MODULE 5

Syllabus:

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

5.1 Enumerations

An *enumeration* is a list of named constants. In Java, enumerations define class types. That is, in Java, enumerations can have constructors, methods and variables. An enumeration is created using the keyword ***enum.*** Following is an example –

```
enum Person
{
     Married, Unmarried, Divorced, Widowed
}
```

The identifiers like ***Married, Unmarried*** etc. are called as ***enumeration Constants.*** Each such constant is implicitly considered as a ***public static final*** member of ***Person***.

After defining enumeration, we can create a variable of that type. Though enumeration is a class type, we need not use ***new*** keyword for variable creation, rather we can declare it just like any primitive data type. For example,

```
Person p= Person.Married;
```

We can use == operator for comparing two enumeration variables. They can be used in ***switch-case*** also. Printing an enumeration variable will print the constant name. That is,

```
System.out.println(p);     // prints as Married
```

Consider the following program to illustrate working of enumerations:

```
enum Person
{
     Married, Unmarried, Divorced, Widowed
}

class EnumDemo
{
     public static void main(String args[])
     {
          Person p1;
```

```
                p1=Person.Unmarried;
                System.out.println("Value of p1 :" + p1);

                Person p2= Person.Widowed;
                if(p1==p2)
                        System.out.println("p1 and p2 are same");
                else
                        System.out.println("p1 and p2 are different");

                switch(p1)
                {
                        case Married: System.out.println("p1 is Married");
                                    break;
                        case Unmarried: System.out.println("p1 is Unmarried");
                                        break;
                        case Divorced: System.out.println("p1 is Divorced");
                                     break;
                        case Widowed: System.out.println("p1 is Widowed");
                                      break;
                }
        }
}
```

5.1.1 The *values()* and *valueOf()* Methods

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

```
public static enum-type[] values()
public static enum-type valueOf(String str)
```

The `values()` method returns an array of enumeration constants. The `valueOf()` method returns the enumeration constant whose value corresponds to the string passed in *str.*

```
enum Person
{
        Married, Unmarried, Divorced, Widowed
}
class EnumDemo
{       public static void main(String args[])
        {       Person p;

                System.out.println("Following are Person constants:");
                Person all[]=Person.values();

                for(Person p1:all)
                        System.out.println(p1);

                p=Person.valueOf("Married");
                System.out.println("p contains "+p);
        }
}
```

Output:

```
Following are Person constants:
Married
Unmarried
Divorced
Widowed
p contains Married
```

5.1.2 Java Enumerations are Class Types

Java enumeration is a class type. That is, we can write constructors, add instance variables and methods, and even implement interfaces. It is important to understand that **each enumeration constant is an object of its enumeration type.** Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
enum Apple
{
	Jonathan(10),  GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

	private int price;

	Apple(int p)
	{
		price = p;
	}

	int getPrice()
	{
		return price;
	}
}

class EnumDemo
{
	public static void main(String args[])
	{
		Apple ap;
		System.out.println("Winesap costs " + Apple.Winesap.getPrice());

		System.out.println("All apple prices:");

		for(Apple a : Apple.values())
			System.out.println(a + " costs " + a.getPrice() + " cents.");
	}
}
```

Output:

Winesap costs 15
All apple prices:
Jonathan costs 10 cents.

```
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

Here, we have member variable ***price,*** a constructor and a member method. When the variable **ap** is declared in **main( )**, the constructor for **Apple** is called once for each constant that is specified.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. Two restrictions that apply to enumerations:

- an enumeration can't inherit another class.
- an **enum** cannot be a superclass.

5.1.3 Enumerations Inherits Enum

All enumerations automatically inherited from **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. We can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value,* and it is retrieved by calling the **ordinal()** method, shown here:

```
final int ordinal( )
```

It returns the ordinal value of the invoking constant. **Ordinal values begin at zero.** We can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

```
final int compareTo(enum-type e)
```

The usage will be –

```
e1.compareTo(e2);
```

Here, e1 and e2 should be the enumeration constants belonging to same enum type. If the ordinal value of e1 is less than that of e2, then compareTo() will return a negative value. If two ordinal values are equal, the method will return zero. Otherwise, it will return a positive number.

We can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**.

```
enum Person
{
      Married, Unmarried, Divorced, Widowed
}

enum MStatus
{
      Married, Divorced
}

class EnumDemo
{
      public static void main(String args[])
      {
         Person p1, p2, p3;
```

```
            MStatus m=MStatus.Married;

            System.out.println("Ordinal values are: ");

            for(Person p:Person.values())
              System.out.println(p + " has a value " + p.ordinal());

            p1=Person.Married;
            p2=Person.Divorced;
            p3=Person.Married;

            if(p1.compareTo(p2)<0)
                 System.out.println(p1 + " comes before "+p2);
            else if(p1.compareTo(p2)==0)
                 System.out.println(p1 + " is same as "+p2);
            else
                 System.out.println(p1 + " comes after "+p2);

            if(p1.equals(p3))
                 System.out.println("p1 & p3 are same");

            if(p1==p3)
                 System.out.println("p1 & p3 are same");

            if(p1.equals(m))
                System.out.println("p1 & m are same");
            else
                 System.out.println("p1 & m are not same");

              //if(p1==m)           Generates error
              //System.out.println("p1 & m are same");
        }
    }
```

5.2 Type Wrappers

Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**. Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, **you can't pass a primitive type by reference to a method.** Also, many of the standard data structures implemented by Java operate on an object, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides ***type wrappers***, which are classes that encapsulate a primitive type within an object.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

- **Character Wrappers:** Character is a wrapper around a char. The constructor for Character is

  Character(char *ch*)

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

  char charValue( )

  It returns the encapsulated character.

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

  Boolean(boolean *boolValue*)
  Boolean(String *boolString*)

  In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false. To obtain a **boolean** value from a **Boolean** object, use

  boolean booleanValue( )

  It return the **boolean** equivalent of the invoking object.

- **The Numeric Type Wrappers:** The most commonly used type wrappers are those that represent numeric values. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

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

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

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

```
Integer(int num)
Integer(String str)
```

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

Ex:

```
class TypeWrap
{
	public static void main(String args[])
	{
		Character ch=new Character('#');
		System.out.println("Character is " + ch.charValue());

		Boolean b=new Boolean(true);
		System.out.println("Boolean is " + b.booleanValue());

		Boolean b1=new Boolean("false");
		System.out.println("Boolean is " + b1.booleanValue());

		Integer iOb=new Integer(12);	//boxing
		int i=iOb.intValue();	//unboxing
		System.out.println(i + " is same as " + iOb);

		Integer a=new Integer("21");
		int x=a.intValue();
		System.out.println("x is " + x);

		String s=Integer.toString(25);
		System.out.println("s is " +s);
	}
}
```

Output:

Character is #
Boolean is true
Boolean is false
12 is same as 12
x is 21
s is 25

5.3 I/O Basics

Java programs perform I/O through streams. A *stream* is a logical device that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. Java defines two types of streams: byte and character. *Byte streams* are used for reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters.

5.3.1 Reading Console Input

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

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

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

Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**. To read a character from a **BufferedReader** , we use ***read()*** method. Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered.

```
import java.io.*;
class BRRead
{
   public static void main(String args[]) throws IOException
   {
      char c;
     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

      System.out.println("Enter characters, 'q' to quit.");

      do
      {
           c = (char) br.read();
           System.out.println(c);
      } while(c != 'q');
   }
}
```

Sample Output:

Enter characters, 'q' to quit.
abcdjqmn
a
b
c
d
j
q

The above program allows reading any number of characters and stores them in buffer. Then, all the characters are read from the buffer till the 'q' is found and are displayed.

In Java, the data read from the console are treated as strings (or sequence of characters). So, if we need to read numeric data, we need to parse the string to respective numeric type and use them later in the program. Following is a program to read an integer value.

```
import java.io.*;
class BRRead
{
  public static void main(String args[]) throws IOException
  {
      int x;
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

      System.out.println("Enter a number:");

      x=Integer.parseInt((br.readLine()).toString());

      x=x+5;
      System.out.println(x);
  }
}
```

5.3.2 Writing Console Output

Console output can be achieved using ***print()*** and ***println()*** methods. The ***PrintStream*** class provides another method ***write()*** which is capable of printing only low-order 8-bit values.

Ex:

```
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
```

5.3.3 PrintWriter Class

PrintWriter is one of the character-based classes. **System.out** is used to write stream of bytes. As there is a limitation for size of bytes, for most generic program (that supports various languages in the world), it is better to use PrintWriter class object to display the output. We can decide whether to flush the stream from the buffer after every newline by setting 2nd argument of the PrintWriter class constructor as ***true.***

```
import java.io.*;
public class PrintWriterDemo
{
  public static void main(String args[])
  {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");

    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

5.3.4 Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte-oriented file stream within a character-based object. Two classes used:

FileInputStream
FileOutputStream

To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Two constructors are of the form:

FileInputStream(String *fileName) throws FileNotFoundException*
FileOutputStream(String *fileName) throws FileNotFoundException*

Here, *fileName specifies the name of the file that you want to open. When you create an* input stream, if the file does not exist, then **FileNotFoundException** is thrown**.** For output streams, if the file cannot be created, then **FileNotFoundException** is thrown**.** When an output file is opened, any preexisting file by the same name is destroyed. When you are done with a file, you should close it by calling **close( ).** To read from a file, you can use a version of **read( )** that is defined within FileInputStream. To write data into a file, you can use the **write( )** method defined by FileOutputStream.

Program to read data from a file:

Note to students: First create a file(using Notepad) with name "test.txt" and save it in the folder (same place where you are going to keep your Java programs). Write some contents into this file. Then create a Java program as shown below –

```
import java.io.*;

class ReadFile
{
  public static void main(String args[]) throws IOException
  {
     int i;
     FileInputStream f;

     try
     {
```

```
            f = new FileInputStream("test.txt");
          } catch(FileNotFoundException e)
          {
               System.out.println("File Not Found");
               return;
          }

          do
          {
             i = f.read();
             if(i != -1)
               System.out.print((char) i);
          } while(i != -1);

          f.close();
        }
    }
```

When you run above program, contents of the *"test.txt"* file will be displayed. If you have not created the *test.txt* file before running the program, then "File Not Found" exception will be caught.

Program to write data into a File:
To read the data from a file, it is obvious that the file must already exist in readable format. But, to write a data into a file, the file need not exist in the folder. Instead, when the statement to open a file to write the data is encountered in the program a file with specified name will be created. The data written will be then stored into it. If the specified file already exists inside the folder, it will be overwritten by the new data. Hence, the programmer must be careful.

Consider the below given program:

```
import java.io.*;

class WriteFile
{
   public static void main(String args[]) throws IOException
   {
     int i;
     FileOutputStream fout;
     char c;

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
     System.out.println("Enter characters, 'q' to quit.");

     try
     {
      fout = new FileOutputStream("test1.txt");
     } catch(FileNotFoundException e)
     {
       System.out.println("Error Opening Output File");
       return;
```

```
      }

      do
      {
          c = (char) br.read();
         fout.write((int)c);
     } while(c != 'q');
   }
}
```

When you run above program, it will ask you to enter few characters. Give some random characters as an input and provide ‘q’ to quit. The program will read all these characters from the buffer and write into the file *“test1.txt”*. Go the folder where you have saved this program and check for a text file *“test1.txt”*. Open the file manually (by double clicking on it) and see that all characters that you have entered are stored in this file.

5.4 Applets

Using Java, we can write either Application or Applet. **Applets** are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

To write an applet, we need to import Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT. The AWT contains support for a window-based, graphical user interface. We also need to import applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet. Consider the below given program:

```
import java.awt.*;
import java.applet.*;

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

public class SimpleApplet extends Applet
{
      public void paint(Graphics g)
      {
         g.drawString("A Simple Applet", 20, 20);
      }
}
```

The class **SimpleApplet** must be declared as public, because it will be accessed by code that is outside the program. The ***paint()*** method is defined by AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. This situation can occur for several reasons:

- the window in which the applet is running can be overwritten by another window and then uncovered.
- the applet window can be minimized and then restored.

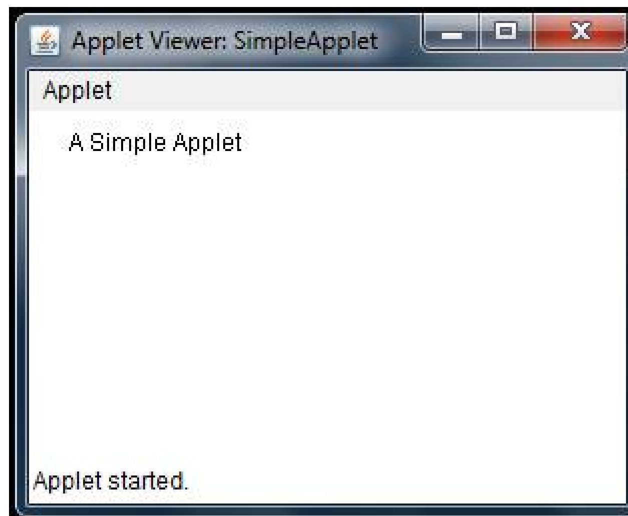
– when the applet begins execution.

The **paint( )** method has one parameter of type ***Graphics.*** This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required. **drawString( )** is a member of the Graphics class used to output a string beginning at the specified X,Y location. (Upper left corner is 0,0)

The compilation of the applet is same as any normal Java program. But, to run the applet, we need some HTML (HyperText Markup Language) support. ***<applet>*** tag is used for this purpose with the attributes *code* which is assigned with name of the class file, and size of the applet window in terms of width and height. The HTML script must be written as comment lines. Use the following statements:

```
javac SimpleApplet.java          //for compilation
appletviewer SimpleApplet.java   //for execution
```

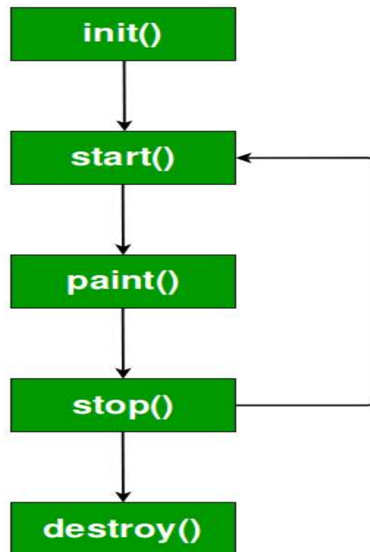
When you run above program, you will get an applet window as shown below –



Applet Life Cycle: Applet class has five important methods, and any class extending Applet class may override these methods. The order in which these methods are executed is known as ***applet life cycle*** as explained below:

- **init():** This is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
- **start( ) :** It is called after init(). It is also called to restart an applet after it has been stopped. start() is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start().
- **paint( ):** This is called each time your applet's output must be redrawn.
- **stop( ) :** This method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop() is called, the applet is probably running. You should use stop() to suspend threads that don't need to run when the applet is not visible. You can restart them when start() is called if the user returns to the page.
- **destroy( ) :** This method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop() method is always called before destroy().

Diagrammatic representation of applet life-cycle is shown in figure given below:



Few key points about applets:

- Applets do not need a **main()** method**.**
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT or Swing

5.5 String Handling

A string is a sequence of character and Java implements strings as objects of type ***String.*** Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed. Once a **String** object has been created, you cannot change the characters of that string. Whenever we need any modifications, a new string object containing modifications has to be created. However, a variable declared as String reference can point to some other String object, and hence can be changed.

In case, we need a modifiable string, we should use ***StringBuffer*** or ***StringBuilder*** classes. ***String, StringBuffer*** and ***StringBuilder*** classes are in ***java.lang*** and are ***final*** classes. Thus, no class can inherit these classes. All these classes implement ***CharSequence*** interface.

5.5.1 The String Constructors

There are several constructors for String class.

1. To create an empty string, use default constructor:
```
String s= new String();
```
2. To create a string and initialize:
```
String s= new String("Hello");
```
3. To create a string object that contains same characters as another string object:
```
String(String strObj);
```

For example,
```
String s= new String("Hello");
String s1= new String(s);
```

4. To create a string having initial values:
```
String(char chars[])
```

For example,
```
char ch[]={'h', 'e', 'l', 'l', 'o'};
String s= new String(ch);  //s contains hello
```

5. To specify a sub-range of a character array as an initializer use the following constructor:
```
String(char chars[ ], int startIndex, int numChars)
```

For example,
```
char ch[]={'a', 'b', 'c', 'd', 'e', 'f'', 'g'};
String s= new String(ch, 2, 3);      //Now, s contains cde
```

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

6. The general forms are:
```
String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)
```

For example,
```
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);   // s1 contains ABCDEF
String s2 = new String(ascii, 2, 3);  // s2 contains CDE
```

- JDK 5 and higher versions have two more constructors. The first one supports the extended Unicode character set.

7. The general form:
```
String(int codePoints[ ], int startIndex, int numChars)
```

here, *codePoints* is array containing Unicode points.

8. Another constructor supports ***StringBuilder:***
```
String(StringBuilder strBuildObj)
```

5.5.2 String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length()** method. For example,
```
String s=new String("Hello");
System.out.println(s.length());  //prints 5
```

5.5.3 Special String Operations

Java supports many string operations. Though there are several string handling methods are available, for the use of programmer, Java does many operations automatically without requiring a call for separate method. This adds clarity to the program. We will now see few of such operations.

- **String Literals:** Instead of using character arrays and *new* operator for creating string instance, we can use string literal directly. For example,

```
char ch[]={'H', 'e', 'l', 'l', 'o'};
String s1=new String(ch);
```

or

```
String s2= new String ("Hello");
```

Can be re-written, for simplicity, as –

```
String s3="Hello";    //usage of string literal
```

A ***String*** object will be created for every string literal and hence, we can even use,

```
System.out.println("Hello".length()); //prints 5
```

- **String Concatenation:** Java does not allow any other operator than + on strings. Concatenation of two or more String objects can be achieved using + operator. For example,

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

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

```
String longStr = "This could have been " +
                 "a very long line that would have " +
                 "wrapped around. But string concatenation " +
                 "prevents this.";
System.out.println(longStr);
```

- **String Concatenation with Other Data Types:** We can concatenate String with other data types. For example,

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

Here, the **int** value in **age** is automatically converted into its string representation within a **String** object. The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**. But, we should be careful while mixing data types:

```
String s= "Four : " + 2 + 2;
System.out.println(s);     //prints Four : 22
```

This is because, "Four :" is concatenated with 2 first, then the resulting string is again concatenated with 2. We can prevent this by using brackets:

```
String s = "Four : " + (2+2);
System.out.println(s);     //prints Four : 4
```

By: Dr. Chetana Hegde, Associate Professor, RNS Institute of Technology, Bangalore – 98
Email: chetanahegde@ieee.org

- **String Conversion and toString():** Java uses **valueOf( )** method for converting data into its string representation during concatenation. **valueOf( )** is a string conversion method defined by **String**. **valueOf( )** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf( )** calls the **toString( )** method on the object. Every class implements **toString( )** because it is defined by **Object**. However, the default implementation of **toString( )** is seldom sufficient. For our own classes, we may need to override toString() to give our own string representation for user-defined class objects. The **toString( )** method has this general form:

```
String toString( )
```

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

```
class Box
{
      double width, height, depth;

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

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

class StringDemo
{
      public static void main(String args[])
      {
            Box b = new Box(10, 12, 14);
            String s = "Box b: " + b;        // concatenate Box object
            System.out.println(s);           // convert Box to string
            System.out.println(b);
      }
}
```

Output:

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

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

5.5.4 Character Extraction Methods

The ***String*** class provides different ways for extracting characters from a string object. Though a ***String*** object is not a character array, many of the ***String*** methods use an index into a string object for their operation.

- **charAt() :** This method is used to extract a single character from a **String.** It has this general form:

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

  Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. For example,

  ```
  char ch;
  ch= "Hello".charAt(1);      //ch now contains e
  ```

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

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

sourceStart	specifies the index of the beginning of the substring
sourceEnd	specifies an index that is one past the end of the desired substring. (i.e. the substring contains the characters from *sourceStart* through *sourceEnd*–1)
target	specifies the array which receives the substring
targetStart	is the index within *target* at which the substring will be copied

Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

```
class StringDemo1
{
      public static void main(String args[])
      {
            String s = "This is a demo of the getChars method.";
            int start = 10;
            int end = 14;
            char buf[] = new char[end - start];
            s.getChars(start, end, buf, 0);
            System.out.println(buf);
      }
}
```

Output:

demo

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

  byte[ ] getBytes( )

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

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

  This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

```
String s1="hello";
char[] ch=s1.toCharArray();

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

5.5.5 String Comparison Methods

The **String** class provides several methods to compare strings or substrings within strings.

- **equals() and equalsIgnoreCase():** To compare two strings for equality, we have two methods:

```
boolean equals(Object str)
boolean equalsIgnoreCase(String str)
```

  Here, *str* is the **String** object being compared with the invoking **String** object. The first method is case sensitive and returns true, if two strings are equal. The second method returns true if two strings are same, whatever may be their case.

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1.equals(s2));     //true
System.out.println(s1.equals(s3));     //false
System.out.println(s1.equals(s4));     //false
System.out.println(s1.equalsIgnoreCase(s4));     //true
```

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

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

startIndex	specifies the index at which the region begins within the invoking **String.**
str2	the **String** being compared.
str2StartIndex	The index at which the comparison will start within *str2*.
numChars	The length of the substring being compared.
ignoreCase	used in second version. If it is **true**, the case of the characters is ignored. Otherwise, case is significant.

```
String s1= "Hello How are you?";
String s2= "how";
```

```
System.out.println(s1.regionMatches(6,s2,0,3));   //false
System.out.println(s1.regionMatches(true,6,s2,0,3));   //true
```

Here, the statement `s1.regionMatches(6,s2,0,3)` will check whether 3 characters of s2 starting from 0th position will match 3 characters of s1 starting from 6th position. Note that, 3 characters starting from 6th position in s1 are "How". And, s2 is "how". These two do not match. If we take another argument *true* for *regionMatches()* method, then case is ignored, and hence it will return true.

- **startsWith( )** and **endsWith():** These are the specialized versions of the **regionMatches()** method. The **startsWith()** method determines whether a given **String** begins with a specified string. The **endsWith()** method determines whether the **String** in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Ex:

```
"Foobar".endsWith("bar")          //true
"Foobar".startsWith("Foo")        //true
```

A second form of **startsWith( )**, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, *startIndex* specifies the index into the invoking string at which point the search will begin.

```
"Foobar".startsWith("bar", 3)  //returns true.
```

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

```
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1.equals(s2));  //true
System.out.println((s1 == s2));     //false
```

- **compareTo():** This method is used to check whether a string is *less than, greater than* or *equal to* the other string. The meaning of *less than, greater than* refers to the dictionary order (based on Unicode). It has this general form:

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

This method will return 0, if both the strings are same. Otherwise, it will return the difference between the ASCII values of first non-matching character. If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase()**, as shown here:

```
int compareToIgnoreCase(String str)
```

Ex:

```
String str1 = "String method tutorial";
String str2 = "compareTo method example";
String str3 = "String method tutorial";

int var1 = str1.compareTo( str2 );
```

```
    System.out.println("str1 & str2 comparison: "+var1);  // -16

    int var2 = str1.compareTo( str3 );
    System.out.println("str1 & str3 comparison: "+var2);  //0
```

5.5.6 Searching Strings

The String class provides two methods `indexOf()` and `lastIndexOf()` that allow you to search a string for a specified character or substring. Both these methods are overloaded to take different types of arguments for doing specific tasks as listed in the table given below –

Method	Purpose
int indexOf(int *ch*)	To search for the first occurrence of a character
int lastIndexOf(int *ch*)	To search for the last occurrence of a character,
int indexOf(String *str*)	To search for the first or last occurrence of a substring
int lastIndexOf(String *str*)	
int indexOf(int *ch*, int *startIndex*)	Used to specify a starting point for the search. Here, *startIndex* specifies the index at which point the search begins. For **indexOf()** method, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )** method, the search runs from *startIndex* to zero.
int lastIndexOf(int *ch*, int *startIndex*)	
int indexOf(String *str*, int *startIndex*)	
int lastIndexOf(String *str*, int *startIndex*)	

Following program demonstrates working of all these functions –

```
class Demo
{
  public static void main(String args[])
  {
     String s = "Now is the time for all good men to come to the aid of their
                                                                  country.";
      System.out.println(s.indexOf('t'));          //7
      System.out.println(s.lastIndexOf('t'));      //65
      System.out.println(s.indexOf("the"));        //7
      System.out.println(s.lastIndexOf("the"));    //55
      System.out.println(s.indexOf('t', 10));      //11
      System.out.println(s.lastIndexOf('t', 60)); //55
      System.out.println(s.indexOf("the", 10));          //44
      System.out.println(s.lastIndexOf("the", 60));      //55
      }
}
```

5.5.7 Modifying a String

Since **String** objects cannot be changed, whenever we want to modify a **String**, we must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with our modifications complete.

- **substring():** Used to extract a substring from a given string. It has two formats:
  - String substring(int *startIndex*): Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.
  - 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.

  Ex:

```
String org = "This is a test. This is, too.";
String result ;

result=org.substring(5);
System.out.println(result);    //is a test. This is, too.

result=org.substring(5, 7);
System.out.println(result);    //is
```

- **concat():** This method can be used 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 +.

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

  is same as

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

- **replace():**The first form of this method replaces all occurrences of one character in the invoking string with another character.

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

  Here, *original* specifies the character to be replaced by the character specified by *replacement*. For example,

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

  puts the string "Hewwo" into s.

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

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

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

  String trim( )

  Here is an example:

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

This puts the string "Hello World" into s by eliminating white-spaces at the beginning and at the end.

5.5.8 Data Conversion using valueOf()

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

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

For example,

```
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);        //prints 3010
```

5.5.9 Changing Case of Characters within a String

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

String toLowerCase()
String toUpperCase()

For example,

```
String str = "Welcome!";
String s1 = str.toUpperCase();
System.out.println(s1);          //prints WELCOME!
String s2= str.toLowerCase();
System.out.println(s2);          //prints welcome!
```

5.5.10 Additional String Methods

Java provides many more methods as shown in the table given below –

Method	Description
int codePointAt(int i)	Returns the Unicode code point at the location specified by i.
int codePointBefore(int i)	Returns the Unicode code point at the location that precedes that specified by i.
int codePointCount(int start, int end)	Returns the number of code points in the portion of the invoking String that is between start and end–1.
boolean contains(CharSequence str)	Returns true if the invoking object contains the string specified by str. Returns false, otherwise.
boolean contentEquals(CharSequence str)	Returns true if the invoking string contains the same string as str. Otherwise, returns false.

boolean contentEquals(StringBuffer str)	Returns true if the invoking string contains the same string as str. Otherwise, returns false.
static String format(String fmtstr, Object ... args)	Returns a string formatted as specified by fmtstr.
static String format(Locale loc, String fmtstr, Object ... args)	Returns a string formatted as specified by fmtstr. Formatting is governed by the locale specified by loc.
boolean matches(string regExp)	Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.
int offsetByCodePoints(int start, int num)	Returns the index with the invoking string that is num code pointsbeyond the starting index specified by start.
String replaceFirst(String regExp, String newStr)	Returns a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr.
String replaceAll(String regExp, String newStr)	Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr.
String[ ] split(String regExp)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp.
String[ ] split(String regExp, int max)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.
CharSequence subSequence(int startIndex, int stopIndex)	Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex. This method is required by the CharSequence interface, which is now implemented by String.

5.5.11 StringBuffer Class

We know that, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writeable character sequences. We can insert characters in the middle or append at the end using this class. **StringBuffer** will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Constructors: The ***StringBuffer*** class has four constructors:

- `StringBuffer()`: Reserves space for 16 characters without reallocation.
- `StringBuffer(int size)` : accepts an integer argument that explicitly sets the size of the buffer
- `StringBuffer(String str)` : accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.

- `StringBuffer(CharSequence chars)` : creates an object that contains the character sequence contained in *chars*

StringBuffer class provides various methods to perform certain tasks, which are mainly focused on changing the content of the string (Remember, String class is immutable – means content of the String class objects cannot be modified). Some of them are discussed hereunder:

- **length()** and **capacity()**: These two methods can be used to find the length and total allocated capacity of ***StringBuffer*** object. As an empty object of StringBuffer class gets 16 character space, the capacity of the object will be sum of 16 and the length of string value allocated to that object. Example:

```
StringBuffer sb = new StringBuffer("Hello");
System.out.println("Original string = " + sb);
System.out.println("length = " + sb.length());     //prints 5
System.out.println("capacity = " + sb.capacity());     //prints 21
```

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

  void ensureCapacity(int *capacity*)

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

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

  char charAt(int *where*)
  void setCharAt(int *where*, char *ch*)

  For **charAt()**, *where* specifies the index of the character being obtained. For **setCharAt()**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. Example:

```
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 would be –

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

- **getChars()**: To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

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

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target.* The index within *target* at which the substring will be copied is passed in *targetStart.* Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

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

  StringBuffer append(String *str*)
  StringBuffer append(int *num*)
  StringBuffer append(Object *obj*)

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

```
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a=").append(a).append("!").toString();
System.out.println(s);      //prints a=42!
```

- **insert()**: The **insert()** method inserts one string into another. It is overloaded to accept values of all the simple types, plus **String**s, **Object**s, and **CharSequence**s. Like **append()**, it calls **String.valueOf( )** to obtain the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. Few forms are:
  – StringBuffer insert(int *index*, String *str*)
  – StringBuffer insert(int *index*, char *ch*)
  – StringBuffer insert(int *index*, Object *obj*)

  Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object. Example:

```
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);     //I like Java
```

- **reverse()**: Used to reverse the characters within a string.

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

- **delete() and deleteCharAt()**: You can delete characters within a **StringBuffer** by using the methods **delete()** and **deleteCharAt().** These methods are shown here:

  StringBuffer delete(int *startIndex*, int *endIndex*)

  It deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index **one past the last character** to

remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1.The resulting **StringBuffer** object is returned.

StringBuffer deleteCharAt(int *loc*)

It deletes the character at the index specified by *loc.* It returns the resulting **StringBuffer** object.
Example:

```
StringBuffer sb = new StringBuffer("This is a test");
sb.delete(4, 7);
System.out.println("After delete: " + sb);    //This a test
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);    //his a test
```

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

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

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

```
StringBuffer sb = new StringBuffer("This is a test");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb); //This was a test
```

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

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

  The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

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

int indexOf(String str, int startIndex)	Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String str)	Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String str, int startIndex)	Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int start, int num)	Returns the index with the invoking string that is num code points beyond the starting index specified by start.
CharSequence subSequence (int startIndex, int stopIndex)	Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex. This method is required by the CharSequence interface, which is now implemented by StringBuffer.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents.

5.5.12 StringBuilder Class

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

5.6 Other Topics

5.6.1 Using *instanceof*

Sometimes, we need to check type of the object during runtime of the program. We may create multiple classes and objects to these classes in a program. In such situations, the ***instanceof*** operator is useful The *instanceof* operator will return Boolean value – true or false.

```
class A
{
   int i, j;
}
class B
{
  int i, j;
}
class C extends A
{
   int k;
}
```

By: Dr. Chetana Hegde, Associate Professor, RNS Institute of Technology, Bangalore – 98
Email: chetanahegde@ieee.org

```
class InstanceOfEx
{
  public static void main(String args[])
  {
     A a = new A();
     B b = new B();
     C c = new C();

     if(a instanceof A)    //this is true
           System.out.println("a is instance of A");   //will be printed

     if(b instanceof A)    //this is false
           System.out.println("b is instance of B");   //will not be printed

      A ob;
     ob = c;

      if(ob instanceof C)   // true because of inheritance
           System.out.println("ob is instance of C"); // will be printed
   }
}
```

5.6.2 Static Import

The statement *static import* expands the capabilities of the import keyword. By following import with the keyword static, an import statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.
We have observed earlier that when we need to use some Math functions, we need to use Math.sqrt(), Math.pow() etc. Using static import feature, we can just use sqrt(), pow() etc. as shown below –

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

class Hypot
{
      public static void main(String args[])
      {
           double side1, side2;
           double hypot;
           side1 = 3.0;
           side2 = 4.0;
           hypot = sqrt(pow(side1, 2) + pow(side2, 2));
           System.out.println(" the hypotenuse is " + hypot);
      }
}
```

5.6.3 Invoking Overloaded Constructors through *this()*

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the **this** keyword**.** The general form is shown here:

this(*arg-list)*

When ***this()*** is executed, the overloaded constructor that matches the parameter list specified by *arg-list* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to **this()** must be the first statement within the constructor.

```
class MyClass
{
      int a,  b;
      MyClass(int i, int j)
      {
            a = i;
            b = j;
      }
      MyClass(int i)
      {
            this(i, i);     // invokes MyClass(i, i)
      }
      MyClass( )
      {
            this(0); // invokes MyClass(0)
      }

      void disp()
      {
            System.out.println("a="+a + " b="+b);
      }
}

class thisDemo
{
  public static void main(String args[])
  {
      MyClass m1 = new MyClass();
      m1.disp();

      MyClass m2 = new MyClass(8);
      m2.disp();

      MyClass m3 = new MyClass(2,3);
      m3.disp();
  }
}
```

Output:

a= 0 b=0
a= 8 b=8
a= 2 b=3

Questions:

1. Define Enumerations. Give an example.
2. Discuss values() and valueOf() methods in Enumerations with suitable examples.
3. "Enumerations in Java are class types" - justify this statement with appropriate examples.
4. Write a note on ordinal() and compareTo() methods.
5. What are wrapper classes? Explain with examples.
6. Write a program to read n integers from the keyboard and find their average.
7. Write a program to read data from keyboard and to store it into a file.
8. Write a program to read data from an existing file and to display it on console.
9. Define an Applet. Write a program to demonstrate simple applet.
10. Explain life cycle of an applet.
11. List and explain any four constructors of String class with suitable examples.
12. Write a note on following String class methods:
    - (i) charAt()
    - (ii) toCharArray()
    - (iii) regionMatches()
    - (iv) startsWith() and endsWith()
    - (v) replace()
    - (vi) trim()
    - (vii) substring()

13. Explain various forms of indexOf() and lastIndexOf() methods with a code snippet.
14. Differentiate StringBuffer class methods length() and capacity().
15. Write a note on StringBuffer class methods:
    - (i) setCharAt()
    - (ii) append()
    - (iii) insert()
    - (iv) reverse()
    - (v) delete()
    - (vi) deleteCharAt()
16. Write a note on
    - (i) instanceof Operator
    - (ii) static import
    - (iii) this()