent** *te*)

WindowFocusListener Interface

- The **WindowFocusListener Interface** defines two methods:
  - **windowGainedFocus( ) -** called when a window gains input focus
  - **windowLostFocus( ) -** called when a window loses input focus

void window**GainedFocus**(**WindowEvent** *we)*

void window**LostFocus**(**WindowEvent** *we)*

WindowListener Interface

- **The WindowListener Interface** defines seven methods.

void window**Activated**(**WindowEvent** *we)*

void window**Closed**(**WindowEvent** *we)*

void window**Closing**(**WindowEvent***we)*

void window**Deactivated**(**WindowEvent** *we)*

void window**Deiconified**(**WindowEvent** *we)*

void window**Iconified**(**WindowEvent** *we)*

void window**Opened**(**WindowEvent** *we)*

WindowListener Interface(contd.)

windowActivated( )	• invoked when a window is **activated**
windowDeactivated( )	• invoked when a window is **deactivated**
windowIconified( )	• invoked if a window is **iconified**,
windowDeiconified( )	• Invoked when a window is **deiconified**
windowOpened( )	• When a window is **opened**
windowClosed( )	• Invoked when a window is **closed**
windowClosing( )	• Invoked when a window is **being closed**.

Simple applet program

```
import java.awt.*;
import java.applet.*;
/*
<applet  code="Sampleapplet"   width=300  height=50>
</applet>
*/
public class Sampleapplet extends Applet{

String msg;
public void init()
 {
setBackground(Color.cyan);
setForeground(Color.red);

}
public void paint(Graphics g) {
msg ="Welcome to first applet";
g.drawString(msg, 10, 30);
}
}
```

```
//TO COMPILE....
 javac Sampletest.java
//TO RUN.......
appletviewer Sampletest.java
```

Using the Delegation Event Model

- To use the delegation event model follow two steps:

1. **Implement the appropriate interface in the listener** so that it will receive the type of event desired.
2. **Implement code to register and unregister** (if necessary) **the listener** as a recipient for the event notifications.

Using the Delegation Event Model(contd.)

- A source may generate several types of events.
  - Each event must be registered separately.
- An object may register to receive several types of events,
  - but it must implement all of the interfaces that are required to receive these events

Handling Mouse Events

- To handle **mouse events**, we must **implement** the **MouseListener** and the **MouseMotionListener** interfaces.
  - We can implement **MouseWheelListener**, also.

Handling Mouse Events-Applet program example.

- Design an applet program with following features
  - It **displays the current coordinates of the mouse** in the applet's ***status window.***
  - Each time a **button** is **pressed**, the word "Down" is displayed at the location of the ***mouse pointer.***
  - Each time the **button is released**, the word "Up" is shown.
  - If a **button** is **clicked**, the message "Mouse clicked" is displayed in the upperleft corner of the ***applet display area***.
  - As the mouse **enters or exits** the applet window, a message is displayed in the upper-leftcorner of the ***applet display area***.
  - When **dragging the mouse,** a * is shown, which tracks with the mouse pointer as it is dragged.
  - **mouseX and mouseY** coordinates are then used by **paint( ) to** display output at the point of these occurrences.

Handling Mouse Events-Applet program example.

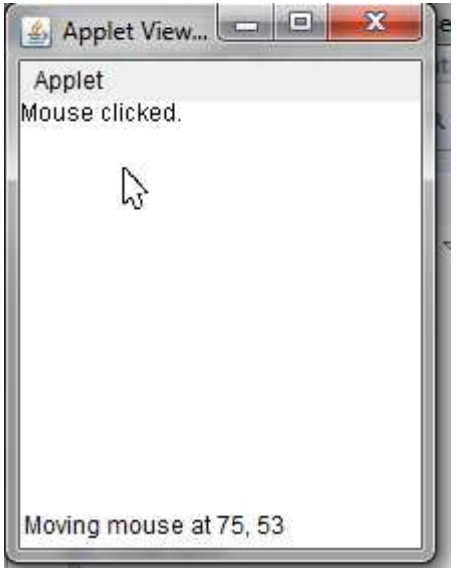
```
import java.awt.event.*;
import java.applet.*;
import java.awt.*;
/*
<applet code="MouseEvents" width=500 height=500>
</applet>
*/
public class MouseEvents extends Applet implements
    MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0;
public void init()
 {
addMouseListener(this);
addMouseMotionListener(this);
}
```

```
public void mouseClicked(MouseEvent me) {
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}

public void mouseEntered(MouseEvent me)
{
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
```

```
public void mouseExited(MouseEvent me)
        {
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited.";
        repaint();
        }
.
public void mousePressed(MouseEvent me)
    {
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
    }
```

```
public void mouseReleased(MouseEvent me)
{
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
public void mouseDragged(MouseEvent me)
{
mouseX = me.getX();
mouseY = me.getY();
msg= "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
```

```
public void mouseMoved(MouseEvent me)
{
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}

public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```



Handling Keyboard Events

- To handle **keyboardevents**, we must **implement** the **KeyListener interface**

Handling Keyboard Events- Applet Example

- When a key is pressed, a **KEY_PRESSED event is generated.**
  - This results in a call to the **keyPressed( )** event handler.
- When the key is released, a **KEY_RELEASED event is generated and**
  - **keyReleased( ) handler is executed.**
- If a character is generated by the keystroke, then **a KEY_TYPED event is sent and**
  - **keyTyped( ) handler is invoked.**
- Thus, each time the user presses a key, at least two and often three events are generated.
- Implement an applet program that echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet  implements KeyListener {
String msg = "";
int X = 10, Y = 20;
public void init()
{
addKeyListener(this);
}
```

```
public void keyPressed(KeyEvent ke)
 {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke)
 {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke)
 {
msg += ke.getKeyChar();
repaint();
}
public void paint(Graphics g)
 {
g.drawString(msg, X, Y);
}
}
```

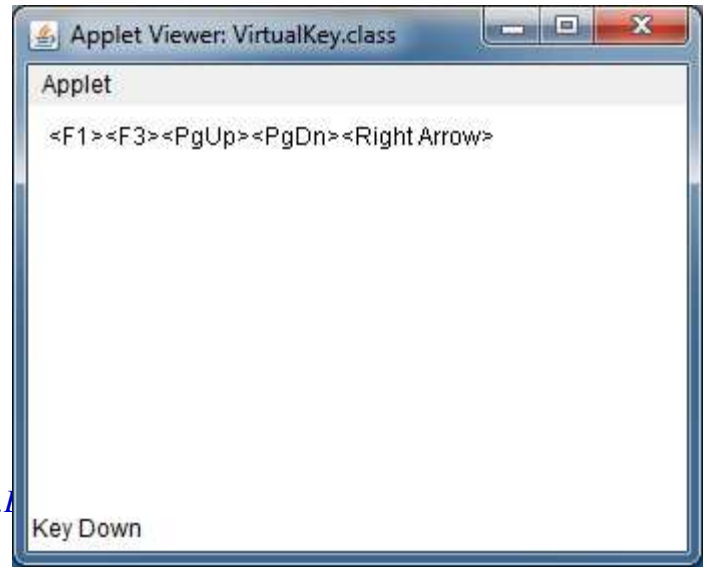
Virtual key display


```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class VirtualKey extends Applet  implements KeyListener {
String msg = "";
int X = 10, Y = 20;
public void init()
{
addKeyListener(this);
}
```

```
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
int key = ke.getKeyCode();
switch(key) {
case KeyEvent.VK_F1:
msg += "<F1>";
break;
case KeyEvent.VK_F2:
msg += "<F2>";
break;
case KeyEvent.VK_F3:
msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
```

```
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;
case KeyEvent.VK_RIGHT:
msg += "<Right Arrow>";
break;
}
repaint();
}
public void keyReleased(KeyEvent ke)
{
showStatus("Key Up");
}
```

```
public void keyTyped(KeyEvent ke)
    {
msg += ke.getKeyChar();
repaint();
}
public void paint(Graphics g)
    {
    g.drawString(msg, X, Y);
    }
}
```



Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 9)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **Multithreaded Programming :**
  - The Java Thread Model
  - The Main Thread
  - Creating Thread

Thread

- Using **thread** we can do multiple activities within a single process.
- E.g In a web browser
  - we can scroll the page while it's downloading an applet or image, play animation and sound concurrently,
  - print a page in the background while you download a new page
- A thread in Java, is the path followed when executing a program.
- All Java **programs** have at least one thread, known as the main thread,
  - which is created by the Java Virtual Machine (JVM) at the program's start, when the main() method is invoked with the main thread.

Multithreaded Programming

- Java provides built-in support for *multithreaded programming*.
- A **multithreaded program** contains **two or more parts** that can run **concurrently(simultaneously)**.
  - Each part of such a program is called a ***thread***,
    - separate memory area is not allocated to threads.
  - Each thread defines a separate path of execution.
- Threads are **lightweight processes** within a process.
- Multithreading is a specialized form of ***multitasking***.
- Multithreading **maximizes the utilization of CPU**

Multitasking

- Multitasking-more than one task run concurrently.
- Two distinct types of multitasking:
  - **process based**
    - *A process-based multitasking* is the feature that allows your computer to **run two or more programs concurrently**
    - E,g. We can execute browser, paint software , calculator , notepad etc at the same time.
  - **thread-based.**
    - The thread is the smallest unit of dispatchable code.
    - A **single program can perform two or more tasks simultaneously.**
    - E.g. text editor can format text at the same time that it is printing, if two actions are being *performed by two separate threads.*

Multithreading

- **ADVANTAGE** Multithreading enables to
  - write **very efficient programs**
  - that make **maximum the use of the CPU**
    - because idle time can be kept to a minimum.
    - This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
  - Threads are independent.
- Applications; Games, animation etc..

Single threaded vs Multithreaded

- In a *single-threaded environment*, one program has to wait for other program to finish —even though the CPU is sitting idle most of the time.
- Multithreading helps to effectively make use of this idle time.

The Java Thread Model

- The Java run-time system **depends on threads** for many things
- All the class libraries are designed with multithreading.
- Java uses threads to enable the entire environment to be **asynchronous**.
  - Asynchronous threading means, a thread once start executing a task, can hold it in mid, save the current state and start executing another task.
- This helps reduce inefficiency by preventing the waste of CPU cycles.

The Java Thread Model(contd.)

- **Single-threaded systems** use an approach called an *event loop with polling*.
  - In this model, *a* single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
  - In a singled-threaded environment, when a thread *blocks* (*that is, suspends execution)* because it is waiting for some resource, the entire program stops running.
- The benefit of Java's **multithreading** is that the **main loop/polling mechanism is eliminated.**
  - One thread can pause **without stopping** other parts of your program.
  - When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

The Java Thread Model(contd.)

Thread –

Threads exist in several states.

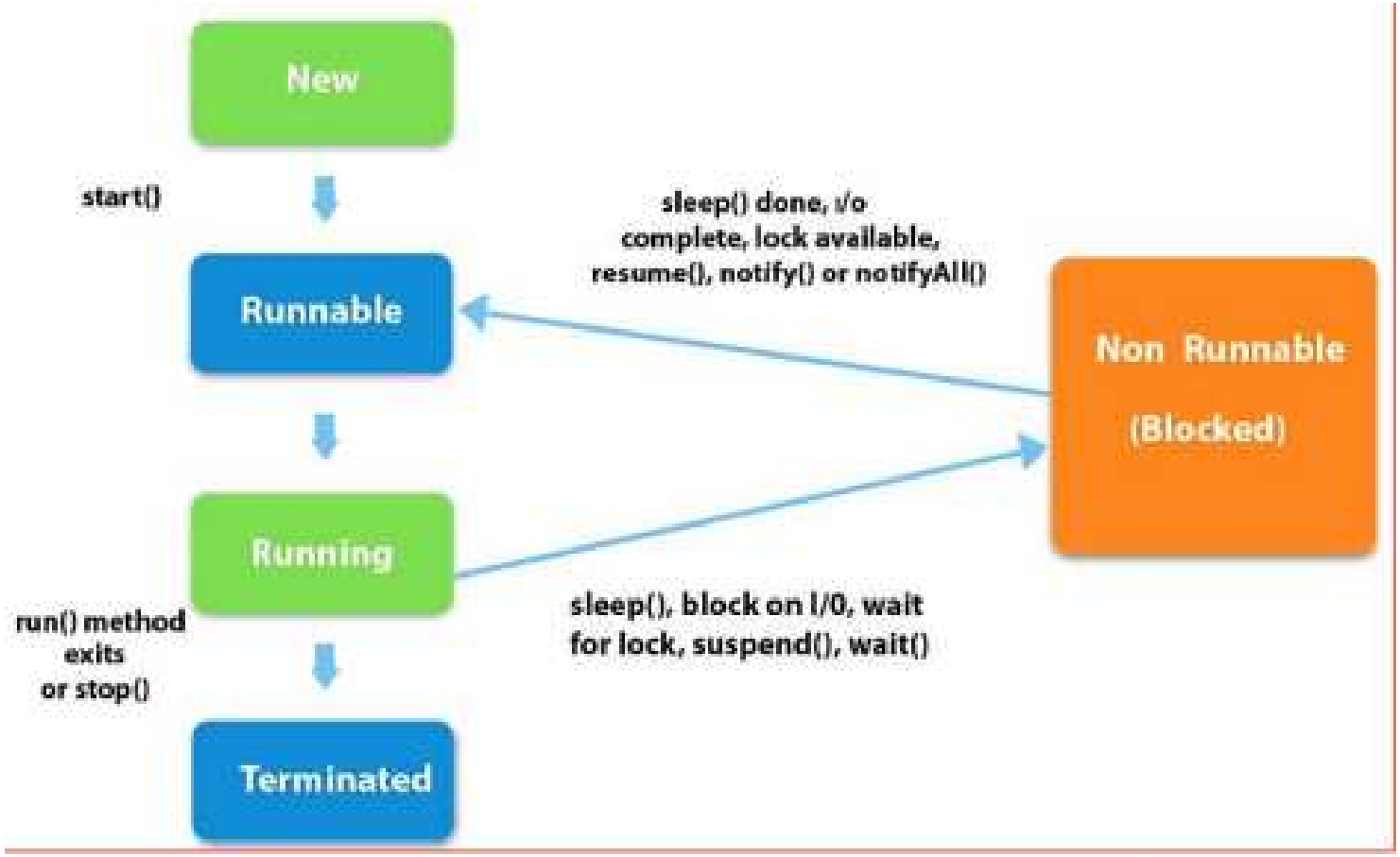
- A thread can be **running**.
- It can be **ready to run(runnable)** as soon as it gets CPU time.
- A running thread can be **suspended(blocked)**, which temporarily suspends its activity.
- A suspended thread can then be **resumed(runnable)**, allowing it to pick up where it left off.
- A thread can be **blocked** when waiting for a resource.
- At any time, a thread can be **terminated**, which halts its execution immediately.
  - Once terminated, a thread cannot be resumed.

The Java Thread Model(contd.)

- Thread States
  - **New**
  - **Runnable –*ready to run***
  - **Running**
  - **Non-Runnable (Blocked)**
  - **Terminated**

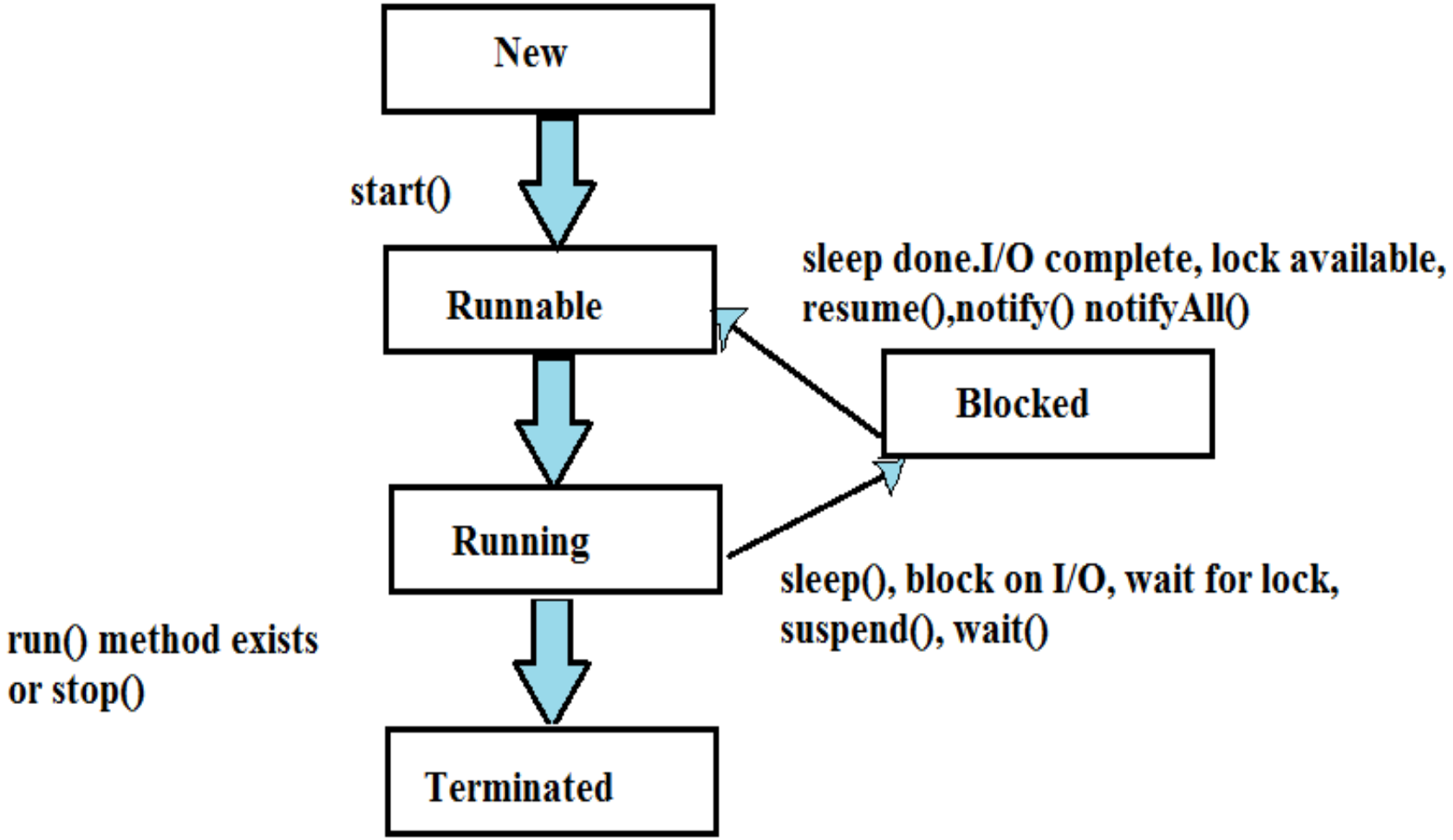
The Java Thread Model(contd.)

- **Thread life cycle**



The Java Thread Model(contd.)

- **Thread life cycle**



The Java Thread Model(contd.)

Thread Priorities

- Java assigns to each thread a priority
  - priority determines how that thread should be treated with respect to the others.
- Thread priorities are **integers**
  - that specify the relative priority of one thread to another
- A higher-priority thread **doesn't run any faster** than a lower-priority thread if it is the only thread running.
- A thread's priority is **used to decide when to switch from one running thread to the next**.
  - Switching from one thread to another is called a **context switch**

The Java Thread Model-Thread priorities(contd.)

- The rules that determine **when a context switch** takes place are:
  - **A thread can voluntarily relinquish control**. This is done by explicitly yielding, sleeping, or blocking on pending I/O. Here all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - **A thread can be preempted by a higher-priority thread.** Here a lower-priority thread is simply preempted(forcely suspended) by a higher-priority thread. i.e. As soon as a higher-priority thread wants to run, it can run. This is called **preemptive multitasking**.

The Java Thread Model-Thread priorities(contd.)

- For operating systems such as Windows,
  - threads of **equal priority** are time-sliced automatically in round-robin fashion.
- For other types of operating systems,
  - threads of **equal priority** must voluntarily yield control to their peers.
  - If they don't, the other threads will not run.

Synchronization

- Multithreading introduces an asynchronous behavior.
- But, when two or more threads need **access** to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*
- Key to synchronization is the concept of the ***monitor*** (also called a *semaphore).*
  - Only one thread can own a monitor at a given time.
- Synchronization in Java can be achieved by
  - **Synchronized Methods**
  - **The synchronized Statement**

Messaging

- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.
- Java's **messaging system**
  - allows a thread to enter a synchronized method on an object,
  - and then wait there until
    - some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its **methods,** its companion interface-**Runnable.**
- Thread encapsulates a thread of execution
- To **create a new thread,** our program will either
  - extend **Thread** ***class*** or
  - implements **Runnable** ***interface*.**

Thread class methods	
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

- When a Java program starts up, one thread begins running immediately.
  - This is usually called the **main thread** of our program, because it is executed when our program **begins**.
- The main thread is important for two reasons:
  1. It is the thread from which other "child" threads will be spawned.
  2. Often, it must be the **last thread to finish execution** because it performs various shutdown actions.

The Main Thread(contd.)

- The main thread is created automatically when our program is started.

- The Main thread can be controlled through a **Thread object.**

  – To do so, we must obtain a reference to the thread by calling the method **currentThread( ),** which is a public static member of Thread class.

  – Its general form is:

static Thread currentThread( )

  - This method returns a reference to the thread in which it is called.

  - Once we have a reference to the main thread, we can control it just like any other thread.

The Main Thread(contd.)

```
class CurrentThreadDemo
{
    public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    t.setName("My Thread");
    System.out.println("After name change: " + t);
    try {
            for(int n = 5; n > 0; n--)
            {
            System.out.println(n);
            Thread.sleep(1000);
            }
    } catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}   } }
```

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Working of the program

- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( ),** and this reference is stored in the local variable t.
- Next, the program displays information about the thread.
- The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed.
- Next, a loop counts down from five, pausing **one second(1000ms)** between each line.
  - This pausing is accomplished by the **sleep( )** method.
  - The argument to sleep( ) specifies the delay period in milliseconds.

Main thread(contd)

In the output ,Thread[**main**,5,**main**]

- Denotes that by default, the **name of the main thread** is **main.**
- Its **priority** is **5,** which is the default value**,**
- **main** is also the **name of the group of threads** to which this thread belongs.

- A ***thread group*** is a data structure that controls the state of a *collection of threads* as a whole.

Main thread(contd)

- If a thread calls **sleep( )** method then execution of that thread is suspended for the specified period of milliseconds.
- Its general form:

```
static void sleep(long milliseconds) throws InterruptedException
```

- The number of milliseconds to suspend is specified in *milliseconds*. *This method may throw* an **InterruptedException.**
- The **sleep( )** method has a second form,

```
static void sleep(long milliseconds, int nanoseconds) throws
                                                InterruptedException
```

  - This form is useful only in environments that allow timing periods as short as nanoseconds.

Main thread(contd)

- We can set the name of a thread by using **setName( ).**
- We can obtain the name of a thread by calling **getName( )**
- These methods are members of the **Thread class** and are declared as:

final void **setName**(String *threadName)*

final String **getName**( )

  - Here, *threadName specifies the name of the thread.*

Creating a Thread

- Java defines two ways for creating thread:
  - implement the **Runnable interface.**
  - extend the **Thread class.**

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable interface.**
- To implement **Runnable, a** class need only implement a single method called **run( ):**

```
public void run( )
```

- Inside **run( ),** we will define the code that constitutes the new thread.
- **run( )** establishes the entry point for another, concurrent thread of execution within ourprogram. This thread will end when **run( )** returns

Implementing Runnable(contd.)

1. Create a class that implements **Runnable.**
2. Instantiate an object of type **Thread** from within that class.
   - ✓ Thread defines several constructors.

Thread(Runnable *threadOb, String threadName)*

   - Here, *threadOb* is an instance of a class that implements the **Runnable interface.** This defines where execution of the thread will begin.
   - The name of the new thread is specified by *threadName*.

1. After the new thread is created, it will start running when we call its **start( ) method,** which is declared within **Thread.**

- **start( )** executes a call to **run( ).**
- The **start( )** method declaration is:

void **start( )**


Java

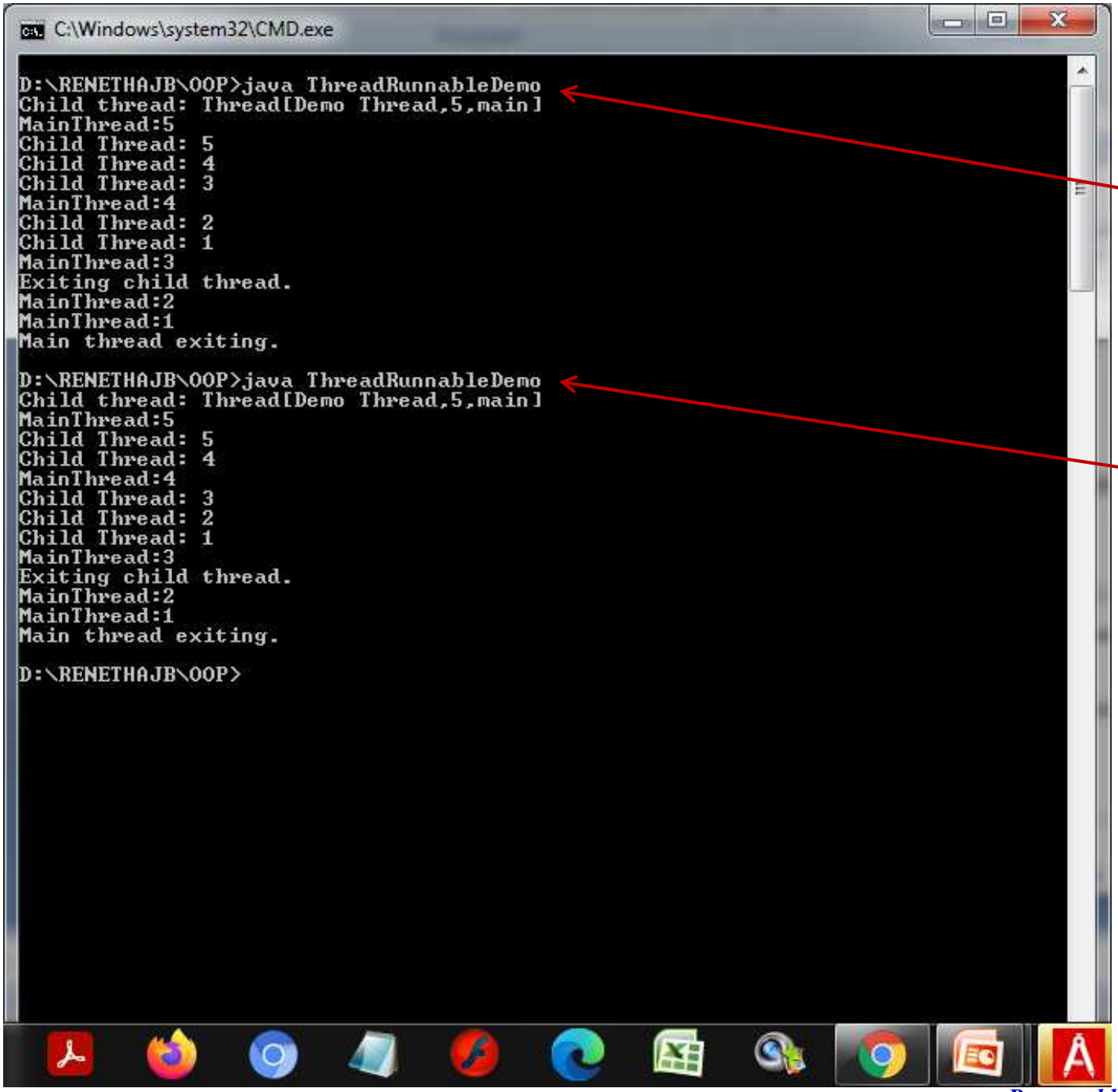
```
class NewThread implements Runnable
 {
      Thread t;
      NewThread()
      {
            t = new Thread(this, "Demo Thread");        //create new Thread
            System.out.println("Child thread: " + t);
            t.start();                                  //thread starts. Calls run()
      }
      public void run()
      {                                 // action to be done by thread
            try {
                  for(int i = 5; i > 0; i--)
                  {
                              System.out.println("Child Thread: " + i);
                              Thread.sleep(500);
                   }
               } catch (InterruptedException e)
                    {
                      System.out.println("Child interrupted.");
                     }
            System.out.println("Exiting child thread.");
      }
}
```

```
class ThreadRunnableDemo
{
    public static void main(String args[])
    {
            new NewThread();
            try {
                    for(int i = 5; i > 0; i--)
                    {
                    System.out.println("MainThread:"+ i);
                    Thread.sleep(1000);
                     }
                } catch (InterruptedException e)
                   {
                            System.out.println("Mainthread interrupted.");
                   }
            System.out.println("Main thread exiting.");
    }
 }
```

Implementing Runnable(contd.)

```
NewThread implements Runnable {
Thread t;
NewThread()
  {
 t = new Thread(this, "Demo Thread");
 System.out.println("Child thread: " + t);
 t.start();
  }
public void run() {
try {
       for(int i = 5; i > 0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(500);       }
       } catch (InterruptedExce ption e)
   { System.out.println("Child interrupted.");
    }
System.out.println("Exiting child thread.");
} }
```

```
class ThreadRunnableDemo
{
public static void main(String args[])
{  new NewThread();
try
{    for(int i = 5; i > 0; i--)
    {System.out.println("MainThread:"+ i);
      Thread.sleep(1000);
      }
  } catch (InterruptedException e) {
System.out.println("Mainthread interrupted.");
                                          }
System.out.println("Main thread exiting.");
}
}
```



Execution

Execution

Implementing Runnable(contd.)

- Inside NewThread's constructor, a new Thread object is created by the following statement:

t = new **Thread**(**this**, "Demo Thread");

- Passing **this** as the first argument indicates that we want the new thread to call the run( ) method on this object.
- Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method.
- This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main( ).
- When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program may vary based on processor speed and task load.

Extending Thread

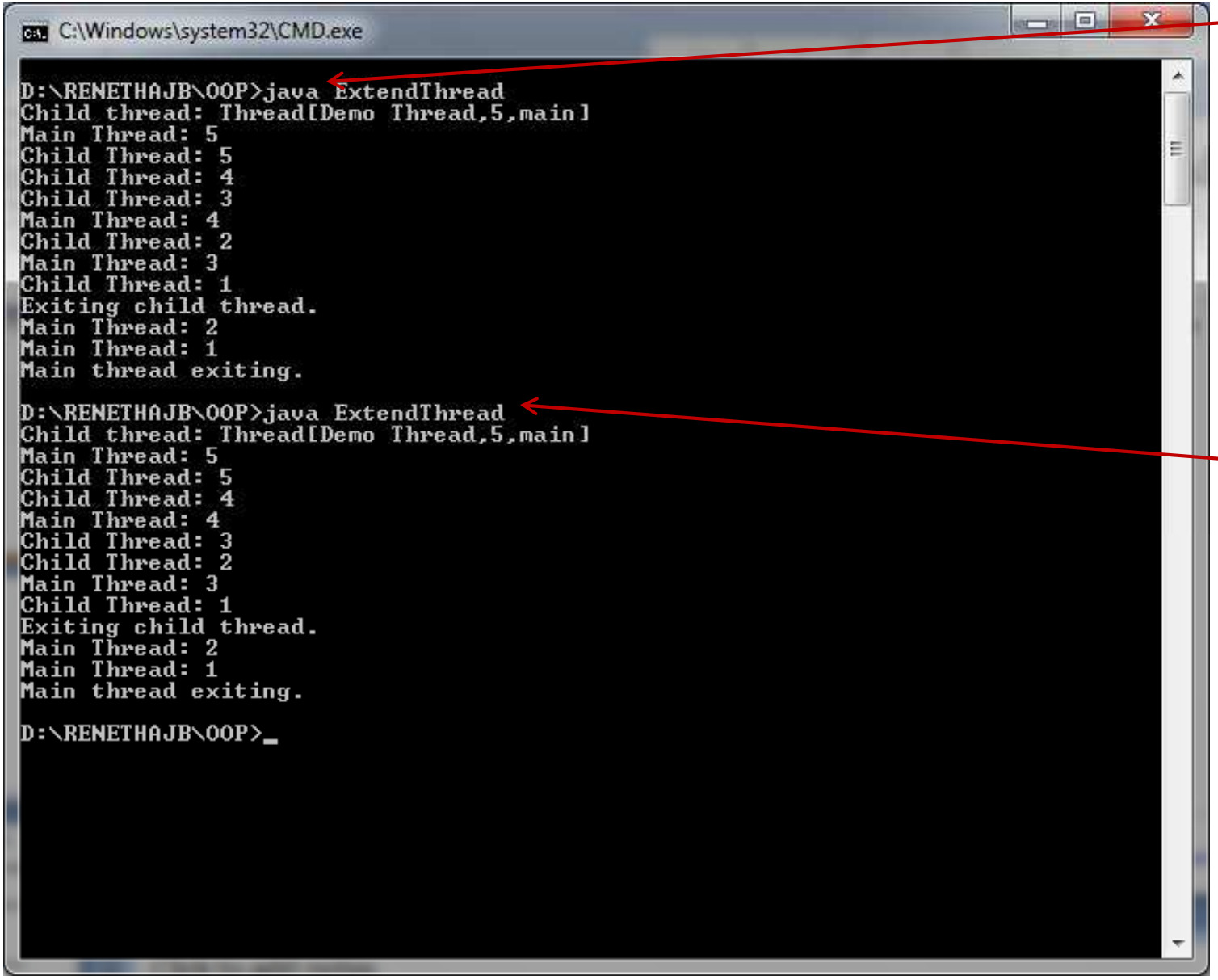
- Another way to create a thread is to
  - Create a **new class that extends Thread,**
    - The extending class must **override** the **run( )** method, which is the entry point for the new thread.
    - It must also call **start( )** to begin execution of the new thread
  - then create an **instance of that class.**

```
class NewThread extends Thread
{
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
    }
    public void run()
    {
        try {
                for(int i = 5; i > 0; i--)
                {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
                }
            } catch (InterruptedException e)
                {
                    System.out.println("Child interrupted.");
                }
        System.out.println("Exiting child thread.");
}     }
```

```
class ExtendThread
{
    public static void main(String args[])
    {
        new NewThread();
        try {
                for(int i = 5; i > 0; i--)
                {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
                }
            } catch (InterruptedException e)
                {
                System.out.println("Main thread interrupted.");
                }
        System.out.println("Main thread exiting.");
    }
}
```

```
class NewThread extends Thread
{
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
    }
    public void run()
    {
    try {
        for(int i = 5; i > 0; i--)
        {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
        }
      } catch (InterruptedException e)
        {
        System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
 }
```

```
class ExtendThread
{
public static void main(String args[])
    {
    new NewThread();
    try {
        for(int i = 5; i > 0; i--)
        {
System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
        }
    } catch (InterruptedException e)
        {
System.out.println("Main threadinterrupted.");
        }
System.out.println("Main thread exiting.");
    }
}
```



Execution

Execution

Extending Thread(contd.)

- The child thread is created by instantiating an object of class **NewThread**, which is derived from **Thread**.
- The call to **super( )** inside NewThread invokes the **Thread** constructor:

```
public Thread(String threadName)
```

  - Here, *threadName specifies the name of the thread*

Choosing an Approach

- If we **will not be overriding any of Thread's other methods,** it is probably best simply to implement **Runnable.**
- The classes should be **extended using Thread** only when they are being **enhanced or modified** in some way.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.

CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 10)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ **Multithreaded Programming :**
  - ❑ Creating Multiple Threads
  - ❑ Synchronization
  - ❑ Suspending, Resuming and Stopping Threads

Creating Multiple Threads

- Our program can spawn as many threads as it needs.
- New threads can be created by
  - Extending **Thread** class
  - Implementing **Runnable** interface

```
class NewThread implements Runnable
    {   String name;
        Thread t;
        NewThread(String threadname)
        {           name = threadname;
                    t = new Thread(this, name);
                    System.out.println("New thread: " + t);
                    t.start();
        }
    public void run()
    {    try { for(int i = 5; i > 0; i--) {
                            System.out.println(name + ": " + i);
                            Thread.sleep(1000);
                            }
          }catch (InterruptedException e) {System.out.println(name +"Interrupted"); }
    System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo
{
    public static void main(String args[])
    {
        new NewThread("One");         // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
                Thread.sleep(10000);
            } catch (InterruptedException e)
                { System.out.println("Main thread Interrupted");
                }
        System.out.println("Main thread exiting.");
        }
}
```

Implementing Runnable(contd.)

```
class NewThread implements Runnable
{    String name;
     Thread t;
     NewThread(String threadname)
     {name = threadname;
     t = new Thread(this, name);
     System.out.println("New thread: " + t);
     t.start();
     }
     public void run()
     { try { for(int i = 5; i > 0; i--)
        { System.out.println(name + ": " + i);
        Thread.sleep(1000);
          }
        }
     catch (InterruptedException e)
     {System.out.println(name +"Interrupted"); }
     System.out.println(name + " exiting.");
     }
}
```

```
class MultiThreadDemo
{public static void main(String args[])
{
new NewThread("One");
new NewThread("Two");
new NewThread("Three");
try{
    Thread.sleep(10000);
    } catch (InterruptedException e)
{ System.out.println("Main thread Interrupted");
                    }
System.out.println("Main thread exiting.");
}
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
Two: 5
New thread: Thread[Three,5,main]
Three: 5
One: 4
Three: 4
Two: 4
Three: 3
One: 3
Two: 3
Three: 2
Two: 2
One: 2
Three: 1
One: 1
Two: 1
Two exiting.
Three exiting.
One exiting.
Main thread exiting.
```

OUTPUT

NOTE:
The ***output*** produced by this program ***may vary based on processor speed and task load***.

All three child threads share the CPU.
The call to **sleep(10000) in main(**).causes the main thread to sleep for ten seconds and Ensures that it will finish last.

isAlive( ) and join()

- Two ways exist to determine whether a thread has finished.
  - ✓ isAlive( ) is defined by Thread, and its general form is shown here:

    final boolean **isAlive( )**
  - – The isAlive( ) method returns **true** if the *thread* upon which it is called is still *running*. It returns false otherwise.
  - ✓ the method that you we more commonly use to wait for a thread to finish is called join( )

    final void **join( )** throws InterruptedException
    - This method **waits until the thread** on which it is called **terminates**

Thread Priorities

- Thread priorities are used by the thread scheduler to decide **when** each thread should be **allowed to run.**
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also **preempt a lower-priority** one.
  - For instance, when a lower-priority thread is running and a **higher-priority thread resumes** (from sleeping or waiting on I/O, for example), **it will preempt the lower priority thread.**

Thread Priorities(contd.)

- To set a thread's priority, use the **setPriority( ) method,** which is a member of Thread.
  - This is its general form:

```
final void setPriority(int level)
```

- To obtain the current priority setting by calling the **getPriority**( ) method of Thread,

```
final int getPriority( )
```

Synchronization

- When two or more threads need access to a **shared resource**, it is necessary to ensure that the **resource will be used by only one thread at a time**. The process by which this is achieved is called *synchronization.*

Synchronization(contd.)

- Key to synchronization is the concept of the **monitor** (also called a ***semaphore***).
- 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.
- *A thread* that owns a monitor can *reenter* the same monitor if it so desires.

Synchronization(contd.)

- We can synchronize our code in any of the following two ways. Both involve the use of the **synchronized keyword**.
  - **synchronized Methods**
  - **The synchronized Statement**

Synchronization(contd.) - **Using Synchronized Methods**

- Synchronization is easy in Java, because all objects have their own **implicit monitor** associated with them.
- To **enter** an object's monitor,
  - just call a method that is 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.**

Synchronized method E.g

- Create a class **Callme** that has method **call( ).**
  - The **call( )** method takes a String parameter called msg.
  - This method tries to **print the msg string inside of square brackets.**
  - After call( ) prints the opening bracket [and the **msg** string, it calls Thread sleep(1000), which pauses the current thread for one second.
  - After that delay call() prints the closing square bracket]

```
class Callme
{
    void call(String msg)
    {
        System.out.print("[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {       System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
```

Synchronized method E.g(contd.)-

- Create a class **Caller.**
- Its constructor takes a reference to *an instance of the* ***Callme*** *class* and a *String*,
  - It store *instance of the* ***Callme*** *in* ***target*** and *String in msg*.
- The constructor of **Caller** also **creates a new thread** that will call this object's run( ) method through ***start()*** method.
  - The thread is started immediately.
  - The run( ) method of Caller calls the **call( )** method on the target instance of Callme, passing in the msg string as argument.

```
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {       target = targ;
            msg = s;
            t = new Thread(this);
            t.start();
    }
    public void run()
    {
            target.call(msg);
    }
}
```

Synchronized method E.g(contd.)-

- **Synch** class starts by
  - creating a single instance of **Callme**, and
  - three instances of Caller,
    - each with a unique message string.
  - The same instance of **Callme** is passed to each **Caller**.

```
class Synch
{
    public static void main(String args[])
    {      Callme target = new Callme();
           Caller ob1 = new Caller(target, "Hello");
           Caller ob2 = new Caller(target, “Ok");
           Caller ob3 = new Caller(target, "World");
           try{      ob1.t.join();
                     ob2.t.join();
                     ob3.t.join();
                }
           catch(InterruptedException e)
           {         System.out.println("Interrupted");
           }
    }
}
```

Without synchronization

```
class Callme
{
    void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
     catch(InterruptedException e)
      { System.out.println("Interrupted");
      }
     System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       target.call(msg);
    } }
```

```
class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, “Ok");
Caller ob3 = new Caller(target, "World");
try{        ob1.t.join();
            ob2.t.join();
            ob3.t.join();
     }
    catch(InterruptedException e)
      { System.out.println("Interrupted“) ;}
  }
}
```

OUTPUT

```
[Hello[World[Ok]
]
]
```

Without synchronization

- Here by calling sleep( ), the call( ) method allows execution to **switch to another thread.** This results in the mixed-up output of the three message strings.
- Here three threads are there. The threads here has no exceution order)
  - One thread tries to print [Hello]
  - One thread tries to print [Hello] [Ok]
  - One thread tries to print [World]
- Threads may execute in any order.
  - So output of this program may be different during different executions.
- All three threads call **the same method, on the same object(target** in main function**), at the same time**. This is known as a ***race condition***, *because the three threads are racing each* other to complete the method.

Without synchronization(contd.)

- One thread executes an invoke call() and prints **[*message*** and then that thread sleeps for 1 second.
- During that time any one of the other threads execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second
- then next thread execute. It invoke call() and prints prints **[*message*** and that thread sleeps for 1 second
- 1 second after the execution of each thread, it wakes up and prints]
- Some of the outputs during executions

```
[Hello[World[Ok]
]
]
```

```
[Ok[World[Hello]
]
]
```

```
[Hello[Ok[World]
]
]
```

- But desired output was [*message] in each line.*
- *One way to solve this problem is to* ***make call() a synchronized method****(serialize access to* ***call( )).***

Synchronized method

- We must *serialize access to* ***call( ).***
  - That is, we must restrict its **access to only one thread at a time**.
  - To do this, we simply need to precede call( )'s definition with the keyword **synchronized**

```
class Callme
{
    synchronized void call(String msg)
    {
            System.out.print("[" + msg);
            try
            {
                    Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {       System.out.println("Interrupted");
            }
            System.out.println("]");
    }
}
```

Using synchronized method

```
class Callme
{
   synchronized void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
     catch(InterruptedException e)
      { System.out.println("Interrupted");
      }
     System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       target.call(msg);
    } }
```

```
class Synch
{
public static void main(String args[])
{ Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Ok");
Caller ob3 = new Caller(target, "World");
try{                    ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
     }
    catch(InterruptedException e)
    { System.out.println("Interrupted") ;}
    }
}
```

OUTPUT *(*outputs may vary)*

```
[Hello]
[World]
[Ok]
```

```
[Hello]
[Ok]
[World]
```

Synchronized method(contd.)

- By prefixing **synchronized** keyword in call() method, it prevents other threads from entering **call( )** while another thread is using it.
  - Here one thread executes an invoke call() and prints [***message*** and then that thread sleeps for 1 second *(during this waiting time other threads using the same object are not allowed to access call())* and after1second it prints].
  - Then any one of the other threads execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second and after1 s it prints].
  - then next thread execute. It invoke call() and prints [***message*** and that thread sleeps for 1 second and after1 s it prints].

The outputs may be different every time we execute

[Hello]
[World]
[Ok]

[Hello]
[Ok]
[World]

Synchronized method(*contd.*)

- If we have a method, or group of methods, that *manipulates the internal state of an object* in a multithreaded situation, we should use the **synchronized keyword** to guard the state from **race conditions**.

- Once a thread enters any **synchronized method** on an instance, **no other thread can enter any other synchronized method on the same instance.**

  – However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

- Creating **synchronized methods** within is an easy and effective means of achieving synchronization, but it **will not work in all cases**.
  - Suppose that we want to **synchronize the access to objects of a class** that **does not use synchronized methods.** Suppose this class was not created by a third party, and we *do not have access to the source code*
  - So we ***can't add synchronized to the appropriate methods*** within the class.
- To solve this, simply put **calls to the methods** defined by this class **inside synchronized block.**

The synchronized Statement(contd.)

- This is the general form of the **synchronized statement:**

```
synchronized(object)
{
    // statements to be synchronized
}
```

  - Here, *object* is a reference to the object being synchronized.
  - A *synchronized block* ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Synchronized block

```
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {       target = targ;
            msg = s;
            t = new Thread(this);
            t.start();
    }
    public void run()
    {
            synchronized(target)
            {
            target.call(msg);
            }
    }
}
```

Using synchronized statement(block)


```
class Callme
{
    void call(String msg)
    { System.out.print("[" + msg);
      try { Thread.sleep(1000); }
      catch(InterruptedException e)
      { System.out.println("Interrupted");}
      System.out.println("]"); } }
class Caller implements Runnable
{   String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
            target = targ;
            msg = s;
            t = new Thread(this);
            t.start();              }
    public void run()
    {       synchronized(target)
            { target.call(msg);
            }
    } }
```

```
class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Ok");
Caller ob3 = new Caller(target, "World");
try{                    ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
     }
    catch(InterruptedException e)
    { System.out.println("Interrupted") ;}
    }
}
```

OUTPUT *(*outputs may vary)*

```
[Hello]
[World]
[Ok]
```

```
[Hello]
[Ok]
[World]
```

The synchronized Statement(contd.)

- The **call( )** method is **not modified** by synchronized**.**
- Instead, the **synchronized statement** is used inside Caller's run( ) method that synchronizes the object **target.**
  - It encloses the statement that calls the function **call()** using the object **target .**

Suspending, Resuming, and Stopping Threads

- Sometimes, suspending execution of a thread is useful.
  - For example, a separate thread can be used **to display the time of day**.
    - If the user doesn't want a clock, then its **thread can be suspended.**
- Once suspended, **restarting** the thread is also a simple matter.

Suspending, Resuming, and Stopping Threads (contd.)

- Prior to Java 2, a program used Thread methods **suspend( )** to pause and **resume( )** to restart the execution of a thread. They have the form :

```
final void suspend( )
```

```
final void resume( )
```

- The **Thread class also defines a method called stop( ) that stops a thread.**

```
final void stop( )
```

Once a thread has been stopped, it cannot be restarted using **resume( ).**

The Modern Way of Suspending, Resuming, and Stopping Threads

- **suspend( ), resume( )** and **stop( )** methods defined by Thread must not be used for new Java programs.
  - These functions are deprecated(not allowed) now. Because they caused serious failures.
- A thread must be designed so that the **run( ) method** periodically **checks** to determine whether that thread should suspend, resume, or stop its own execution.
  - 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.**

Suspending, Resuming, and Stopping Threads (contd.)

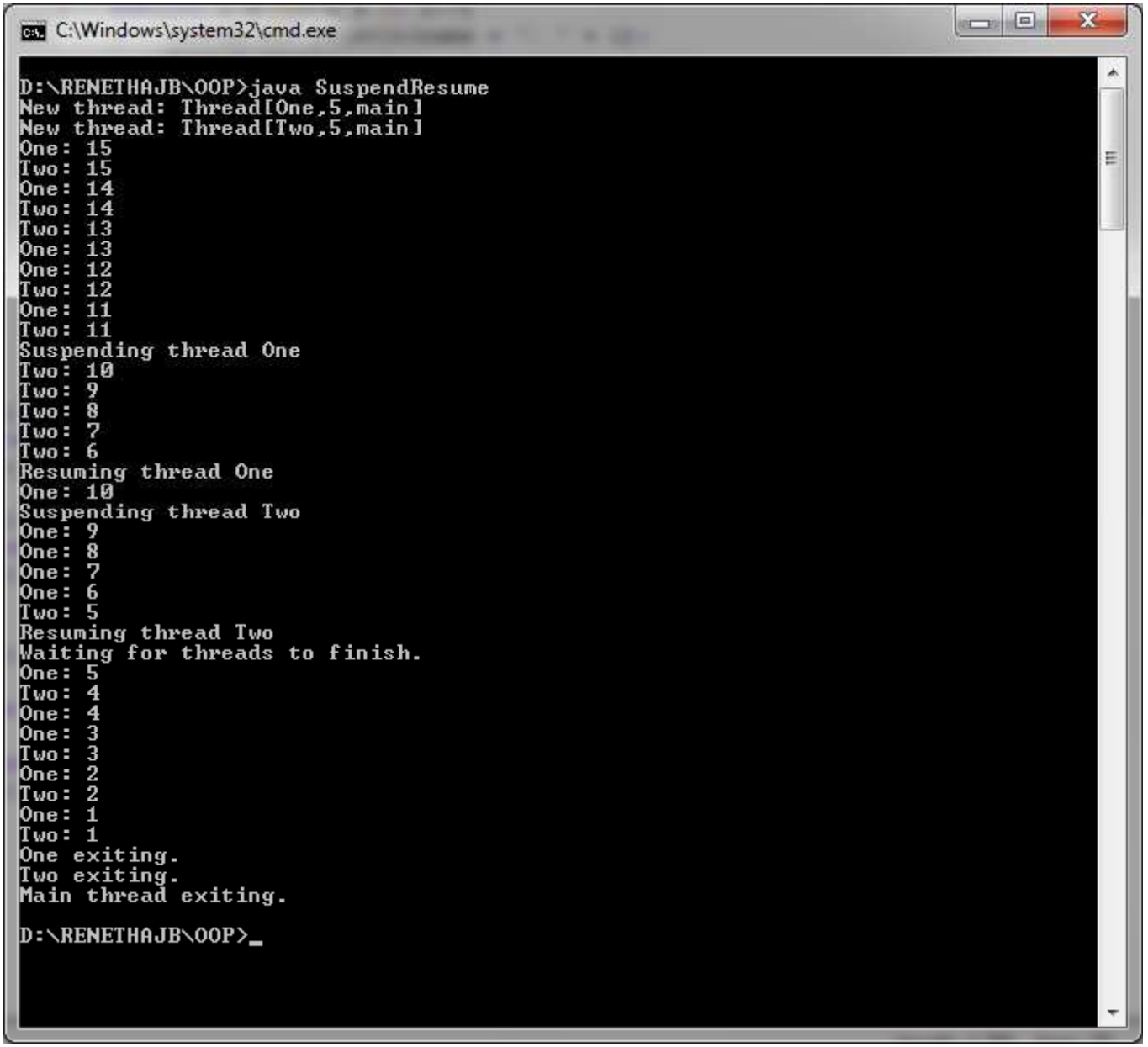
- **wait( )** and **notify( )** methods are inherited from **Object** can be used to control the execution of a thread.
  - **wait( )** method is invoked to suspend the execution of the thread.
  - **notify( )** to wake up the thread.

```
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
    }
```

```
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.");
    }
    System.out.println(name + " exiting.");
}
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();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads
    to finish.");
    ob1.t.join();
    ob2.t.join();
     } catch (InterruptedException e)
    {
    System.out.println("Main thread
    Interrupted");
    }
System.out.println("Main thread exiting.");
    }
    }
```

OUTPUT



*Output may be **different** during different executions.*

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.