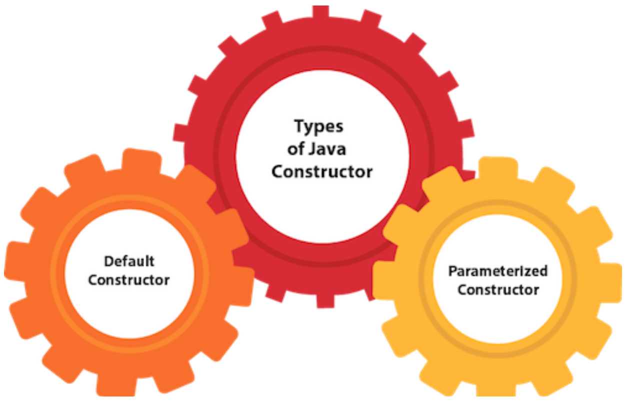
# UNIT -2 Derived Syntactical Constructs in Java

**Constructors in Java**
In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
It is a special type of method which is used to initialize the object.
Every time an object is created using the new() keyword, at least one constructor is called.
**Need of Constructor**
Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in computer’s memory), then can a box be there with no value defined for its dimensions. The answer is no.
So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).
**When is a Constructor called ?**
Each time an object is created using new() keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the data members of the same class. A constructor is invoked at the time of object or instance creation.

## Types of Java constructors

There are two types of constructors in Jav

1. Default constructor (no-arg construc
2. Parameterized constructor



### 1.Java Default Constructor

**No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don’t define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
Default constructor provides the default values to the object like 0, null, etc. depending on the type.
Example :

```
class Bike1{
//creating a default constructor
Bike1(){
System.out.println("Bike is created");
}
//main method
public static void main(String args[])
{
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

```
Bike is created
```

## 2.Parameterized constructor

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

Example

```
class Student{
    int id;
    String name;
    //creating a parameterized constructor
    Student(int i,String n)
{
    id = i;
    name = n;
    }
    //method to display the values
    void display()
{
System.out.println(id+" "+name);
}

    public static void main(String args[])
 {
    //creating objects and passing values
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
class Student{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student(int i,String n)
{
    id = i;
    name = n;
}
```

```
    //creating three arg constructor
    Student(int i,String n,int a)
{
    id = i;
    name = n;
    age=a;
    }
    void display()
{
System.out.println(id+" "+name+" "+age);
}

    public static void main(String args[])
{
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

```
Output:
111 Karan 0
222 Aryan 25
```

**Java Method**

**A Java method is a collection of statements that are grouped together to perform an operation.** When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console. Methods are used to perform certain actions, and they are also known as **functions**. Why use methods? To reuse code: define the code once, and use it many times.

**What is a method?**

In mathematics, you might have studied about functions. For example, f(x) = x2 is a function that returns squared value of x.

```
If x = 2, then f(2) = 4


If x = 3, f(3) = 9
```

Similarly, in programming, a function is a block of code that performs a specific task. In object-oriented programming, method is a jargon used for function. Methods are bound to a class and they define the behavior of a class.

**Types of Java methods**

Depending on whether a method is defined by the user, or available in standard library, there are two types of methods:

1. Standard Library Methods
2. User-defined Methods

**Standard Library Methods**

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- `print()` is a method of `java.io.PrintStream`. The `print("...")` prints the string inside quotation marks.
- `sqrt()` is a method of Math class. It returns square root of a number.

Here's an working example:

```
public class Numbers {
    public static void main(String[] args) {
        System.out.print("Square root of 4 is: " + Math.sqrt(4));
    }
}
```

When you run the program, the output will be:

```
Square root of 4 is: 2.0
```

## User-defined Method

You can also define methods inside a class as per your wish. Such methods are called user-defined methods.

```
class Main {

   public static void main(String[] args) {
       System.out.println("About to encounter a method.");

       // method call
       myMethod();

       System.out.println("Method was executed successfully!");
   }

   // method definition
   private static void myMethod(){
       System.out.println("Printing from inside myMethod()!");
   }
}
```

When you run the program, the output will be:

```
About to encounter a method.
Printing from inside myMethod().
Method was executed successfully!
```

### Example: Return Value from Method

```
class SquareMain {
    public static void main(String[] args) {
        int result;
        result = square();
        System.out.println("Squared value of 10 is: " + result);
    }

          public static int square() {
        // return statement
        return 10 * 10;
    }

}
```

When you run the program, the output will be:

```
Squared value of 10 is: 100
```

**Java Program to Show the Nesting of Methods**

This is a Java Program to Show the Nesting of Methods. When a method in java calls another method in the same class, it is called Nesting of methods.

```
import java.util.Scanner;
public class Nesting_Methods
{
    int perimeter(int l, int b)
    {
            int pr = 12 * (l + b);
            return pr;
    }
    int area(int l, int b)
    {
            int pr = perimeter(l, b);
            System.out.println("Perimeter:"+pr);
            int ar = 6 * l * b;
            return ar;
    }
    int volume(int l, int b, int h)
    {
        int ar = area(l, b);
        System.out.println("Area:"+ar);
        int vol ;
        vol = l * b * h;
        return vol;
    }
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter length of cuboid:");
        int l = s.nextInt();
        System.out.print("Enter breadth of cuboid:");
        int b = s.nextInt();
        System.out.print("Enter height of cuboid:");
        int h = s.nextInt();
        Nesting_Methods obj = new Nesting_Methods();
        int vol = obj.volume(l, b, h);
        System.out.println("Volume:"+vol);
    }
}
```

Output:

```
$ javac Nesting_Methods.java
$ java Nesting_Methods

Enter length of cuboid:5
Enter breadth of cuboid:6
Enter height of cuboid:7
Perimeter:132
Area:180
Volume:210
```

## Parameter Passing in Java

The two most prevalent modes of passing arguments to methods are "passing-by-value" and "passing-by-reference". Different programming languages use these concepts in different ways. **As far as Java is concerned, everything is strictly *Pass-by-Value*.**

Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***. Also, the arguments which

### 1.Pass-by-Value

When a parameter is pass-by-value, the caller and the callee method operate on two different variables which are copies of each other. Any changes to one variable don't modify the other.

It means that while calling a method, **parameters passed to the callee method will be clones of original parameters.** Any modification done in callee method will have no effect on the original parameters in caller method.

#### 1.a.Passing Primitive Types

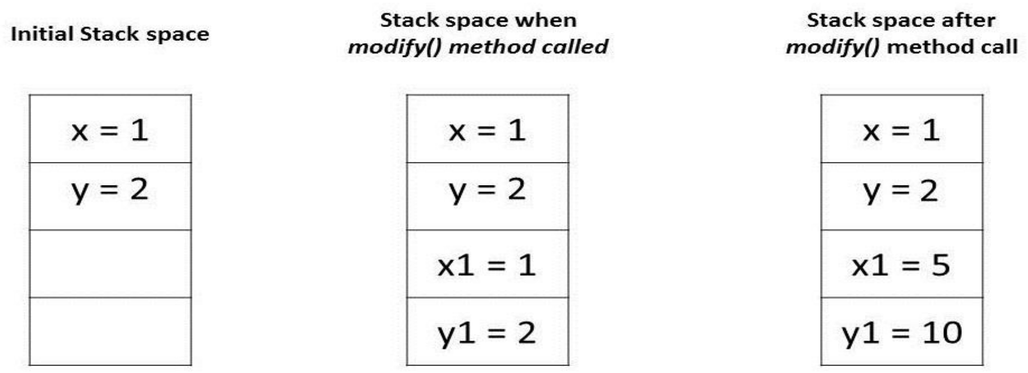
The Java Programming Language features eight primitive data types. **Primitive variables are directly stored in stack memory. Whenever any variable of primitive data type is passed as an argument, the actual parameters are copied to formal arguments and these formal arguments accumulate their own space in stack memory.**

The lifespan of these formal parameters lasts only as long as that method is running, and upon returning, these formal arguments are cleared away from the stack and are discarded.

```
public class PrimitivesUnitTest {

    @Test
    public void whenModifyingPrimitives_thenOriginalValuesNotModified() {

        int x = 1;
        int y = 2;

        // Before Modification
        assertEquals(x, 1);
        assertEquals(y, 2);

        modify(x, y);
        // After Modification
        assertEquals(x, 1);
        assertEquals(y, 2);
    }

    public static void modify(int x1, int y1) {
        x1 = 5;
        y1 = 10;
    }
}
```

Let's try to understand the assertions in the above program by analyzing how these values are stored in memory:

1. The variables *"x"* and *"y"* in the main method are primitive types and their values are directly stored in the stack memory
2. When we call method *modify()*, an exact copy for each of these variables is created and stored at a different location in the stack memory
3. Any modification to these copies affects only them and leaves the original variables unaltered



1.
In cts
are referred from references called reference variables.

A Java object, in contrast to Primitives, is stored in two stages. The reference variables are stored in stack memory and the object that they're referring to, are stored in a Heap memory.

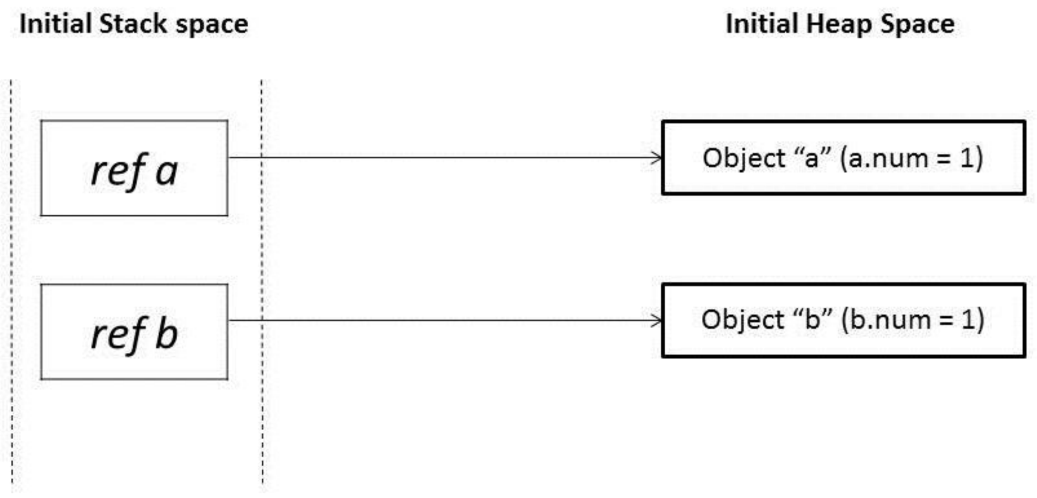
**Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.**

**As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object.** However, if we allocate a new object to the passed reference variable, then it won't be reflected in the original object.

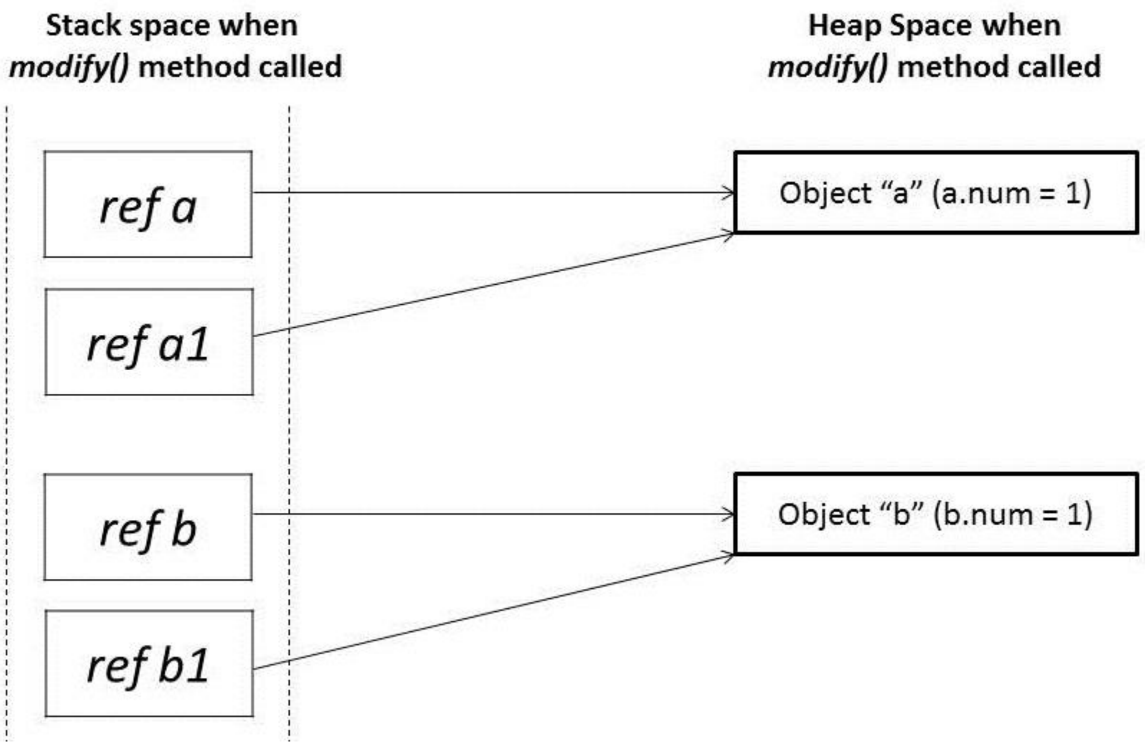
Let's try to comprehend this with the help of a code example:

```
public class NonPrimitivesUnitTest {

    @Test
    public void whenModifyingObjects_thenOriginalObjectChanged() {
        Foo a = new Foo(1);
        Foo b = new Foo(1);

        // Before Modification
        assertEquals(a.num, 1);
        assertEquals(b.num, 1);

        modify(a, b);

        // After Modification
        assertEquals(a.num, 2);
        assertEquals(b.num, 1);
    }

    public static void modify(Foo a1, Foo b1) {
        a1.num++;

        b1 = new Foo(1);
        b1.num++;
    }
}

class Foo {
    public int num;

    public Foo(int num) {
        this.num = num;
    }
}
```

Let's analyze the assertions in the above program. We have passed objects a and b in modify() method that has the same value 1. Initially, these object references are pointing to two distinct object locations in a heap space

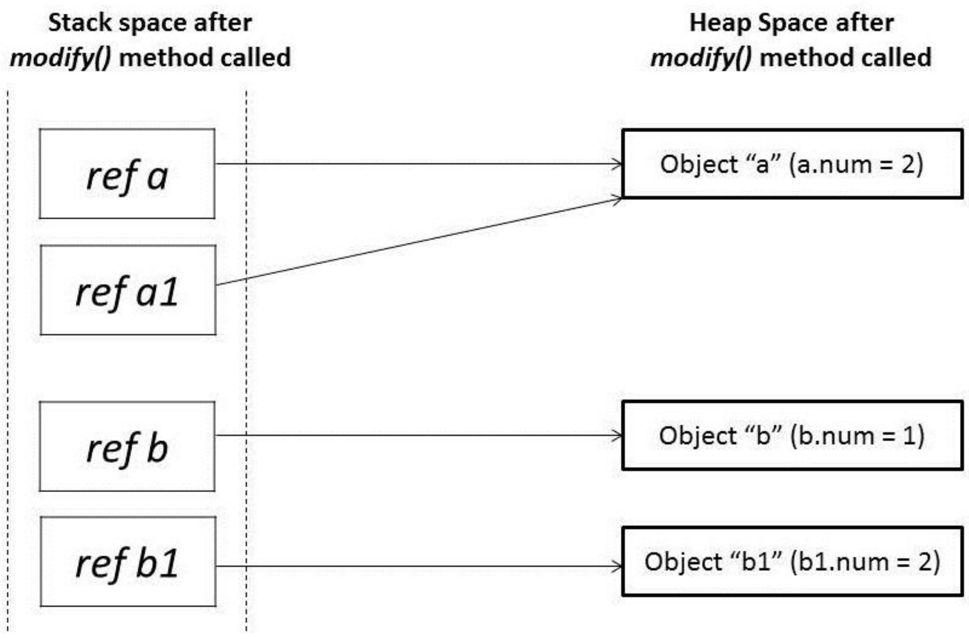


When these references *a* and *b* are passed in the *modify()* method, it creates mirror copies of those references *a1* and *b1* which point to the same old objects



in the *modify()* method, when we modify reference *a1*, it changes the original object. However, for a reference *b1,* we have assigned a new object. So it's now pointing to a new object in heap memory.

Any change made to *b1* will not reflect anything in the original object:



## Conclusion

We looked at how parameter passing is handled in case of Primitives and Non-Primitives.

We learned that parameter passing in Java is always Pass-by-Value. However, the context changes depending upon whether we're dealing with Primitives or Objects:

1. For Primitive types, parameters are pass-by-value
2. For Object types, the object reference is pass-by-value

**As far as Java is concerned, everything is strictly *Pass-by-Value*.**

### 2.Pass-by-Reference

When a parameter is pass-by-reference, the caller and the callee operate on the same object.

It means that when a variable is pass-by-reference, **the unique identifier of the object is sent to the method.** Any changes to the parameter's instance members will result in that change being made to the original value.

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

| Java Constructor | Java Method |
|---|---|
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

### This Keyword

**Keyword** 'THIS' in **Java** is a reference variable that refers to the current object. "this" is a reference to the current object, whose method is being called upon. You can use **"this" keyword** to avoid naming conflicts in the method/constructor of your instance/object.

## Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

### 1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

**Understanding the problem without this keyword**

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

**Solution of the above problem by this keyword**

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

**Program where this keyword is not required**

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

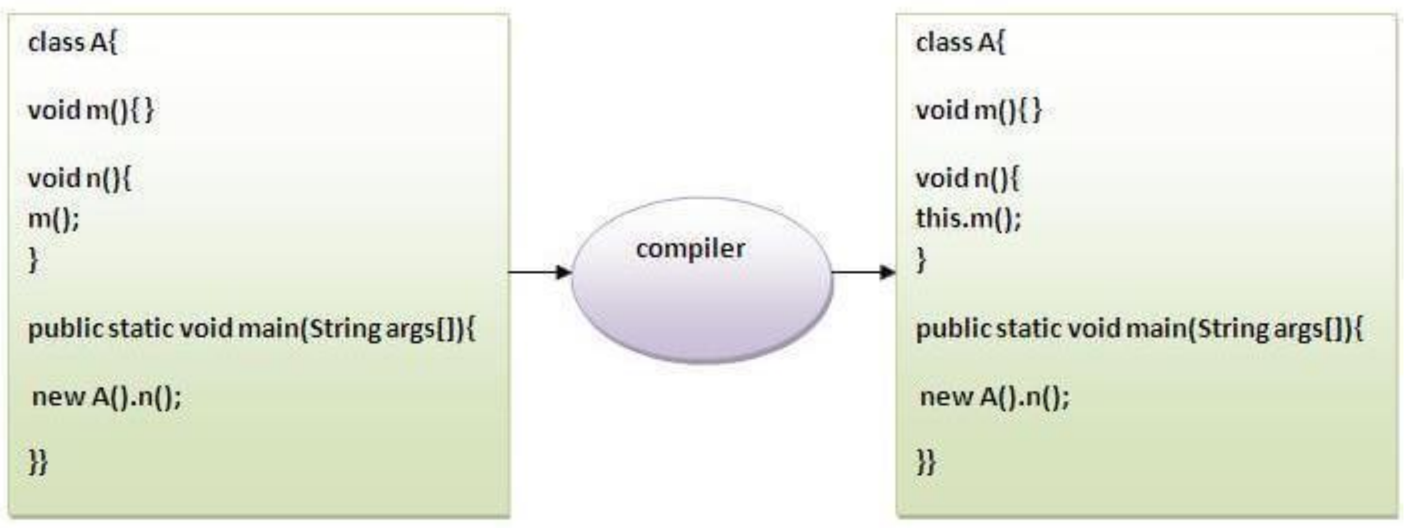
Output:

```
111 ankit 5000
112 sumit 6000
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

**2) this: to invoke current class method**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

**3) this() : to invoke current class constructor**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
```

```
A a=new A(10);
}}
```

Output:

```
hello a
10
```

**Calling parameterized constructor from default constructor:**

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

**Real usage of this() constructor call**

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
```

```
class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit java null
112 sumit java 6000
```

Rule: Call to this() must be first statement in constructor.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class TestThis8{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

```
Compile Time Error: Call to this must be first statement in constructor
```

**4) this: to pass as an argument in the method**

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

Output:

```
method is invoked
```

**Application of this that can be passed as an argument:**

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

### 5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

```
Output:10
```

### 6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

**Syntax of this that can be returned as a statement**

```
return_type method_name(){
```

```
2. return this;
3. }
```

## Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.The arguments passed from the console can be received in the java program and it can be used as an input.So, it provides a convenient way to check the behavior of the program for the different values

**Example: To Learn java Command Line Arguments**

```
class Demo{
     public static void main(String b[]){
          System.out.println("Argument one = "+b[0]);
          System.out.println("Argument two = "+b[1]);
     }
}
```

## Varargs: Variable-Length Arguments

Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called **varargs** and it 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**.
- 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) {
        //...
}
/**
 * Demonstrate variable-length arguments.
 */
class VarArgs { // vaTest() now uses a vararg.
    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 vaTest() can be called with a
    // variable number of arguments.
```

```
        vaTest(); // no args
        vaTest(10); // 1 arg
        vaTest(1, 2, 3); // 3 args
    }
}
```

The output of the program is:

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

The output shows that method vaTest() takes variable number of arguments and hence it is varargs method.

A Definition of Java Garbage Collection

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

## How Java Garbage Collection Works

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted. The garbage collection implementation lives in the JVM. Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification. Although there are many JVMs, Oracle's HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process. In the first step, unreferenced objects are identified and marked as ready for garbage collection. In the second step, marked objects are deleted. Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap. The compaction process makes it easier to allocate memory to new objects sequentially after the block of memory allocated to existing objects.

### Java Object finalize() Method

Finalize() is the method of Object class. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

Syntax

```
protected void finalize() throws Throwable
```

Throw

**Throwable** - the Exception is raised by this method

Example

```
public class JavafinalizeExample1 {
     public static void main(String[] args)
    {
        JavafinalizeExample1 obj = new JavafinalizeExample1();
        System.out.println(obj.hashCode());
        obj = null;
        // calling garbage collector
        System.gc();
        System.out.println("end of garbage collection");

    }
    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output

```
705927765
end of garbage collection
finalize method called
```

## Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

## Methods of Object class

The Object class provides many methods. They are as follows:

| **Method** | **Description** |
|---|---|
| public final Class getClass() √<br><br>(imp) | returns the Class class object of this object.<br><br>The Class class can further be used to get the metadata |

| | |
|---|---|
| | of this class. |
| public int hashCode() √ | returns the hashcode number for this object. |
| public boolean equals(Object obj) √ | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() √ | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

**Access Modifiers in Java**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers.**

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
|---|---|---|---|---|
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

1) Private
The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor
If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

```
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

```
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output:Hello

4) Public

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

//save by A.java

```
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}

Output:Hello
```

## Java Arrays

Normally, an array is a collection of similar type of elements which have a contiguous memory location.

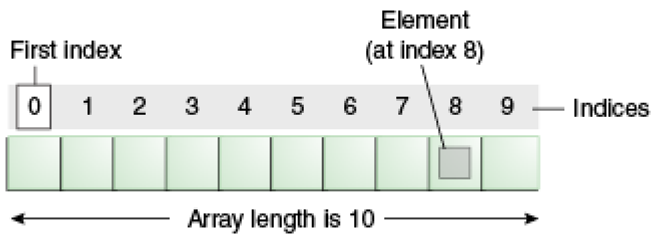
**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



**Advantages**

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

**Disadvantages**

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Example

```
public class Tester {

public static void main(String[] args) {

double[] myList = {1.9, 2.9, 3.4, 3.5};

// Print all the array elements

for (double element: myList) {

System.out.print(element + " ");

}

System.out.println();

int[][] multidimensionalArray = { {1,2},{2,3}, {3,4} };

for(int i = 0 ; i < 3 ; i++) {

//row

for(int j = 0 ; j < 2; j++) {

System.out.print(multidimensionalArray[i][j] + " ");

}

System.out.println();

}

}

}
```

Output

```
1.9 2.9 3.4 3.5

1 2

2 3
```

3 4

## Java String

In Java, string is basically an object that represents sequence of char values.
An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

**1) String Literal**

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

**2) By new keyword**

```
String s=new String("Welcome");//creates two objects and one reference variable
```

**Java String Example**

```
public class StringExample{

public static void main(String args[]){

String s1="java";//creating string by java string literal

char ch[]={'s','t','r','i','n','g','s'};

String s2=new String(ch);//converting char array to string

String s3=new String("example");//creating java string by new keyword

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}}
```

output

```
java
strings
example
```

**Java String class methods**

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

| No. | Method | Description |
|---|---|---|
| 1 | char charAt(int index) | returns char value for the particular index |
| 2 | int length() | returns string length |
| 3 | static String format(String format, Object... args) | returns a formatted string. |
| 4 | static String format(Locale l, String format, Object... args) | returns formatted string with given locale. |
| 5 | String substring(int beginIndex) | returns substring for given begin index. |
| 6 | String substring(int beginIndex, int endIndex) | returns substring for given begin index and end index. |
| 7 | boolean contains(CharSequence s) | returns true or false after matching the sequence of char value. |
| 8 | static String join(CharSequence delimiter, CharSequence... elements) | returns a joined string. |
| 9 | static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements) | returns a joined string. |
| 10 | boolean equals(Object another) | checks the equality of string with the given object. |
| 11 | boolean isEmpty() | checks if string is empty. |
| 12 | String concat(String str) | concatenates the specified string. |

|  |  |  |
|---|---|---|
| 13 | String replace(char old, char new) | replaces all occurrences of the specified char value. |
| 14 | String replace(CharSequence old, CharSequence new) | replaces all occurrences of the specified CharSequence. |
| 15 | static String equalsIgnoreCase(String another) | compares another string. It doesn't check case. |
| 16 | String[] split(String regex) | returns a split string matching regex. |
| 17 | String[] split(String regex, int limit) | returns a split string matching regex and limit. |
| 18 | String intern() | returns an interned string. |
| 19 | int indexOf(int ch) | returns the specified char value index. |
| 20 | int indexOf(int ch, int fromIndex) | returns the specified char value index starting with given index. |
| 21 | int indexOf(String substring) | returns the specified substring index. |
| 22 | int indexOf(String substring, int fromIndex) | returns the specified substring index starting with given index. |
| 23 | String toLowerCase() | returns a string in lowercase. |
| 24 | String toLowerCase(Locale l) | returns a string in lowercase using specified locale. |
| 25 | String toUpperCase() | returns a string in uppercase. |
| 26 | String toUpperCase(Locale l) | returns a string in uppercase using specified locale. |

|  |  |  |
|---|---|---|
| 27 | String trim() | removes beginning and ending spaces of this string. |
| 28 | static String valueOf(int value) | converts given type into string. It is an overloaded method. |

**public char charAt(int index)**

This method requires an integer argument that indicates the position of the character that the method returns.This method returns the character located at the String's specified index. Remember, String indexes are zero-based—for example,

```
String x = "airplane";
System.out.println( x.charAt(2) ); // output is 'r'
```

**public String concat(String s)**

This method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke the method—for example,

```
String x = "book";
System.out.println( x.concat(" author") ); // output is "book author"
```

The overloaded + and += operators perform functions similar to the concat()method—for example,

```
String x = "library";
System.out.println( x + " card"); // output is "library card"
String x = "United";
x += " States"
System.out.println( x ); // output is "United States"
```

**public boolean equalsIgnoreCase(String s)**

This method returns a boolean value (true or false) depending on whether the value of the String in the argument is the same as the value of the String used to invoke the method. This method will return true even when characters in the String objects being compared have differing cases—for example,

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

**public int length()**

This method returns the length of the String used to invoke the method—for example,

```
String x = "01234567";
System.out.println( x.length() ); // returns "8"
```

**public String replace(char old, char new)**

This method returns a String whose value is that of the String used to invoke the method, updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—for example,

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') ); // output is  "oXoXoXoX"
```

**public String substring(int begin)/ public String substring(int begin, int end)**

The substring() method is used to return a part (or substring) of the String used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters to the end of the original String. If the call has two arguments, the substring returned will end with the character located in the nth position of the original String where n is the second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned String will be in the original String's 7 position, which is index 6. Let's look at some examples:

```
String x = "0123456789"; // the value of each char is the same as its index!
System.out.println( x.substring(5) ); // output is "56789"
System.out.println( x.substring(5, 8)); // output is "567"
```

**public String toLowerCase()**

This method returns a String whose value is the String used to invoke the method, but with any uppercase characters converted to lowercase—for example,

```
String x = "A New Java Book";
System.out.println( x.toLowerCase() ); // output is "a new java book"
```

**public String toUpperCase()**

This method returns a String whose value is the String used to invoke the method, but with any lowercase characters converted touppercase—for example,

```
String x = "A New Java Book";
System.out.println( x.toUpperCase() ); // output is"A NEW JAVA BOOK"
```

**public String trim()**

This method returns a String whose value is the String used to invoke the method, but with any leading or trailing blank spaces removed—for example,

```
String x = " hi ";
System.out.println( x + "x" ); // result is" hi x"
System.out.println(x.trim() + "x"); // result is "hix"
```

**public char[ ] toCharArray( )**

This method will produce an array of characters from characters of String object. For example

```
String s = "Java";
Char [] arrayChar = s.toCharArray();  //this will produce array of size 4
```

**public boolean contains("searchString")**

This method returns true of target String is containing search String provided in the argument. For example-

```
String x = "Java is programming language";
System.out.println(x.contains("Amit")); // This will print false
System.out.println(x.contains("Java")); // This will print true
```

Below program demonstrate all above methods.

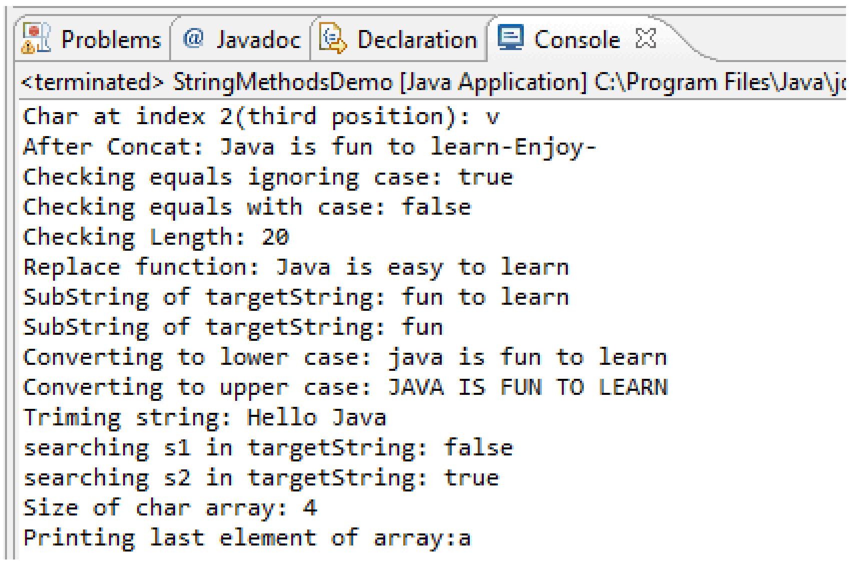
Java Code: Go to the editor

```
public class StringMethodsDemo {

        public static void main(String[] args) {

                String targetString = "Java is fun to learn";

                String s1= "JAVA";

                String s2= "Java";

                String s3 = "  Hello Java  ";



                System.out.println("Char at index 2(third position): " +
targetString.charAt(2));

                System.out.println("After Concat: "+ targetString.concat("-Enjoy-"));

                System.out.println("Checking equals ignoring case: "
+s2.equalsIgnoreCase(s1));

                System.out.println("Checking equals with case: " +s2.equals(s1));

                System.out.println("Checking Length: "+ targetString.length());

                System.out.println("Replace function: "+ targetString.replace("fun",
"easy"));

                System.out.println("SubString of targetString: "+
targetString.substring(8));

                System.out.println("SubString of targetString: "+
targetString.substring(8, 12));

                System.out.println("Converting to lower case: "+
targetString.toLowerCase());

                System.out.println("Converting to upper case: "+
targetString.toUpperCase());

                System.out.println("Triming string: " + s3.trim());

                System.out.println("searching s1 in targetString: " +
targetString.contains(s1));

                System.out.println("searching s2 in targetString: " +
targetString.contains(s2));
```

```
                char [] charArray = s2.toCharArray();

                System.out.println("Size of char array: " + charArray.length);

                System.out.println("Printing last element of array: " + charArray[3]);


        }


}
```

Copy

Output:



```
<terminated> StringMethodsDemo [Java Application] C:\Program Files\Java\j
Char at index 2(third position): v
After Concat: Java is fun to learn-Enjoy-
Checking equals ignoring case: true
Checking equals with case: false
Checking Length: 20
Replace function: Java is easy to learn
SubString of targetString: fun to learn
SubString of targetString: fun
Converting to lower case: java is fun to learn
Converting to upper case: JAVA IS FUN TO LEARN
Triming string: Hello Java
searching s1 in targetString: false
searching s2 in targetString: true
Size of char array: 4
Printing last element of array:a
```

## Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

### Important Constructors of StringBuffer class

| Constructor | Description |
|---|---|
| StringBuffer() | creates an empty string buffer with the initial capacity of 16. |
| StringBuffer(String str) | creates a string buffer with the specified string. |

| | |
|---|---|
| StringBuffer(int capacity) | creates an empty string buffer with the specified capacity as length. |

## StringBuffer Methods

1. length(): Returns the StringBuffer object's length.
2. capacity(): Returns the capacity of the StringBuffer object.

```
package com.journaldev.java;

public class StringBufferExample {

	public static void main(String[] args) {
		StringBuffer sb = new StringBuffer("Hello");
		int sbLength = sb.length();
		int sbCapacity = sb.capacity();
		System.out.println("String Length of " + sb + " is " + sbLength);
		System.out.println("Capacity of " + sb + " is " + sbCapacity);
	}

}
```

Output produced by above StringBuffer example program:

```
String Length of Hello is 5
Capacity of Hello is 21
```

3. append(): appends the specified argument string representation at the end of the existing String Buffer. This method is overloaded for all the primitive data types and Object.

```
StringBuffer sb = new StringBuffer("Hello ");

sb.append("World ");

sb.append(2017);

System.out.println(sb);
```

**Output:** Hello World 2017

4. insert(): insert() method takes two parameters – the index integer value to insert a value and the value to be inserted. The index tells StringBuffer where to insert the passed character sequence. Again this method is overloaded to work with primitive data types and Objects.

```
StringBuffer sb = new StringBuffer("HelloWorld ");

sb.insert(5, " ");

sb.insert(sb.length(), 2017);

System.out.println(sb);
```

**Output:** Hello World 2017

5. reverse(): Reverses the existing String or character sequence content in the buffer and returns it. The object must have an existing content or else a NullPointerException is thrown.

```
StringBuffer sb = new StringBuffer("Hello World");

System.out.println(sb.reverse());
```

**Output:** dlroW olleH

6. delete(int startIndex, int endIndex): accepts two integer arguments. The former serves as the starting delete index and latter as the ending delete index. Therefore the character sequence between startIndex and endIndex-1 are deleted. The remaining String content in the buffer is returned.

```
StringBuffer sb = new StringBuffer("Hello World");

System.out.println(sb.delete(5,11)); //prints Hello
```

7. deleteCharAt(int index): deletes single character within the String inside the buffer. The location of the deleted character is determined by the passed integer index. The remaining String content in the buffer is returned.

```
StringBuffer sb = new StringBuffer("Hello World!");

System.out.println(sb.deleteCharAt(11)); //prints "Hello World"
```

8. replace(int startIndex, int endIndex, String str): Accepts three arguments: first two being the starting and ending index of the existing String Buffer. Therefore the character sequence between startIndex and endIndex-1 are removed. Then the String passed as third argument is inserted at startIndex.

```
StringBuffer sb = new StringBuffer("Hello World!");

System.out.println(sb.replace(6,11,"Earth")); //prints "Hello Earth!"
```

That’s all for StringBuffer in Java. Most of the time we don’t need to use StringBuffer because String is immutable and we can use StringBuilder in single threaded environments. You should use StringBuffer only when multiple threads are modifying it’s contents.

## Java Vector Class

Java Vector class comes under the java.util package. The vector class implements a growable array of objects. Like an array, it contains the component that can be accessed using an integer index.

Vector is very useful if we don't know the size of an array in advance or we need one that can change the size over the lifetime of a program.

Vector implements a dynamic array that means it can grow or shrink as required. It is similar to the ArrayList, but with two differences-

- Vector is synchronized.

- The vector contains many legacy methods that are not the part of a collections framework

**Vectors Constructors**

Listed below are the multiple variations of vector constructors available to use:

1. **Vector(int initialCapacity, int Increment)** – Constructs a vector with given initialCapacity and its Increment in size.
2. **Vector(int initialCapacity) –** Constructs an empty vector with given initialCapacity. In this case, Increment is zero.
3. **Vector()** – Constructs a default vector of capacity 10.
4. **Vector(Collection c) –** Constructs a vector with a given collection, the order of the elements is same as returned by the collection's iterator.

| SN | Method | Description |
|---|---|---|
| 1) | add() | It is used to append the specified element in the given vector. |
| 2) | addAll() | It is used to append all of the elements in the specified collection to the end of this Vector. |
| 3) | addElement() | It is used to append the specified component to the end of this vector. It increases the vector size by one. |
| 4) | capacity() | It is used to get the current capacity of this vector. |
| 5) | clear() | It is used to delete all of the elements from this vector. |
| 6) | clone() | It returns a clone of this vector. |
| 7) | contains() | It returns true if the vector contains the specified element. |
| 8) | containsAll() | It returns true if the vector contains all of the elements in the specified collection. |
| 9) | copyInto() | It is used to copy the components of the vector into the specified array. |

| | | |
|---|---|---|
| 10) | elementAt() | It is used to get the component at the specified index. |
| 11) | elements() | It returns an enumeration of the components of a vector. |
| 12) | ensureCapacity() | It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument. |
| 13) | equals() | It is used to compare the specified object with the vector for equality. |
| 14) | firstElement() | It is used to get the first component of the vector. |
| 15) | forEach() | It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| 16) | get() | It is used to get an element at the specified position in the vector. |
| 17) | hashCode() | It is used to get the hash code value of a vector. |
| 18) | indexOf() | It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 19) | insertElementAt() | It is used to insert the specified object as a component in the given vector at the specified index. |
| 20) | isEmpty() | It is used to check if this vector has no components. |

|  |  |  |
|---|---|---|
| 21) | iterator() | It is used to get an iterator over the elements in the list in proper sequence. |
| 22) | lastElement() | It is used to get the last component of the vector. |
| 23) | lastIndexOf() | It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 24) | listIterator() | It is used to get a list iterator over the elements in the list in proper sequence. |
| 25) | remove() | It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged. |
| 26) | removeAll() | It is used to delete all the elements from the vector that are present in the specified collection. |
| 27) | removeAllElements() | It is used to remove all elements from the vector and set the size of the vector to zero. |
| 28) | removeElement() | It is used to remove the first (lowest-indexed) occurrence of the argument from the vector. |
| 29) | removeElementAt() | It is used to delete the component at the specified index. |
| 30) | removeIf() | It is used to remove all of the elements of the collection that satisfy the given predicate. |
| 31) | removeRange() | It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive. |

| | | |
|---|---|---|
| 32) | replaceAll() | It is used to replace each element of the list with the result of applying the operator to that element. |
| 33) | retainAll() | It is used to retain only that element in the vector which is contained in the specified collection. |
| 34) | set() | It is used to replace the element at the specified position in the vector with the specified element. |
| 35) | setElementAt() | It is used to set the component at the specified index of the vector to the specified object. |
| 36) | setSize() | It is used to set the size of the given vector. |
| 37) | size() | It is used to get the number of components in the given vector. |
| 38) | sort() | It is used to sort the list according to the order induced by the specified Comparator. |
| 39) | spliterator() | It is used to create a late-binding and fail-fast Spliterator over the elements in the list. |
| 40) | subList() | It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive. |
| 41) | toArray() | It is used to get an array containing all of the elements in this vector in correct order. |
| 42) | toString() | It is used to get a string representation of the vector. |
| 43) | trimToSize() | It is used to trim the capacity of the vector to the vector's current size. |

```
import java.util.*;
public class VectorExample1 {
```

```java
        public static void main(String args[]) {
            //Create an empty vector with initial capacity 4
            Vector<String> vec = new Vector<String>(4);
            //Adding elements to a vector
            vec.add("Tiger");
            vec.add("Lion");
            vec.add("Dog");
            vec.add("Elephant");
            //Check size and capacity
            System.out.println("Size is: "+vec.size());
            System.out.println("Default capacity is: "+vec.capacity());
            //Display Vector elements
            System.out.println("Vector element is: "+vec);
            vec.addElement("Rat");
            vec.addElement("Cat");
            vec.addElement("Deer");
            //Again check size and capacity after two insertions
            System.out.println("Size after addition: "+vec.size());
            System.out.println("Capacity after addition is: "+vec.capacity());
            //Display Vector elements again
            System.out.println("Elements are: "+vec);
            //Checking if Tiger is present or not in this vector
              if(vec.contains("Tiger"))
              {
                 System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));

              }
              else
              {
                 System.out.println("Tiger is not present in the list.");
              }
              //Get the first element
            System.out.println("The first animal of the vector is = "+vec.firstElement());

            //Get the last element
            System.out.println("The last animal of the vector is = "+vec.lastElement());
        }
}
```

**Output:**

```
Size is: 4
Default capacity is: 4
Vector element is: [Tiger, Lion, Dog, Elephant]
Size after addition: 7
Capacity after addition is: 8
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
```

```
Tiger is present at the index 0
The first animal of the vector is = Tiger
The last animal of the vector is = Deer
```

## Wrapper classes in Java

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.

### Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

next →← prev

## Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and obje primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects primitives automatically. The automatic conversion of primitive into an object is known as and vice-versa unboxing.

### Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primit will not change the original value. But, if we convert the primitive value in an obje change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serializat have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with ob
- **Collection Framework:** Java collection framework works with objects only. All classes collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list o wrapper classes are given below:

| **Primitive Type** | **Wrapper class** |
|---|---|
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

## Java Wrapper classes Example

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;
//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}}
```

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
```

```
char value: a
boolean value: true
```

## Java Enums

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

```
class EnumExample1{
//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//creating the main method
public static void main(String[] args) {
//printing all enum
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());

}}
```

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```