**Topic Covered:**
The String Class and some of its methods, OOP principles, Encapsulation, Inheritance, Polymorphism, Method Overloading, Constructor Overloading, Passing and Returning Objects

## THE STRING CLASS

The String class can be used to create objects consisting of a sequence of characters. String constructors accept String literals, String objects, or no argument, which creates an empty String. The length method returns the number of characters in the String object. The toUpperCase and toLowerCase methods return a String in upper or lower case. The charAt method extracts a character from a String, while the substring method extracts a String from a String. The indexOf method searches a String for a character or substring.

| ***String* Class Constructor Summary** |
|---|
| **String( String str )**<br>allocates a *String* object with the value of *str*, which can be a *String* object or a *String* literal |
| **String( )**<br>allocates an empty *String* object |

| ***String* Class Method Summary** | |
|---|---|
| **Return value** | **Method name and argument list** |
| **int** | **length( )**<br>returns the length of the *String* |
| **String** | **toUpperCase( )**<br>converts all letters in the *String* to uppercase |
| **String** | **toLowerCase( )**<br>converts all letters in the *String* to lowercase |
| **char** | **charAt( int index )**<br>returns the character at the position specified by *index* |
| **int** | **indexOf( String searchString )**<br>returns the index of the beginning of the first occurrence of *searchString* or −1 if *searchString* is not found |
| **int** | **indexOf( char searchChar )**<br>returns the index of the first occurrence of *searchChar* in the String or −1 if *searchChar* is not found |
| **String** | **substring( int startIndex, int endIndex )**<br>returns a substring of the *String* object beginning at the character at index *startIndex* and ending at the character at index *endIndex* − *1* |

**PROGRAM 1:**

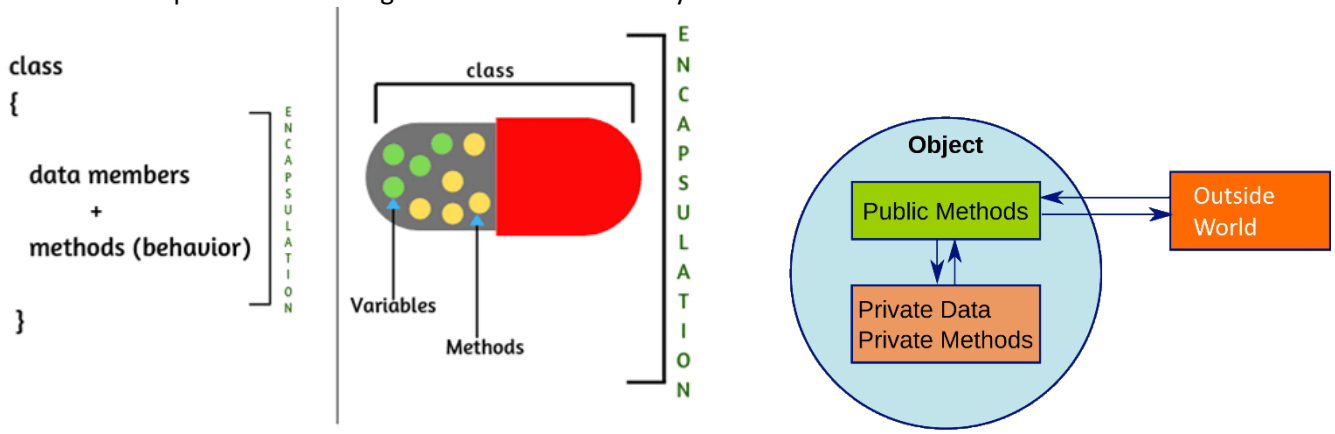
```
public class StringDemo {
public static void main ( String[] args ){

String s1 = new String( "OOP in Java " );
System.out.println( "s1 is: " + s1 );

String s2 = "is not that difficult. ";
System.out.println( "s2 is: " + s2 );

String s3 = s1 + s2; // new String is s1, followed by s2

System.out.println( "s1 + s2 returns: " + s3 );
System.out.println( "s1 is still: " + s1 ); // s1 is unchanged
System.out.println( "s2 is still: " + s2 ); // s2 is unchanged

String greeting1 = "Hi"; // instantiate greeting1
System.out.println( "\nThe length of " + greeting1 + " is "
+ greeting1.length( ) );

String greeting2 = new String( "Hello" ); // instantiate greeting2

int len = greeting2.length( );  // len will be assigned 5

System.out.println( "The length of " + greeting2 + " is " + len );

String empty = new String( );
System.out.println( "The length of the empty String is "
+ empty.length( ) );

String greeting2Upper = greeting2.toUpperCase( );
System.out.println( );
System.out.println( greeting2 + " converted to upper case is "
+ greeting2Upper );

System.out.println("2nd Character = " + greeting2.charAt(1));

String invertedName = "Lincoln, Abraham";
int comma = invertedName.indexOf( ',' ); // find the comma

System.out.println( "\nThe index of " + ',' + " in "
+ invertedName + " is " + comma );

// extract all characters up to comma
String lastName = invertedName.substring( 0, comma );
System.out.println( "Dear Mr. " + lastName );
}
}
```

# OOP PRINCIPLES

## ENCAPSULATION

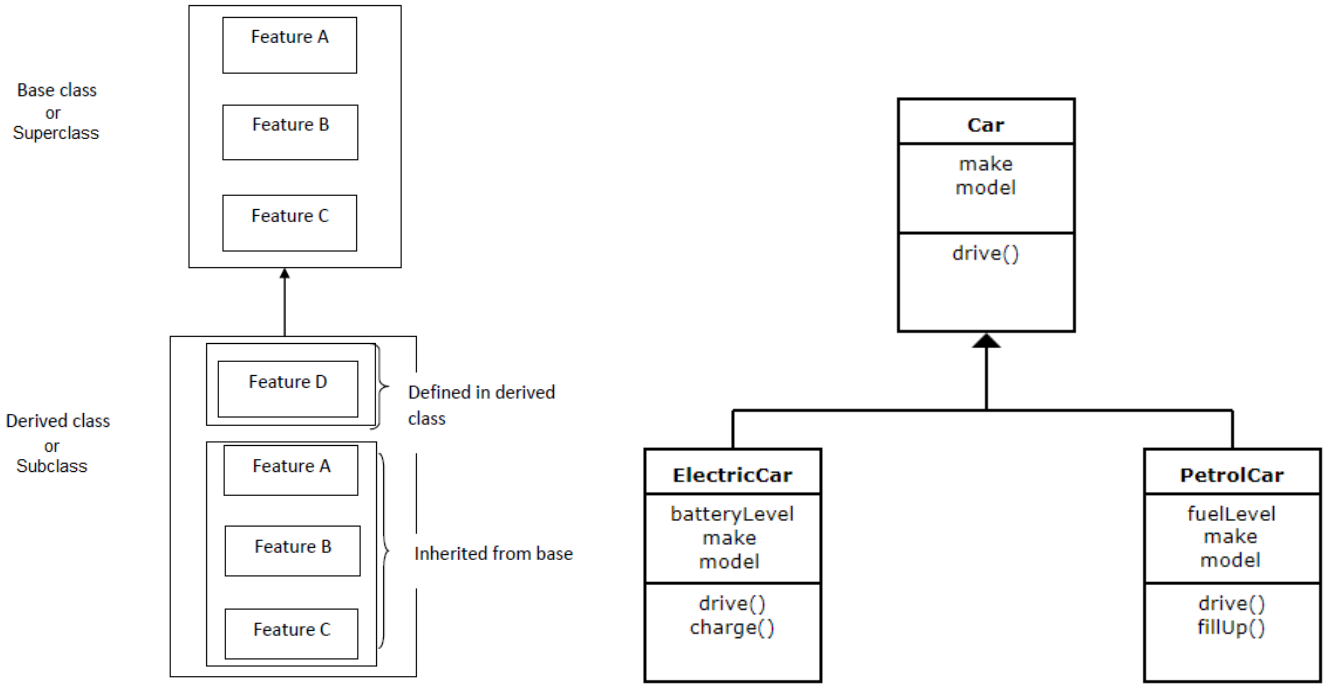
Encapsulation is all about wrapping variables and methods in one single unit. Encapsulation is also known as data hiding. When you design your class you may (and you should) make your variables hidden from other classes and provide methods to manipulate the data instead. To achieve encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.




## INHERITANCE

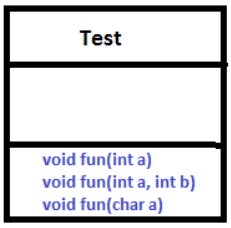
Inheritance is the OOP ability that allows Java classes to be derived from other classes. It transfers the characteristics of a class to other classes that are derived from it. The parent class is called a superclass and the derivatives are called subclasses. Subclasses inherit fields and methods from their superclasses.



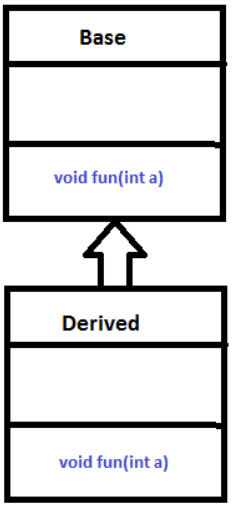
## POLYMORPHISM

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered as one of the important features of Object Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms. In Java polymorphism is mainly divided into two types:

- **Compile time polymorphism:** It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.
- **Runtime polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.




Overloading




Overriding

## METHOD OVERLOADING

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. Argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters. In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters. For example:

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters. For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters. For example:

```
add(int, float)
add(float, int)
```

**PROGRAM 2:** Demonstrate method overloading. To run Program save this file "Overload.java"

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}

class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

**INVALID CASE OF METHOD OVERLOADING:**

If two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

**AUTOMATIC TYPE CONVERSION IN OVERLOADING**

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

**PROGRAM 3:** Automatic type conversions apply to overloading. Save this file "Overload.java"

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

**CONSTRUCTOR OVERLOADING**

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.

**PROGRAM 4:** Constructor Overloading

```
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;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
```

```
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

**PASSING AND RETURNING OBJECTS IN JAVA**

Although Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.

When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

**PROGRAM 5:** Passing objects to methods.

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

**PROGRAM 6:** Here, Box allows one object to initialize another.

```
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
```

```
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

**PROGRAM 7:** Primitive types are passed by value.

```
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

**PROGRAM 8:** Objects are passed by reference.

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

**PROGRAM 9:** Returning an object from methods.

```
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: " + ob2.a);
}
}
```