MODULE 5

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

ENUMERATIONS

- **"Enumeration** means a list of named constant**,** enum in java is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc.
- It is available from JDK 1.5.
- An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword.
- Each enumeration constant is *public*, *static* and *final* by default.
- Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**.
- Enumeration variables are used and declared in much a same way as you do a primitive variable.

ENUMERATION FUNDAMENTALS

How to Define and Use an Enumeration

- An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject //Enumeration defined
{
JAVA, CPP, C, DBMS
}
```

- Identifiers **JAVA, CPP, C and DBMS** are called **enumeration constants**. These are public, static and final by default.
- Variables of Enumeration can be defined directly without any **new** keyword.

Ex: Subject **sub;**

- Variables of Enumeration type can have only enumeration constants as value.

- We define an enum variable as: enum_variable = enum_type.enum_constant;

  Ex: sub = Subject.Java;
- Two enumeration constants can be compared for equality by using the = = relational operator.

Example:

```
if(sub == Subject.Java)
{
...
}
```

Program 1: Example of Enumeration

```
enum WeekDays
{
        sun, mon, tues, wed, thurs, fri, sat
}
class Test
{
        public static void main(String args[])
        {
                WeekDays wk; //wk is an enumeration variable of type WeekDays
                wk = WeekDays.sun;
                //wk can be assigned only the constants defined under enum type Weekdays
                System.out.println("Today is "+wk);
        }
}
```

Output : Today is sun

Program 2: Example of Enumeration using switch statement

```
enum SpecialStud
{
        ABI, GAUTHAM, NANDA, SOURABH, SUHAS, SHARATH
}
```

```
class Test
{
        public static void main(String args[])
        {
                SpecailStud s;
                s = SpecailStud.SHARATH;
                switch(s)
                {
                        case ABI:
                        System.out.println("Cricket champion " + s.ABI);
                        break;
                        case GAUTHAM:
                        System.out.println("Football Champion " + s.GAUTHAM);
                        break;
                        case NANDA:
                        System.out.println("KLE student" + s.NANDA);
                        break;
                        case SOURABH:
                        System.out.println("Mechanical Hero " + s.SOURABH);
                        break;
                        case SUHAS:
                        System.out.println("Talkitive Hero " + s.SUHAS);
                        break;
                        case SHARATH:
                        System.out.println("Bunking Hero " + s.SHARATH);
                        break;

                }
        }
}
```

Output:

```
Bunking Hero SHARATH
```

values() and valueOf() Methods

- The java compiler internally adds the values() method when it creates an enum.
- The values() method returns an array containing all the values of the enum.
- Its general form is,

  public **static** *enum-type[ ]* **values()**
- valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method.
- It's general form is,

  public **static** *enum-type* **valueOf** (String *str*)

Program 3: Example of enumeration using values() and valueOf() methods:

```
enum SpecialStud
{
        ABI, GAUTHAM, NANDA, SOURABH, SUHAS, SHARATH
}
class Test
{
        public static void main(String args[])
        {
                SpecialStud s;
                System.out.println("All constants of enum type Students are:");
                SpecialStud sArray[] = SpecialStud.values();
                //returns an array of constants of type Restaurants
                for(SpecailStud a : sArray) //using foreach loop
                                System.out.println(a);
                s = SpecailStud.valueOf("SUHAS");
                System.out.println("TALKITIVE BOY " + s);
        }
}
```

Output:

```
All constants of enum type students are:
ABI
```

```
GAUTHAM
NANDA
SOURABH
SUHAS
SHARATH
TALKITIVE BOY SUHAS
```

Points to remember about Enumerations

1. Enumerations are of class type, and have all the capabilities that a Java class has.
2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

JAVA ENUMERATIONS ARE CLASS TYPES

Enumeration with Constructor, instance variable and Method

- Java Enumerations Are Class Type.
- It is important to understand that each enum constant is an object of its enumeration type.
- 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

Program 4: Example of Enumeration with Constructor, instance variable and Method

```
enum Apple2
{
        Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
        // variable
        int price;
        // Constructor
        Apple2(int p)
        {
                price = p;
        }
//method
        int getPrice()
        {
                return price;
        }
}
public class EnumConstructor
{
        public static void main(String[] args)
        {
                Apple2 ap;
                // Display price of Winesap.
                System.out.println("Winesap costs " + Apple2.Winesap.getPrice() + "
                cents.\n");
                System.out.println(Apple2.GoldenDel.price);
                // Display all apples and prices.
                System.out.println("All apple prices:");
                for(Apple2 a : Apple2.values())
                System.out.println(a + " costs " + a.getPrice() + " cents.");
        }
}
```

Output:

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

- In this example as soon as we declare an enum variable(Apple2 ap) the constructor is called once, and it initializes value for every enumeration constant with values specified with them in parenthesis.

ENUMERATIONS INHERITS ENUM

- All enumerations automatically inherit one: **java.lang.Enum.** This class defines several methods that are available for use by all enumerations.
- You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the **ordinal( )** method.
- It has this general form: final int ordinal( )
- It returns the ordinal value of the invoking constant. Ordinal values begin at zero.
- Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.
- You can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method.
- It has this general form: final int compareTo(enum-type e), Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant
- If the invoking constant has an ordinal value less than e's, then compareTo( ) returns a negative value.
- If the two ordinal values are the same, then zero is returned.
- If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

Program 5: Example with ordinal(), comapareTo and equals() methods

```
enum Apple5
{
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
public class EnumOrdinal {
        public static void main(String[] args) {
                Apple5 ap, ap2, ap3;
                System.out.println("Here are all apple constants and their ordinal values: ");
                for(Apple5 a : Apple5.values())
                        System.out.println(a + " " + a.ordinal());
                ap = Apple5.RedDel;
                ap2 = Apple5.GoldenDel;
                ap3 = Apple5.RedDel;
                System.out.println();
                if(ap.compareTo(ap2) < 0)
                System.out.println(ap + " comes before " + ap2);
                if(ap.compareTo(ap2) > 0)
                System.out.println(ap2 + " comes before " + ap);
                if(ap.compareTo(ap3) == 0)
                System.out.println(ap + " equals " + ap3);
                System.out.println();
                if(ap.equals(ap2))
                System.out.println("Error!");
                if(ap.equals(ap3))
                System.out.println(ap + " equals " + ap3);
                if(ap == ap3)
                System.out.println(ap + " == " + ap3);
        }
}
```

Output:

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

Program 6: Example of Decisions Makers

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

Output:

```
No
Soon
Soon
Yes
```

TYPE WRAPPERS

- Java uses primitive data types such as int, double, float etc. to hold the basic data types for the sake of performance.
- Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type.
- For example, many data structures in Java operate on objects. So you cannot use primitivedata types with those data structures.
- To handle such type of situations, Java provides type Wrappers which provide classes that encapsulate a primitive type within an object.
- Character : It encapsulates primitive type char within object.

Character (char *ch*)

- To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

  char charValue( ): It returns the encapsulated character.

- Boolean : It encapsulates primitive type boolean within object.

  Boolean (boolean *boolValue*) : *boolValue* must be either **true** or **false**

  Boolean(String *boolString*): if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

- Numeric type wrappers : It is the most commonly used type wrapper.

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

Example:

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

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

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

I/O, Applets, and Other Topics

I/O Basics:

- Till now in the progrms **print( )** and **println( )**, none of the I/O methods have been used significantly.
- The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java's Abstract Window Toolkit (AWT) or Swing for interaction with the user.
- Text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world.
- Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs
- Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent

Streams:

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- An input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.
- An output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network.
- Java defines **two types of streams: byte and character**

Byte Streams and Character Streams

Byte Streams:

- *Byte streams* provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- The original version of Java (Java 1.0) was I/O was byte-oriented.

- At the lowest level, all I/O is still byte-oriented.

Character Streams:

- *Character streams* provide a convenient means for handling input and output of characters.
- They use Unicode and, therefore, can be internationalized.
- Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated.
- The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes

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

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream

InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output

The Character Stream Classes

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

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
PipedReader	Input pipe
PipedWriter	Output pipe

The Predefined Streams:

- All Java programs automatically import the **java.lang** package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System.**
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to standard input, which is the keyboard by default.

- **System.err** refers to the standard error stream, which also is the console by default.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console.

Reading Console Input

- The only way to perform console input was to use a byte stream during older versions. The preferred method of reading console input now is to use a character-oriented stream, which makes your program easier to internationalize and maintain.
- 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.

  BufferedReader(Reader *inputReader*)
- Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class.
- 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.*
- *Combining together we have*

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

Reading Characters:

- To read a character from a BufferedReader, use read( ).
- The version of read( ) that we will be using is

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

Example:

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');
        }
}
```

Output:

```
Enter characters, 'q' to quit.
Pinkuq
P
i
n
k
u
q
```

Reading Strings

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

  String readLine( ) throws IOException: it returns a String object.

Example:

```
import java.io.*;
class BRReadLines {
        public static void main(String args[]) throws IOException
        {
```

```
                    // create a BufferedReader using System.in
                    BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
                    String str;
                    System.out.println("Enter lines of text.");
                    System.out.println("Enter 'stop' to quit.");
                    do {
                              str = br.readLine();
                              System.out.println(str);
                    } while(!str.equals("stop"));
          }
}
```

Output:

```
Enter lines of text.
Enter 'stop' to quit
Suhas speaks fluent english stop
Suhas
speaks
fluent
english
stop
```

Writing Console Output

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

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

Example:

```
class WriteDemo {
                    public static void main(String args[]) {
```

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

The PrintWriter Class:

- **PrintWriter** is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your program.
- **PrintWriter** defines several constructors. The one we will use is shown here:

    PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

- Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a **println( )** method is called.
- If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.
- **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**.
- For example, this line of code creates a **PrintWriter** that is connected to console output:

    PrintWriter pw = new PrintWriter(System.out, true);

Example:

```
import java.io.*;
public class PrintWriterDemo {
        public static void main(String args[]) {
                PrintWriter pw = new PrintWriter(System.out, true);
                pw.println("Abdul Kalamji");
                int i = -7;
                pw.println(i);
        }
}
```

Output:

```
Abdul Kalamji
-7
```

Reading and Writing Files

- Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files.
- To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.

    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( )**.

    void close( ) throws IOException
- To read from a file, you can use a version of **read( )**

    int read( ) throws IOException

```
import java.io.*;
class ShowFile {
        public static void main(String args[]) throws IOException
        {
                int i;
                FileInputStream fin;
                try {
                        fin = new FileInputStream(args[0]);
                } catch(FileNotFoundException e) {
                        System.out.println("File Not Found");
                        return;
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Usage: ShowFile File");
                        return;
                }
                do {
                        i = fin.read();
                        if(i != -1) System.out.print((char) i);
                } while(i != -1);
                fin.close();
        }
}
```

- To write to a file, you can use the **write( )** method

```
import java.io.*;
class CopyFile {
        public static void main(String args[]) throws IOException
        {
                int i;
                FileInputStream fin;
                FileOutputStream fout;
                try {
                        // open input file
                        try {
                                fin = new FileInputStream(args[0]);
                        } catch(FileNotFoundException e) {
                                System.out.println("Input File Not Found");
                                return;
                        }
                        try {
                                fout = new FileOutputStream(args[1]);
                        } catch(FileNotFoundException e) {
                                System.out.println("Error Opening Output File");
                                return;
                        }
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Usage: CopyFile From To");
                        return;
        }
        // Copy File
        try {
            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);
        } catch(IOException e) {
            System.out.println("File Error");
        }
        fin.close();
        fout.close();
    }
}
```

Applet Fundamentals

- *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.

Example:

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

Description:

- The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface.

- The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that you create must be a subclass of **Applet**.
- The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program
- Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the 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. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
- The **paint( )** method has one parameter of type **Graphics**
- Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class.
- This method outputs a string beginning at the specified X,Y location. It has the following general form:

  void drawString(String *message*, int *x*, int *y*)
- Here, *message* is the string to be output beginning at *x,y.*
- Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method.

- However, running **SimpleApplet** involves a different process. In fact, there are two ways in which you can run an applet:

1. Executing the applet within a Java-compatible web browser.
2. Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

To execute an applet in a web browser, you need to write a short HTML text file that contains a tag that loads the applet.

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

- The **width** and **height** statements specify the dimensions of the display area used by the applet.

Steps for excution of applet

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

Points that you should remember now:

- 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.

The transient and volatile Modifiers

- Java defines two interesting type modifiers: **transient** and **volatile**
- When an instance variable is declared as **transient**, then its value need not persist when an object is stored.

For example:

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

- Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.
- The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program

Example:

- In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable.

- The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered.
- While this approach works fine, it may be inefficient at times.
- In some cases, all that really matters is that the master copy of a variable always reflects its current state.
- To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable

Using instanceof

- Knowing the type of an object during run time: we use instanceof
- The **instanceof** operator has this general form:

  objref instanceof *type*
- Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**.

Example:

```
class A {
        int i, j;
}
class B {
        int i, j;
}
class C extends A {
        int k;
}
class InstanceOf {
        public static void main(String args[]) {
                A a = new A();
                B b = new B();
                C c = new C();
                if(a instanceof A)
                System.out.println("a is instance of A");
                if(b instanceof B)
                System.out.println("b is instance of B");
                if(c instanceof C)
                System.out.println("c is instance of C");
                if(c instanceof A)
                System.out.println("c can be cast to A");
        }
}
```

Output

```
a is instance of A
b is instance of B
c is instance of C
c can be cast to A
```

strictfp

- By modifying a class or a method with strictfp, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java.
- When a class is modified by strictfp, all the methods in the class are also modified by strictfp automatically.

```
strictfp class MyClass { //...
```

- Most programmers never need to use strictfp, because it affects only a very small class of problems.

Native Methods

- Occasionally you may want to call a subroutine that is written in a language other than Java.
- Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code.

For example,

- You may want to call a native code subroutine to achieve faster execution time.
- Or, you may want to use a specialized, third-party library, such as a statistical package.

- Java provides the native keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method

For example:

```
public native int meth() ;
public class NativeDemo {
        int i;
        public static void main(String args[]) {
                NativeDemo ob = new NativeDemo();
                ob.i = 10;
                System.out.println("This is ob.i before the native method:" +
                ob.i);
                ob.test(); // call a native method
                System.out.println("This is ob.i after the native method:" +
                ob.i);
        }
        // declare native method
        public native void test() ;
        // load DLL that contains static method
        static {
                System.loadLibrary("NativeDemo");
        }
}
```

- Notice that the **test( )** method is declared as **native** and has no body. This is the method that we will implement in C shortly.
- The library is loaded by the **loadLibrary( )** method, which is part of the **System** class. This is its general form:

  static void loadLibrary(String *file name*)
- After you enter the program, compile it to produce **NativeDemo.class**. Next, you must use **javah.exe** to produce one file: **NativeDemo.h**.
- To produce **NativeDemo.h**, use the following command:

```
javah -jni NativeDemo
```

- This command produces a header file called **NativeDemo.h**. This file must be included in the C file that implements **test( )**.

```
#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
        jclass cls;
        jfieldID fid;
        jint i;
        printf("Starting the native method.\n");
        cls = (*env)->GetObjectClass(env, obj);
        fid = (*env)->GetFieldID(env, cls, "i", "I");
        if(fid == 0) {
                printf("Could not get field id.\n");
                return;
        }
        i = (*env)->GetIntField(env, obj, fid);
        printf("i = %d\n", i);
        (*env)->SetIntField(env, obj, fid, 2*i);
        printf("Ending the native method.\n");
}
```

Problems with Native Methods

1. Potential security risk

- Because a native method executes actual machine code, it can gain access to any part of the host system.
- That is, native code is not confined to the Java execution environment. This could allow a virus infection
- The loading of DLLs can be restricted, and their loading is subject to the approval of the security manager

2. Loss of portability

- Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program each native method is CPU- and operating system–dependent, each DLL is inherently nonportable.
- Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed.

Using assert

- It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program.
- At run time, if the condition actually is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown
- The **assert** keyword has two forms. The first is shown here:

  assert *condition*;
- Here, *condition* is an expression that must evaluate to a Boolean result.

Example:

```
class AssertDemo {
        static int val = 3;
        // Return an integer.
        static int getnum() {
                return val--;
        }
        public static void main(String args[])
        {
                int n;
                for(int i=0; i < 10; i++) {
                        n = getnum();
                        assert n > 0; // will fail when n is 0
                        System.out.println("n is " + n);
                }
        }
}
```

- The second form of **assert** is shown here:

  assert *condition* : *expr*;

- *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails.

```
class AssertDemo {
        // get a random number generator
        static int val = 3;
        // Return an integer.
        static int getnum() {
                return val--;
        }
        public static void main(String args[])
        {
                int n = 0;
                for(int i=0; i < 10; i++) {
                        assert (n = getnum()) > 0; // This is not a good idea!
                        System.out.println("n is " + n);
                }
        }
}
```

Assertion Enabling and Disabling Options

- When executing code, you can disable or enable assertions
- To enable assertions in a package called **MyPack**, use

  -ea:MyPack

- To disable assertions in **MyPack**, use

  -da:MyPack

Static Import

- JDK 5 added a new feature to Java called *static import* that 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

Example:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
// Compute the hypotenuse of a right triangle.
class Hypot {
        public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        // Here, sqrt() and pow() can be called by themselves,
        // without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));
        System.out.println("Given sides of lengths " + side1 + " and " + side2 + " the
        hypotenuse is " + hypot);
        }
}
```

- There are two general forms of the **import static** statement.
- The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

  import static *pkg.type-name.static-member-name*;
- The second form of static import imports all static members of a given class or interface. Its general form is shown here:

  import static *pkg.type-name*.*;

For example,

- this brings the static field **System.out** into view:

  import static java.lang.System.out;
- After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
                    out.println("Welcome to ECE and Mechanical students");
```

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*);

```
class MyClass {
          int a;
          int b;
          // initialize a and b individually
          MyClass(int i, int j) {
                    a = i;
                    b = j;
          }
          // initialize a and b to the same value
          MyClass(int i) {
                    this(i, i); // invokes MyClass(i, i)
          }
          // give a and b default values of 0
          MyClass( ) {
                    this(0); // invokes MyClass(0)
          }
}
```

- One reason why invoking overloaded constructors through **this( )** can be useful is that it can prevent the unnecessary duplication of code.
- In many cases, reducing duplicate code decreases the time it takes to load your class because often the object code is smaller.
- Constructors that call **this( )** will execute a bit slower than those that contain all of their initialization code inline.
- There are two restrictions you need to keep in mind when using **this( )**.

- First, you cannot use any instance variable of the constructor's class in a call to **this()**.
- Second, you cannot use **super( )** and **this( )** in the same constructor because each must be the first statement in the constructor.

String Handling

The String Constructors

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

  String s = new String();

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

  String(char *chars*[ ])

Example:

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

This constructor initializes **s** with the string “man”.

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

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

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

Example:

```
char chars[] = { 'h', 'e', 'l', 'l', 'o' };
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **llo**.

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

  String(String *strObj*)

Example:

```
class MakeString {
        public static void main(String args[]) {
                char c[] = {'r', 'a', 'm', 'a'};
                String s1 = new String(c);
                String s2 = new String(s1);
```

```
		System.out.println(s1);
		System.out.println(s2);
	}
}
```

Output:

```
rama
rama
```

- 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.
- The byte formats are Their forms are shown here:

  String(byte *asciiChars*[ ])

  String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

Example:

```
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 Added by J2SE 5

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

  String(int *codePoints*[ ], int *startIndex*, int *numChars);*

- The second new constructor supports the new **StringBuilder** class. It is shown here:

String(StringBuilder *strBuildObj*)

String Length

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

  int length( );

Example:

```
char chars[] = {‘k’, ‘r’, ‘i’, ‘s’, ‘h’};
String s = new String(chars);
System.out.println(s.length());// displays 5
```

Special String Operations

String Literals

- To explicitly create a **String** instance from an array of characters by using the **new** operator. an easier way to do this using a string literal.

Example:

```
char chars[] = {‘k’, ‘r’, ‘i’, ‘s’, ‘h’};
String s1 = new String(chars);
String s2 = "krish"; // use string literal
```

String Concatenation

- Java does not allow operators to be applied to **String** objects.
- The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object

Example:

```
String marks = "35";
String s = "He scored " + marks+ " in internals.";
System.out.println(s);// He scored 35 in internals
```

String Concatenation with Other Data Types

- You can concatenate strings with other types of data.

Example1:

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

System.out.println(s);

This fragment displays

four: 22

Example2:

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

Now **s** contains the string “four: 4”.

String Conversion and toString( )

- Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**
- Every class implements **toString( )** because it is defined by **Object**
- The **toString( )** method has this general form:

  String toString( )

Example:

```
class Box {
        double  width;
        double height;
        double depth;
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        public String toString() {
                return "Dimensions are " + width + " by " + depth + " by " + height + ".";
        }
}
class toStringDemo {
        public static void main(String args[]) {
                Box b = new Box(10, 12, 14);
                String s = "Box b: " + b; // concatenate Box object
                System.out.println(b); // convert Box to string
                System.out.println(s);
        }
}
```

Output:

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

Character Extraction

charAt( ):

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

  char charAt(int *where*);

Example:

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

getChars( )

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

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

Example:

```
String s = "Abhimanyu is a good cricket player";
int start = 16;
int end = 20;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
```

Output:

```
good
```

getBytes( )

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

  byte[ ] getBytes( )

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( )

String Comparison

equals( ) and equalsIgnoreCase( )

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

  boolean equals(Object *str*)

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

  boolean equalsIgnoreCase(String *str*)

Example:

```
class equalsDemo {
public static void main(String args[]) {
String s1 = "Sourabh";
String s2 = "Sourabh";
String s3 = "Nikil";
String s4 = "SOURABH";
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));
}
}
```

regionMatches( )

- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string
- Here are the general forms for these two methods:

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

  boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object.
- The 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.

startsWith( ) and endsWith( )

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

  boolean startsWith(String *str*)

  boolean endsWith(String *str*)

Example:

"Nandasagar".endsWith("gar")

and

"Nandasagar".startsWith("Nan")

are both **true**.

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

  boolean startsWith(String *str*, int *startIndex*);

"Nandasagar".startsWith("gar", 7) returns **true**.

equals( ) Versus ==

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

Example:

```
String s1 = "Sunaina";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));// true
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2)); // false
```

compareTo( )

- To know which is *less than, equal to,* or *greater than **String** method **compareTo( )** serves this purpose. It has this 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.

Example : Bubble sort

```
class SortString {
		static String arr[] = {
		“abi”, “nanda”, “gautham”, “saurab”, “akshay”, “suhas”
		};
		public static void main(String args[]) {
			for(int j = 0; j < arr.length; j++) {
			for(int i = j + 1; i < arr.length; i++) {
```

```
                    if(arr[i].compareTo(arr[j]) < 0) {
                        String t = arr[j];
                        arr[j] = arr[i];
                        arr[i] = t;
                        }
                    }
                    System.out.println(arr[j]);
                }
            }
}
```

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

  int compareToIgnoreCase(String *str*)

Searching Strings

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or substring.
- The methods return the index at which the character or substring was found, or –1 on failure.
- To search for the first occurrence of a character, use

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

  int lastIndexOf(int *ch*)
- To search for the first or last occurrence of a substring, use

  int indexOf(String *str*)

  int lastIndexOf(String *str*)
- You can specify a starting point for the search using these forms:

  int indexOf(int *ch*, int *startIndex*)

  int lastIndexOf(int *ch*, int *startIndex*)

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

Modifying a String

substring( )

- You can extract a substring using **substring( )**. It has two forms.
- The first is String substring(int *startIndex*)
- The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

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

Example:

```
class StringReplace {
        public static void main(String args[]) {
                String org = "This is a test. This is, too.";
                String search = "is";
                String sub = "was";
                String result = "";
                int i;
                do { // replace all matching substrings
                System.out.println(org);
                i = org.indexOf(search);
                if(i != -1) {
                        result = org.substring(0, i);
                        result = result + sub;
                        result = result + org.substring(i + search.length());
                        org = result;
                        }
                } while(i != -1);
        }
}
```

The output from this program is shown here:

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

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

concat( )

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

  String concat(String *str*)

For example,

String s1 = "Nanda";

String s2 = s1.concat("sagar");

puts the string "Nandasagar" into **s2**. It generates the same result as the following sequence:

String s1 = "Nanda";

String s2 = s1 + "sagar";

replace( )

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

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

- Here, *original* specifies the character to be replaced by the character specified by *replacement*.
- The resulting string is returned. For example,

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

  puts the string "Hewwo" into **s**.

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

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

trim( )

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

  String trim( )

Example:

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

This puts the string “Hello World” into **s**.

Data Conversion Using valueOf( )

- The **valueOf( )** method converts data from its internal format into a human-readable form 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*[ ])

- There is a special version of **valueOf( )** that allows you to specify a subset of a **char** array. It has this general form:

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

Changing the Case of Characters Within a String

- The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase.
- The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase
- General forms of these methods:

  String toLowerCase( )

  String toUpperCase( )

Example:

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

Output:

```
Origianl: Lord Krishna
Uppercase: LORD KRISHNA
Lowercase: lord krishna
```

Additional String Methods

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

StringBuffer

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth

StringBuffer Constructors

- **StringBuffer** defines these four constructors:
- **StringBuffer( ):** The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- **StringBuffer(int *size*):** accepts an integer argument that explicitly sets the size of the buffer.
- **StringBuffer(String *str*):** accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation
- **StringBuffer(CharSequence *chars*):** creates an object that contains the character sequence contained in *chars.*

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.
- They have the following general forms:

```
int length( )
int capacity( )
```

```
class StringBufferDemo {
	public static void main(String args[]) {
		StringBuffer sb = new StringBuffer("Krishna");
		System.out.println("buffer = " + sb); // displays Krishna
		System.out.println("length = " + sb.length());// displays 7
		System.out.println("capacity = " + sb.capacity());// displays 23
```

```
        }
}
```

ensureCapacity( )

- If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer **ensureCapacity( )** has this general form:

  void ensureCapacity(int *capacity*)

  Here, *capacity* specifies the size of the buffer

setLength( )

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

  void setLength(int *len*)

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*)

Example:

StringBuffer sb = new StringBuffer("Krishna");

sb.charAt(2)// selects i

sb.setCharAt(2, ‘u’)// sets second index value to u means krushna

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.
- The array that will receive the characters is specified by *target*.

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*)

Example:

```
class appendDemo {
        public static void main(String args[]) {
                String s;
                int a = 42;
                StringBuffer sb = new StringBuffer(40);
                s = sb.append("a = ").append(a).append("!").toString();
                System.out.println(s);
        }
}
```

The output of this example is shown here:

```
a = 42!
```

- The **append( )** method is most often called when the + operator is used on **String** objects.
- Java automatically changes modifications to a **String** instance into similar operations on a **StringBuffer** instance.
- There are many optimizations that the Java run time can make knowing that **String** objects are immutable

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. These are a few of its forms:

  StringBuffer insert(int *index*, String *str*)

StringBuffer insert(int *index*, char *ch*)

StringBuffer insert(int *index*, Object *obj*)

Example:

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

The output of this example is shown here:

```
I like Java!
```

reverse( )

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

  StringBuffer reverse( )

Example:

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

Here is the output produced by the program:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

- 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*)

StringBuffer deleteCharAt(int *loc*)

Example:

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

The following output is produced:

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

replace( )

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

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

Example:

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

Here is the output:

```
After replace: This was a test.
```

substring( )

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

  String substring(int *startIndex*)

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

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

Additional StringBuffer Methods

Method	**Description**
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **StringBuffer**.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

Example:

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

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

The output is shown here:

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

StringBuilder

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