example of package and access modifiers, String class and its methods
String class methods(cont.), StringBuffer, StringBuilder, Abstract class
Interface (data member, method) implements and extends keyword, examples, varargs
package example from lab manual, Wrapper classes, autoboxing, unboxing

String
In Java, string is not primitive type or array of characters. String can be used to declared string variables. Quoted string constant can be assigned to string variable.

How to create a String object??

1. By string literal

e.g. String s = “DDIT”;
Also declaration and initialization can be in different statements.

```
String s;
s=”DDIT”;
String t = “DDIT”;
```

Here new object for t is not created. Then what happens?

Every time a string literal (here DDIT is a string literal) is created, JVM checks in **String constant pool**.

- If the string already exists in the string constant pool, then a reference to the pooled instance is returned.
- If the string does not exist in the pool, then new String object is instantiated and placed in the pool.

2. By new keyword

```
String s = new String (“DDIT”);
Or
String mystring=”DDIT”;
String s=new String (mystring);
Or
char[] ch = {‘D’,’D’,’I’,’T’};
String s = new String (ch);
```

In Java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable. Once string object is created its data or state can't be changed but a new string object is created.

```
String myname = "Niyati";
myname.concat("Buch");
System.out.println(myname);                    //Niyati

String fullname = myname.concat("Buch");
System.out.println(myname);                    //Niyati
System.out.println(fullname);                  //NiyatiBuch
```

```
String s1="Hello";
String s2=new String ("Hello");
String s3="Hello";
String s4=new String ("Hello");

s1==s2 false
s1==s3 true
s2==s4 false
```

s1 and s3 are reference to same object.
s1 and s2 are different objects.

String comparison

1. **equals()** returns boolean. It compares values. E.g. s1.equals(s2)

2. **==** operator returns boolean. It compares references not values.

3. **compareTo()** returns integer. E.g. s1.compareTo(s2)
   if **s1==s2** then **zero**; if **s1>s2** then **+ve** value; if **s1<s2** then **–ve** value

String Concatenation

1. **Using + operator**

```
String fn = "N", mn = "J", ln = "B";
String name = fn + mn + ln;
System.out.println(name);                        //NJB

String text = "MyText";
String newtext = text + 2;
System.out.println(newtext);                     //MyText2
String newtext1 = text + 2 + 2;
System.out.println(newtext1);                    //MyText22
String newtext2 = text + (2 + 2);
System.out.println(newtext2);                    //MyText4
```

2. **Using public String concat(String) method** (example on previous page)

Note: public String toString() method of Object class can/should be overridden in any class.

Popular methods:
1. char charAt(int where)
2. char[ ] toCharArray( )
3. boolean startsWith(String str)
4. boolean endsWith(String str)
5. int indexOf(int ch)
6. int lastIndexOf(int ch)
7. int indexOf(String str)
8. int lastIndexOf(String str)

9. String trim( )
10. String substring(int startIndex)
11. String substring(int startIndex, int endIndex)
    startIndex->0 inclusive and endIndex->1 exclusive
12. String replace(char original, char replacement)
13. String replace(CharSequence original, CharSequence replacement)
    [CharSequence is an interface which is implemented by String, StringBuilder and StringBuffer classes]
14. String replaceAll(String regExp, String newStr)

```
public class **StringReplace** {
        public static void main(String args[]) {
                String org = "This is a test. This is, too.";
                String search = "is";
                String sub = "was";
                String org = org.replaceAll("is","was");
                System.out.println(org)  ;
        }
}
```

15. static String valueOf(boolean b) true or false
16. static String valueOf(char *ch*)
17. static String valueOf(char *chars*[ ])
18. static String valueOf(int *num*)
19. static String valueOf(double *num*)
20. static String valueOf(float *num*)
21. static String valueOf(long *num*)
22. static String valueOf(Object *ob*)
23. String toLowerCase( )
24. String toUpperCase( )
25. static String join(CharSequence delim, CharSequence . . . strs)

```
class **StringJoin**Demo {
        public static void main(String args[]) {
                String result = String.join(" ", "Alpha", "Beta", "Gamma");
                System.out.println(result);
                result = String.join(", ", "ABC", "DEF","PQR XYZ");
        //the delimiter need not be just a single character. Here, comma followed by space.
                System.out.println(result);
        }
}
```

Output:

```
Alpha Beta Gamma
ABC, DEF, PQR XYZ
```

StringBuilder and **StringBuffer** classes can be used to create mutable string.

1. StringBuffer( ): reserves room for 16 characters without reallocation
2. StringBuffer(int size): explicitly sets the size of the buffer
3. StringBuffer(String str): sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation
4. StringBuffer(CharSequence chars): same as StringBuffer(String str)

The current length of a StringBuffer can be found via the **int length( )** method, while the total allocated capacity can be found through the **int capacity( )** method.
Methods like **append**, **insert**, **reverse**, **delete**, **replace** returns **reference to StringBuffer** object which may or may not be assigned to a new variable.

Example:

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

Output: **a = 42!**

Example:

```
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like "); //1st argument is index
System.out.println(sb);
```

Output: **I like Java!**

Example:

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

Output:
```
abcdef
fedcba
```

Example:

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

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

Example:

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

Output: **After replace: This was a test.**

Some more methods:
1. String substring(int startIndex)
2. String substring(int startIndex, int endIndex)
3. int indexOf(String str), int indexOf(String str, int startIndex)
4. int lastIndexOf(String str), int lastIndexOf(String str, int startIndex)

Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class.
- It can have abstract and non-abstract methods (method with the body).

Abstraction (one of the four pillars of OOP)

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

- A class which is declared as abstract is known as an abstract class.
- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

A method which is declared as abstract and does not have implementation is known as an **abstract method.**

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

```
abstract class Fruit
{
        int rate;
        Fruit(){}
        Fruit(int rate){
                this.rate = rate;
        }
        abstract void displayTaste();
        abstract void displayColor();
        void healthy()  {
                System.out.println("All fruits are healthy");
        }
}
```

```
class Mango extends Fruit
{
        Mango(){}
        Mango(int rate) {
                super(rate);
        }
        void setRate(int rate)  {
                this.rate=rate;
        }
        void displayColor()    {
                System.out.println("Mango is yellow in color");
        }
        void displayTaste()    {
                System.out.println("Mango is sweet in taste");
}       }

class Grapes extends Fruit{
        void displayColor()    {
                System.out.println("Grapes are green in color");
        }
        void displayTaste()    {
                System.out.println("Grapes are sour in taste");
        }
        void setRate(int rate)  {
                this.rate=rate;
}       }

class DemoFruit{
        public static void main(String args[]) {
                Mango m  = new Mango(80);
                //Mango m  = new Mango();
                m.displayColor();
                m.displayTaste();
                m.healthy();
                //m.setRate(80);
                System.out.println(m.rate);

                Grapes g = new Grapes();
                g.displayColor();
                g.displayTaste();
                g.healthy();
                g.setRate(100);
                System.out.println(g.rate);
}       }
```

```
Mango is yellow in color
Mango is sweet in taste
All fruits are healthy
80

Grapes are green in color
Grapes are sour in taste
All fruits are healthy
100
```

Interface in Java

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

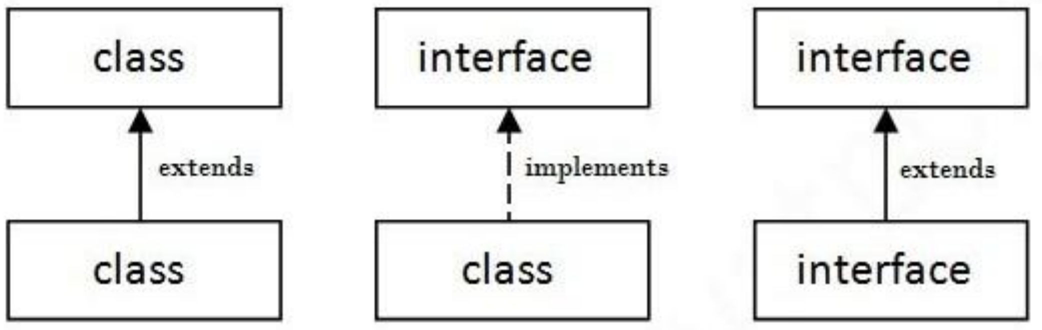
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

- An interface is declared by using the interface keyword.
- It provides **total abstraction**; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.



```
interface Characteristics{
            void locomotion();
            void eats();
}
class Fish implements Characteristics{
            public void locomotion(){
                        System.out.println("swims");
            }
            public void eats()   {
                        System.out.println("eats flakes");
            }
            public void method()   {
                        System.out.println(" in method");
}           }

class Dog implements Characteristics{
            public void locomotion(){
                        System.out.println("walks on 4 legs");
            }
            public void eats()   {
                        System.out.println("eats dog-food");
}           }

class InterfaceDemo{
            public static void main(String args[]) {
                        Fish f = new Fish();
                        f.locomotion();
                        f.eats();
                        f.method();

                        Dog d = new Dog();
                        d.locomotion();
                        d.eats();

                        Characteristics c = new Fish();
                        c.locomotion();
                        c.eats();
                        //c.method; //cannot find symbol
                        ((Fish)c).method();
}           }
```

```
swims
eats flakes
 in method

walks on 4 legs
eats dog-food

swims
eats flakes
 in method
```

Varargs

varargs is short for variable-length arguments.
A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.

```
// Use an array to pass a variable number of arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray
{
        static void vaTest(int v[]) {
                System.out.print("Number of args: " + v.length +
                " Contents: ");
                for(int x : v)
                        System.out.print(x + " ");
                System.out.println();
        }
        public static void main(String args[])
        {
                // Notice how an array must be created to hold the arguments.
                int n1[] = { 10 };
                int n2[] = { 1, 2, 3 };
                int n3[] = { };
                vaTest(n1); // 1 arg
                vaTest(n2); // 3 args
                vaTest(n3); // no args
        }
}
```

OUTPUT:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

A variable-length argument is specified by three periods (…).
For example, here is how vaTest( ) is written using a vararg: static void vaTest(int ... v) {

Remember, the varargs parameter must be last. For example, the following declaration is incorrect: int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

There is one more restriction to be aware of: there must be only one varargs parameter.
For example, this declaration is also invalid: int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

Varargs and overloading.

```
class VarArgs3 {
            static void vaTest(int ... v) {}
            static void vaTest(boolean ... v) {}
            public static void main(String args[]) {
                        vaTest(1, 2, 3);
                        vaTest(true, false, false);
}           }
```

A varargs method can also be overloaded by a non-varargs method.
For example, vaTest(int x) is a valid overload of vaTest( ) in the foregoing program.
This version is invoked only when one int argument is present.
When two or more int arguments are passed, the varargs version vaTest (int…v) is used.

vaTest(); // Error: Ambiguous!

Because the vararg parameter can be empty, this call could be translated into a call to vaTest(int …) or vaTest(boolean …). Both are equally valid. Thus, the call is inherently ambiguous.

Wrapper Classes

- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.
- Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically.
- The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.
- Double, Float, Long, Integer, Short, Byte, Character, and Boolean. – 8 wrapper classes
  - **Primitive to Wrapper**
    ```
    int  a = 20;                    // primitive
    Integer i = Integer.valueOf(a); // converting int to Integer explicitly
    Integer j = a;                  // autoboxing
    ```
  - **Wrapper to Primitive**
    ```
    Integer a = new Integer(3);     // boxing
    int i= a.intValue();            // converting Integer to int explicitly
    int j = a;                      // unboxing
    ```
  - **String to primitive**
    ```
    String s = 10;
    int n =Integer.parseInt(s);
    ```

Note:
valueOf() :- static method of Wrapper class to explicitly convert primitive to Wrapper object.
xxxValue():- non-static method of Wrapper class to explicitly convert Wrapper object to primitive
parseXxx():- static method of Wrapper class to explicitly convert String to primitive