

STRING HANDLING, INHERITANCE AND INTERFACES:

STRING HANDLING:

Exploring the String Class:

- **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type **String**.
- Even string constants are actually **String** objects. For example, in the statement.....

System.out.println("This is a String, too");

- The string "This is a String, too" is a **String** constant.
- The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered.
- While this may seem like a serious restriction, it is not, for two reasons:
 - If you need to change a string, you can always create a new one that contains the modifications.
 - Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java.
- Strings can be constructed in a variety of ways. The easiest is to use a statement like this:
String myString = "this is a test";
- Once you have created a **String** object, you can use it anywhere that a string is allowed.
- For example, this statement displays **myString**: System.out.println(myString);
- Java defines one operator for **String** objects: +. It is used to concatenate two strings.
- For example, this statement..... results in **myString** containing "I like Java."
String myString = "I" + " like " + "Java.";

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
class StringDemo {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1 + " and " + strOb2;

        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```

- The **String** class contains several methods that you can use. Here are a few along with their general form
 - We can test two strings for equality by using **equals()**. boolean equals(String object);
 - We can obtain the length of a string by calling the **length()** method... int length();
 - We can obtain the character at a specified index within a string by calling **charAt()**.
char charAt(int index);
- Here is a program that demonstrates these methods...

```
// Demonstrate String arrays.
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;

        System.out.println("Length of strOb1: " +
                           strOb1.length());

        System.out.println("Char at index 3 in strOb1: " +
                           strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

- Of course, we can have arrays of strings, just like you can have arrays of any other type of object. For example...

```
// Demonstrate String arrays.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "one", "two", "three" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " +
                               str[i]);
    }
}
```

-
- The **String**, **StringBuffer**, and **StringBuilder** classes are defined in `java.lang`. Thus, they are available to all programs automatically.
 - All are declared **final**, which means that none of these classes may be subclassed.
 - This allows certain optimizations that increase performance to take place on common string operations.
 - All three implement the **CharSequence** interface.
 - One last point:
 - To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created.
 - However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors:

- The **String** class supports several constructors. To create an empty **String**, you call the default constructor.
- For e.g., **String s = new String();** will create an instance of **String** with no characters in it.
- Frequently, you will want to create strings that have initial values. To create a **String** initialized by an array of characters, use the constructor shown here...
 - **String(char chars[]);**
 - `char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);`
 - This constructor initializes **s** with the string “abc”
- We can specify a subrange of a character array as an initializer using the following constructor.
 - **String(char chars[], int startIndex, int numChars);**
 - Here, `startIndex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use. Here is an example:
 - `char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }; String s = new String(chars, 2, 3); // This initializes s with the characters cde.`
- We can construct a **String** object that contains the same character sequence as another **String** object using this constructor (Copy constructor).....
 - **String(String strObj);** //Here, **strObj** is a **String** object.
- 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.

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

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

The output from this program is as follows:
Java
Java
```

- Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array.
- Their forms are shown here:
String(byte asciiChars[]);
String(byte asciiChars[], int startIndex, int numChars);
 - Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange.
- In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.
- The following program illustrates these constructors...

```
//Constructing String from SubSequence of byte array
class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70};

        String s1 = new String(ascii);
        System.out.println(s1);

        String s2 = new String(ascii, 2, 3);
        System.out.println(s2);
    }
}
```

Special String Operations:

- Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- These operations include...
 - The automatic creation of new String instances from string literals,

-
- Concatenation of multiple String objects by use of the + operator, and
 - The conversion of other data types to a string representation.
 - There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

❖ String Literals:

- The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator.
- However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object.
- Thus, we can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings...

```
char chars[] = { 'a', 'b', 'c' };           String s1 = new String(chars);  
String s2 = "abc"; // use string literal.
```

- Because a **String** object is created for every string literal, we can use a string literal any place you can use a **String** object.
- For example, we can call methods directly on a quoted string as if it were an object reference, as the following statement shows.
- It calls the **length()** method on the string “abc”. As expected, it prints “3”.
- `System.out.println("abc".length());`

❖ String Concatenation:

- In general, Java does not allow operators to be applied to **String** objects.
- The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result.
- We can chain together a series of + operations.

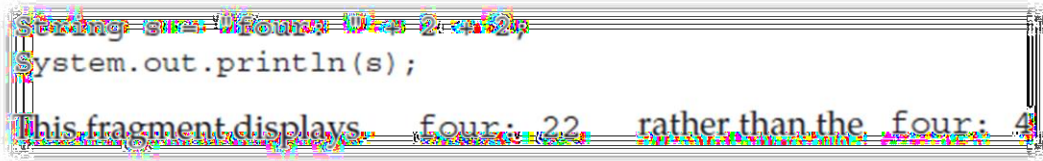
❖ String Concatenation with Other Data Types:

- We can concatenate strings with other types of data. For e.g., consider this slightly different version of the earlier example...

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

- In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before.
- This is because the **int** value in **age** is automatically converted into its string representation within a **String** object.

-
- The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of **String**.
 - Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following...



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

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

- Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first.
- This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:
- `String s = "four: " + (2 + 2);` Now `s` contains the string "four: 4".

❖ **String Conversion and toString():**

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method `valueOf()` defined by `String`.
- `valueOf()` is overloaded for all the simple types and for type `Object`.
 - For the simple types, `valueOf()` returns a string that contains the human-readable equivalent of the value with which it is called.
 - For objects, `valueOf()` calls the `toString()` method on the object.
- Every class implements `toString()` because it is defined by `Object`. However, the default implementation of `toString()` is seldom sufficient.
- The `toString()` method has this general form: `String toString();`
- By overriding `toString()` for classes that you create, you allow them to be fully integrated into Java's programming environment.
- For example, they can be used in `print()` and `println()` statements and in concatenation expressions.
- The following program demonstrates this by overriding `toString()` for the `Box` class.

```
// Override toString() for Box class.
```

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

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

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

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

        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}
```

The output of this program is shown here:

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

Some more important methods:

❖ **getChars():**

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

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

- Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.
- Thus, the substring 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.

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

        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Here is the output of this program: demo

❖ **getBytes()**:

- There is an alternative to **getChars()** that stores the characters in an array of bytes.
- This method is called **getBytes()**, and it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form: `byte[] getBytes()` Other forms of **getBytes()** are also available.
- **getBytes()** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters.
- For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

❖ **toCharArray():**

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

char[] toCharArray();

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

❖ `equals()` and `equalsIgnoreCase()`;

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

boolean equalsIgnoreCase(String str);

❖ **startsWith()** and **endsWith()**

- The **startsWith()** method determines whether a given **String** begins with a specified string.
- Conversely, **endsWith()** determines whether the **String** in question ends with a specified string. They have the following general forms.
- boolean startsWith(String str); boolean endsWith(String str);**

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + " equals " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " +
                           s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " +
                           s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                           s1.equalsIgnoreCase(s4));
    }
}
```

The output from the program is shown here:

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

- Here, str is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For e.g.,:

- "Foobar".endsWith("bar") and "Foobar".startsWith("Foo") are both **true**.

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

boolean startsWith(String str, int startIndex);

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

❖ **equals()** Versus **==**

- It is important to understand that the **equals()** method and the **==** operator perform two different operations.
 - As just explained..., 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.

❖ **compareTo()**:

- Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next.
 - A string is less than another if it comes before the other in dictionary order.

```

class EqualsNotEqualTo {

    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);

        System.out.println(s1 + " equals " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}

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

```

- A string is greater than another if it comes after the other in dictionary order. The `String` method `compareTo()` serves this purpose. It has this general form:

`int compareTo(String str);`

- Here, `str` is the **String** being compared with the invoking **String**.
- The result of the comparison is returned and is interpreted, as shown here...

Value	Meaning
Less than zero	The invoking string is less than <i>str</i> .
Greater than zero	The invoking string is greater than <i>str</i> .
Zero	The two strings are equal.

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

```

// A bubble sort for Strings.
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };
    public static void main(String args[]) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
                if(arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            }
            System.out.println(arr[j]);
        }
    }
}

```

Searching Strings:

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
 - **indexOf()** Searches for the first occurrence of a character or substring.
 - **lastIndexOf()** Searches for the last occurrence of a character or substring.
- These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or -1 on failure.
 - To search for the first occurrence of a character, use **int indexOf(int ch)**;
 - To search for the last occurrence of a character, use **int lastIndexOf(int ch)**; Here, ch is the character being sought.
- To search for the first or last occurrence of a substring, Here, str specifies the substring...

int indexOf(String str); int lastIndexOf(String str);

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
    public static void main(String args[]) {
        String s = "Now is the time for all good men " +
                   "to come to the aid of their country.";

        System.out.println(s);
        System.out.println("indexOf(t) = " +
                           s.indexOf('t'));
        System.out.println("lastIndexOf(t) = " +
                           s.lastIndexOf('t'));
        System.out.println("indexOf(the) = " +
                           s.indexOf("the"));
        System.out.println("lastIndexOf(the) = " +
                           s.lastIndexOf("the"));
        System.out.println("indexOf(t, 10) = " +
                           s.indexOf('t', 10));
        System.out.println("lastIndexOf(t, 60) = " +
                           s.lastIndexOf('t', 60));
        System.out.println("indexOf(the, 10) = " +
                           s.indexOf("the", 10));
        System.out.println("lastIndexOf(the, 60) = " +
                           s.lastIndexOf("the", 60));
    }
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String:

- Because **String** objects are immutable, whenever you want to modify a **String**,
 - You must either copy it into a StringBuffer or StringBuilder, or
 - Use one of the following String methods, which will construct a new copy of the string with your modifications complete.

❖ **substring()**:

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

String substring(int startIndex);

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

String substring(int startIndex, int endIndex);

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

```
// Substring replacement.
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;

        do { // replace all matching substrings
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
                result = result + org.substring(i + search.length());
                org = result;
            }
        } while(i != -1);
    }
}
```

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);**
- This method creates a new object that contains the invoking string with the contents of str appended to the end.
- **concat()** performs the same function as +. For example,
- String s1 = "one"; String s2 = s1.concat("two"); puts the string “onetwo” into s2.
- It generates the same result as the following sequence: String s1 = "one"; String s2 = s1 + "two";

❖ **replace()**

- This 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. The resulting string is returned. For example...
- String s = "Hello".replace('l', 'w'); puts the string “Hewwo” into s.

❖ **trim()**

- The **trim()** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form: **String trim();**
- Here is an example: String s = " Hello World ".trim(); This puts the string “Hello World” into s.

StringBuffer:

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- As you know, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- Java uses both classes heavily, but many programmers deal only with String and let Java manipulate StringBuffers behind the scenes by using the overloaded + operator.

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

❖ **length() and capacity():**

- The current length of a StringBuffer can be found via the length() method, while the total allocated capacity can be found through the capacity() method.
- They have the following general forms: **int length();** **int capacity();**

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Here is the output
buffer = Hello
length = 5
capacity = 21

❖ **ensureCapacity()**

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

❖ **setLength():**

- To set the length of the buffer within a StringBuffer object, use setLength(). Its general form is shown here:
void setLength(int len);
- Here, len specifies the length of the buffer. This value must be nonnegative.

❖ **charAt() and setCharAt():**

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

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

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

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

Here is the output generated by this program:

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

❖ **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()**. This allows subsequent calls to be chained together, as shown in the following example...


```
// Demonstrate append()
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

The output of this example is shown here:

a = 42!

❖ insert()

- The insert() method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- Like `append()`, it calls `String.valueOf()` to obtain the string representation of the value it is called with.
- This string is then inserted into the invoking `StringBuffer` object. These are a few of its forms...
 - **`StringBuffer insert(int index, String str);`**
 - **`StringBuffer insert(int index, char ch);`**
 - **`StringBuffer insert(int index, Object obj);`**
- Here, index specifies the index at which point the string will be inserted into the invoking `StringBuffer` object.

```
// Demonstrate insert()
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("I Java!");

        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

The output of this example is shown here:

I like Java!

❖ **reverse():**

- You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:
`StringBuffer reverse()`;
- This method returns the reversed object on which it was called. The following program demonstrates **reverse()**:

```
// Using reverse() to reverse a StringBuffer:
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");

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

❖ **delete() and deleteCharAt():**

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

StringBuffer delete(int startIndex, int endIndex);

StringBuffer deleteCharAt(int loc);

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

```
// Demonstrate delete() and deleteCharAt()
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.

StringBuilder:

- J2SE5 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, we must use `StringBuffer` rather than `StringBuilder`.

Inheritance:

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines characters common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
 - In the terminology of Java, a class that is inherited is called a superclass.
 - The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance Basics:

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.
- The general form of a **class** declaration that inherits a superclass is shown here...

```
class subclass-name extends superclass-name {  
    // body of class  
}
```

```
A simple example of inheritance.

Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}

Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }

    void sum() {
        System.out.println("i+j+k: " + (i+j+k));
    }
}

class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members
           its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

- You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass.
- You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance:

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example...
 - This program will not compile because the reference to **j** inside the **sum()** method of **B** causes an access violation.
 - Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- Each subclass can precisely tailor its own classification.

```
/* In a class hierarchy, private members remain
   private to their class.
```

```
   This program contains an error and will not
   compile.
```

```
*/
```

```
// Create a superclass.
```

```
class A {
    int i; // public by default
    private int j; // private to A
```

```
    void setij(int x, int y) {
        i = x;
        j = y;
    }
}
```

```
// A's j is not accessible here.
```

```
class B extends A {
    int total;
    void sum() {
        total = i + j; // ERROR, j is not accessible here
    }
}
```

```
class Access {
    public static void main(String args[]) {
        B subOb = new B();

        subOb.setij(10, 12);

        subOb.sum();
        System.out.println("Total is " + subOb.total);
    }
}
```

A Superclass Variable Can Reference a Subclass Object:

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following...
 - Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects.
 - Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
 - That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
 - This is why plainbox can't access weight even when it refers to a BoxWeight object.
 - If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out.

```

class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +
                           weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox
           does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}

```

- It is not possible for a Box reference to access the weight field, because Box does not define one.

Using super:

- In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been.
 - For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box()**.
- Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members.
- However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).
- In this case, there would be no way for a subclass to directly access or initialize these variables on its own.
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. **super** has two general forms...
 - The first calls the superclass constructor.
 - The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors:

- A subclass can call a constructor defined by its superclass by use of the following form of **super...**
super(arg-list);
- Here, arg-list specifies any arguments needed by the constructor in the superclass.
- **super()** must always be the first statement executed inside a subclass' constructor.

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
    double weight; // weight of box

    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

A Second Use for super:

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:
super.member
- Here, member can be either a method or an instance variable.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- Consider this simple class hierarchy.....


```

// Using super to overcome name hiding.
class A {
    int i;
}

// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A

    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

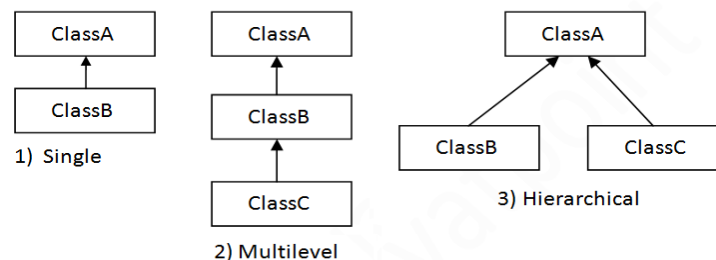
class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}

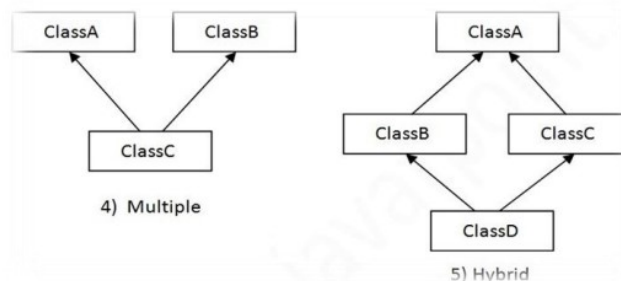
```

Types of inheritance in java:

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.



- Note: Multiple inheritance is not supported in java through class.



- **Why multiple inheritance is not supported in java?**

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes.
- If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes.
- So whether you have same method or different, there will be compile time error now.

```
class A{
    void msg(){System.out.println("Hello");}
}
class B{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
    Public Static void main(String args[]){
        C obj=new C();
        obj.msg();//Now which msg() method would be invoked?
    }
}
```

When Constructors Are Called:

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?
- For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa?
- The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used.
- If **super()** is not used, then the default or parameter-less constructor of each superclass will be executed.
- The following program illustrates when constructors are executed.

```
// Demonstrate when constructors are called.

// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}

// Create a subclass by extending class A.
class B extends A {
    B() {
        System.out.println("Inside B's constructor.");
    }
}

// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}

class CallingCons {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

- If you think about it, it makes sense that constructors are executed in order of derivation.
 - Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Method Overriding:

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden. Consider the following.....
 - When show() is invoked on an object of type B, the version of show() defined within B is used.
 - That is, the version of show() inside B overrides the version declared in A.
- If you wish to access the superclass version of an overridden method, you can do so by using super

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}
```

```
class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}
```

The output produced by this program is shown here:

k: 3

- For example:
 - In this version of B, the superclass version of show() is invoked within the subclass' version. This allows all instance variables to be displayed.

<pre>class B extends A { int k; B(int a, int b, int c) { super(a, b); k = c; } }</pre>	<pre>void show() { super.show(); // this calls A's show() System.out.println("k: " + k); }</pre>
---	--

- Method overriding occurs only when the names and the type signatures of the two methods are identical.
- If they are not, then the two methods are simply overloaded. For e.g., consider this modified version of the preceding example.

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}
```

```
class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}
```

The output produced by this program is shown here:

k: 3

- The version of **show()** in **B** takes a **String** parameter. This makes its type signature different from the one in **A**, which takes no parameters.
- Therefore, no overriding (or name hiding) takes place. Instead, the version of **show()** in **B** simply overloads the version of **show()** in **A**.

Dynamic Method Dispatch:

- While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power.
- Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value.
- However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

- Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object.
 - Java uses this fact to resolve calls to overridden methods at run time.
 - Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
 - Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
 - In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
 - Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. For e.g.,

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}

class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}
```

```
class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
        A r; // obtain a reference of type A
        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme

        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme

        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

-
- This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme()** declared in **A**.
 - Inside the **main()** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.
 - The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**.
 - As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call.
 - Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme()** method.

Why Overridden Methods?

- As stated earlier, overridden methods allow Java to support run-time polymorphism.
 - Polymorphism is essential to object-oriented programming for one reason.....
 - It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
 - Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
 - Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
 - The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.
- Let's look at an example that uses method overriding.
 - The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object.
 - It also defines a method called **area()** that computes the area of an object.
 - The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**.
 - Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively


```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Area is undefined.");
        return 0;
    }
}
```

```
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
```

```
class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure figref;

        figref = r;
        System.out.println("Area is " + figref.area());

        figref = t;
        System.out.println("Area is " + figref.area());

        figref = f;
        System.out.println("Area is " + figref.area());
    }
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Using Abstract Classes:

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

-
- Such a class determines the nature of the methods that the subclasses must implement.
 - One way this situation can occur is when a super class is unable to create a meaningful implementation for a method.
 - This is the case with the class Figure used in the preceding example.
 - The definition of area() is simply a placeholder. It will not compute and display the area of any type of object.
 - As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass.
 - You can handle this situation two ways...
 - One way, as shown in the previous example, is to simply have it report a warning message.
 - While this approach can be useful in certain situations—such as debugging—it is not usually appropriate.
 - You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.
 - Consider the class **Triangle**. It has no meaning if **area()** is not defined.
 - In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.
 - You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.
 - These methods are sometimes referred to as subclasses responsibility because they have no implementation specified in the superclass.
 - Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
 - To declare an abstract method, use this general form...

abstract type name (parameter-list);

- As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, **an abstract class cannot be directly instantiated with the new operator.**
- Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

-
- Here is a simple example of a class with an abstract method, followed by a class which implements that method.....

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();

    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}

class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

- Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class.
- One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable.
- Abstract classes can include as much implementation as they see fit.
- Abstract classes can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.
- You will see this feature put to use in the next example.....
 - As the comment inside **main()** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract.
 - And, all subclasses of **Figure** must override **area()**.
 - To prove this to yourself, try creating a subclass that does not override **area()**. You will receive a compile-time error.
 - Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**.

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created

        figref = r;
        System.out.println("Area is " + figref.area());

        figref = t;
        System.out.println("Area is " + figref.area());
    }
}
```

- The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**.

Using final with Inheritance:

- The keyword **final** has three uses.
- First, it can be used to create the equivalent of a named constant. The other two uses of **final** apply to inheritance. Both are examined here.
- ❖ **Using final to Prevent Overriding:**
 - While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.
 - To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
 - Methods declared as **final** cannot be overridden. The following fragment illustrates **final**...

```

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

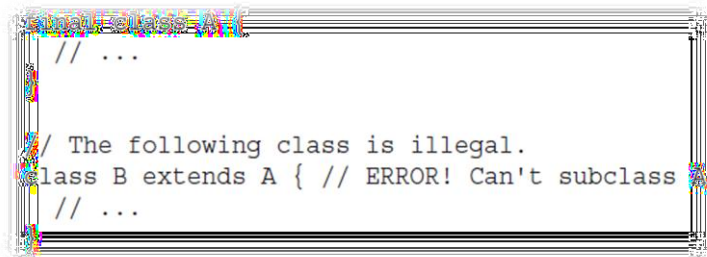
class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

- Methods declared as **final** can sometimes provide a performance enhancement:
 - The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass.
 - When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
 - Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at runtime. This is called late binding.
 - However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

❖ Using final to Prevent Inheritance:

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Here is an example of a final class...



```

// ...

// The following class is illegal.
class B extends A { // ERROR! Can't subclass
    // ...
}

```

- As the comments imply, it is illegal for B to inherit A since A is declared as final.

Interfaces:

- Using the keyword **interface**, you can fully abstract a class interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
 - Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.
 - Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.
- Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy.
- Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.

Defining an Interface:

- An interface is defined much like a class. This is the general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    // ...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.

-
- When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
 - name is the name of the interface, and can be any valid identifier.
 - Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list.
 - They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.
 - Variables can be declared inside of interface declarations.
 - They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.
 - Here is an example of an interface definition.
 - It declares a simple interface that contains one method called **callback()** that takes a single integer parameter.

```
interface Callback {  
    void callback(int param);  
}
```

Implementing Interfaces:

- Once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the implements clause looks like this.

```
class classname [extends superclass] [implements interface [,interface...]] {  
    // class-body  
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public.
- Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.
- Here is a small example class that implements the Callback interface shown earlier.

```
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {

        System.out.println("callback called with " + p);
    }
}
```

- Notice that callback() is declared using the public access specifier.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of Client implements callback() and adds the method nonIfaceMeth().

```
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }

    void nonIfaceMeth() {
        System.out.println("Classes that implement interfaces " +
            "may also define other members, too.");
    }
}
```

Accessing Implementations through Interface References:

- We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the “callee.”
- This process is similar to using a superclass reference to access a subclass object.

```

class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}

```

The output of this program is shown here:

```

callback called with 42

```

Partial Implementations:

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- For example.....
 - Here, the class Incomplete does not implement callback() and must be declared as abstract. Any class that inherits Incomplete must implement callback() or be declared abstract itself.

```

abstract class Incomplete implements Callback {
    int a, b;
    void show() {
        System.out.println(a + " " + b);
    }
    // ...
}

```

Variables in Interfaces:

- We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants.

```

import java.util.Random;
interface SharedConstants {
    int NO = 0; int YES = 1;
    int MAYBE = 2; int LATER = 3;
    int SOON = 4; int NEVER = 5;
}

```

```

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)    return NO;
        else if (prob < 60)    return YES;
        else if (prob < 75)    return LATER;
        else if (prob < 98)    return SOON;
        else    return NEVER;
    }
}

```

Interfaces Can Be Extended:

- One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.
- Following is an example...
 - As an experiment, you might want to try removing the implementation for **meth1()** in **MyClass**. This will cause a compile-time error.
 - As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

```

// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
    void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }
    public void meth2() {
        System.out.println("Implement meth2().");
    }
    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

```

```

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}

```

- Virtually all real programs that you write in Java will be contained within packages.
- A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

Multiple inheritance is achieved via default methods:

- Multiple inheritance is a feature in which an object or class can inherit characteristics and behavior from more than one parent object or parent class.
- We know that in java (until jdk 7), inheritance in java was supported by extends keyword which is used to create a child class from a parent class. You cannot extend from two classes.
- Until java 7, interfaces were only for declaring the contracts which implementing classes MUST implement. So there was no specific behavior attached with interfaces which a class can inherit.
- So, even after a class was capable of implementing as many interfaces as it want, it was not appropriate to term as multiple inheritance.
- But since java 8's default methods, interfaces have behavior as well.
- So now if a class implement two interfaces and both defines default methods, then it is essentially inheriting behaviors from two parents which is multiple inheritance.
- For example, in below code Animal class does not define any of it's own behavior; rather it is inheriting behavior from parent interfaces.

```
class Batsman {
    String name;
    int noOfRuns;
    int noOfMatches;

    Batsman(String s, int runs, int match) {
        name=s;
        noOfRuns=runs;
        noOfMatches=match;
    }

    void display() {
        System.out.println("Name:      "+ name);
        System.out.println("No. of Runs:  "+ noOfRuns);
        System.out.println("No. of Matches: "+ noOfMatch);
    }
}

interface Bowler {
    public int getwickets();
    public int getNoOfCaughtwickets();
}
```

```
class AllRounder extends Batsman implements Bowler {
    int NoOfwickets;
    int NoOfCwickets;

    AllRounder(.....){
        super(.....);
    }

    public int getwickets() {
        return NoOfwickets;
    }
    public int getNoOfCaughtwickets() {
        return NoOfCwickets;
    }
}
```

```
interface Moveable
{
    default void moveFast(){
        System.out.println("I am moving fast, buddy !!");
    }
}

interface Crawlable
{
    default void crawl(){
        System.out.println("I am crawling !!");
    }
}

public class Animal implements Moveable, Crawlable
{
    public static void main(String[] args)
    {
        Animal self = new Animal();

        self.moveFast();
        self.crawl();
    }
}
```
