UNIT 1

**Basics of JAVA**

1. **INTRODUCTION**

Java program can be defined as a collection of objects that communicate via invoking each other's methods. Let us understand the class, object, methods, and instance variables mean.

* **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
* **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
* **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
* **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

Example

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

public static void main(String []args) {

System.out.println("Hello World"); // prints Hello World

}

}

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps −

* Open notepad and add the code as above.
* Save the file as: MyFirstJavaProgram.java.
* Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
* Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
* Now, type ' java MyFirstJavaProgram ' to run your program.
* You will be able to see ' Hello World ' printed on the window.

Output

C:\> javac MyFirstJavaProgram.java

C:\> java MyFirstJavaProgram

Hello World

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

* **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
* **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

**Example:** *class MyFirstJavaClass*

* **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**cExample:** *public void myMethodName()*

* **Program File Name** − Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

**Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

* **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

**Java Modifiers**

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

* **Access Modifiers** − default, public , protected, private
* **Non-access Modifiers** − final, abstract, strictfp

**Comments in Java**

Java supports single-line and multi-line comments similar to C and C++. These comments are ignored by Java compiler.

Example

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\* This is an example of multi-line comments.

\*/

public static void main(String []args) {

// This is an example of single line comment

/\* This is also an example of single line comment. \*/

System.out.println("Hello World");

}

}

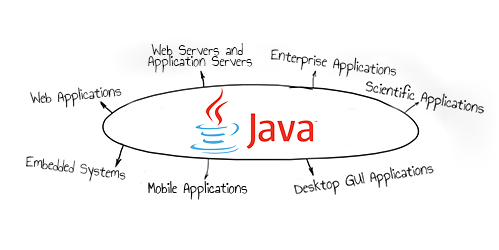
Output

Hello World

**Using Blank Lines**

* A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

1. [**Applications of Java Programming Language**](https://www.invensis.net/blog/it/applications-java-programming-language/)

[](http://www.invensis.net/blog/wp-content/uploads/2015/05/Applications-of-Java-Programming-Invensis.jpg)

Java is widely applicable across different types of applications

Java language was developed by Sun Microsystems in 1995. In subsequent years, the language has become the backbone of millions of applications across multiple platforms including Windows, Macintosh and UNIX-based desktops, Android-based mobiles, embedded systems and enterprise solutions. According to Oracle (that acquired Sun Microsystems in 2010), Java now runs on more than 3 billion devices.

## Types of Applications that Run on Java

### 1. Desktop GUI Applications:

Java provides GUI development through various means like Abstract Windowing Toolkit (AWT), Swing and JavaFX.  While AWT contains a number of pre-constructed components such as menu, button, list, and numerous third-party components, Swing, a GUI widget toolkit, additionally provides certain advanced components like trees, tables, scroll panes, tabbed panel and lists. JavaFX, a set of graphics and media packages, provides Swing interoperability, 3D graphic features and self-contained deployment model which facilitates quick scripting of Java applets and applications.

### 2. Mobile Applications:

Java Platform, Micro Edition (Java ME or J2ME) is a cross-platform framework to build applications that run across all Java supported devices, including feature phones and smart phones. Further, applications for Android, one of the most popular mobile operating systems, are usually scripted in Java using the Android Software Development Kit (SDK) or other environments.

### 3. Embedded Systems:

Embedded systems, ranging from tiny chips to specialized computers, are components of larger electromechanical systems performing dedicated tasks. Several devices, such as SIM cards, blue-ray disk players, utility meters and televisions, use embedded Java technologies.   According to Oracle, 100% of Blu-ray Disc Players and 125 million TV devices employ Java.

### 4. Web Applications:

Java provides support for web applications through Servlets, Struts or JSPs.  The easy programming and higher security offered by the programming language has allowed a large number of government applications for health, social security, education and insurance to be based on Java. Java also finds application in development of eCommerce web applications using open-source eCommerce platforms, such as Broadleaf*.*

### 5. Web Servers and Application Servers:

The Java ecosystem today contains multiple Java web servers and application servers.  While Apache Tomcat, Simple, Jo!, Rimfaxe Web Server (RWS) and Project Jigsaw dominate the web server space, WebLogic, WebSphere, and Jboss EAP dominate commercial application server space.

### 6. Enterprise Applications:

Java Enterprise Edition (Java EE) is a popular platform that provides API and runtime environment for scripting and running enterprise software, including network applications and web-services. Oracle claims Java is running in 97% of enterprise computers.  The higher performance guarantee and faster computing in Java has resulted in high frequency trading systems like Murex to be scripted in the language. It is also the backbone for a variety of banking applications which have Java running from front user end to back server end.

### 7. Scientific Applications:

Java is the choice of many software developers for writing applications involving scientific calculations and mathematical operations. These programs are generally considered to be fast and secure, have a higher degree of portability and low maintenance. Applications like MATLAB use Java both for interacting user interface and as part of the core system.

In conclusion, [Java is widely applicable across different types of applications](https://www.invensis.net/it-outsourcing-services/outsource-java-application-development-services/?utm_source=invensis-blog&utm_campaign=blog-post&utm_medium=content-link&utm_term=applications-java-programming-language). It offers cross-functionality and portability, and these features, among many others, make Java the programming language of choice for software development of a specific nature.

# B[enefits of Java over Other Programming Languages](https://www.invensis.net/blog/it/benefits-of-java-over-other-programming-languages/)

[](http://www.invensis.net/blog/wp-content/uploads/2015/05/Benefits-of-Java-over-Other-Programming-Languages-Invensis.jpg)

Java has significant advantages over other languages and environments

Java is a high level programming language and computing platform developed by Sun Microsystems in 1995. Since then, the language has been regularly updated with Java SE 8.0 version being the latest version, released in March 2014.

Based on the advantages of Java, it gained wide popularity and multiple configurations have been built to suit various types of platforms including Java SE for Macintosh, Windows and UNIX, Java ME for Mobile Applications and Java EE for Enterprise Applications.

With the growing importance of web based and mobile based applications, Java today is the foundation for most networked applications and is considered to be useful for scripting, web-based content, enterprise software, games and mobile applications.

## [Applications of Java

Every enterprise uses Java in one way or other.  As per Oracle, more than 3 billion devices run applications designed on the development platform.

[Java is used to design the following applications](https://www.invensis.net/it-outsourcing-services/outsource-java-application-development-services/?utm_source=invensis-blog&utm_campaign=blog-post&utm_medium=content-link&utm_term=benefits-of-java-over-other-programming-languages):

* Desktop GUI applications
* Embedded systems
* Web applications, including eCommerce applications, front and back office electronic trading systems, settlement and confirmation systems, data processing projects, and more
* Web servers and application servers
* Mobile applications including Android applications
* Enterprise applications
* Scientific applications
* Middleware products]

### Advantages of Java

* Java offers higher cross- functionality and portability as programs written in one platform can run across desktops, mobiles, embedded systems.
* Java is free, simple, object-oriented, distributed, supports multithreading and offers multimedia and network support.
* Java is a mature language, therefore more stable and predictable. The Java Class Library enables cross-platform development.
* Being highly popular at enterprise, embedded and network level, Java has a large active user community and support available.
* Unlike C and C++, Java programs are compiled independent of platform in *bytecode* language which allows the same program to run on any machine that has a JVM installed.
* Java has powerful development tools like Eclipse SDK and NetBeans which have debugging capability and offer integrated development environment.
* Increasing language diversity, evidenced by compatibility of Java with Scala, Groovy, JRuby, and Clojure.
* Relatively seamless forward compatibility from one version to the next

1. **Java Keywords**

The following list shows the reserved words in Java. These reserved words may not be used variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | assert | boolean | break |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | final | finally | float |
| for | goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |
| short | static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | while |  |  |

1. **constant in Java**

**How do I declare a constant in Java**

Java does not directly support constants. However, a static final variable is effectively a constant.

The static modifier causes the variable to be available without loading an instance of the class where it is defined. The final modifier causes the variable to be unchangeable.

Java constants are normally declared in ALL CAPS. Words in Java constants are normally separated by underscores.

An example of constant declaration in Java is written below:

|  |  |  |
| --- | --- | --- |
| [?](https://www.java-tips.org/java-se-tips-100019/24-java-lang/802-how-do-i-declare-a-constant-in-java.html)   |  |  | | --- | --- | | 1  2  3 | public class MaxUnits {     public static final int MAX\_UNITS = 25;  } | |

1. **Java Variables**

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed. All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

variables in java

1. **int** data=50;//Here data is variable

### Types of Variable

There are three types of variables in java:

* local variable
* instance variable
* static variable

[Following are the types of variables in Java −

* Local Variables
* Class Variables (Static Variables)
* Instance Variables (Non-static Variables)]

types of variables in java

#### 1) Local Variable

A variable which is declared inside the method is called local variable.

#### 2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

#### 3) Static variable

A variable that is declared as static is called static variable. It cannot be local.

### Example to understand the types of variables in java

### import java.io.\*;

### class stst

### {

### static int count=0;

### public void increment()

### {

### count++;

### }

### public static void main(String args[])

### {

### VariableDemo obj1=new VariableDemo();

### VariableDemo obj2=new VariableDemo();

### obj1.increment();

### obj2.increment();

### System.out.println("Obj1: count is="+obj1.count);

### System.out.println("Obj2: count is="+obj2.count);

### }

### }

**Class** A{

**int** data=50;//instance variable

**static** **int** m=100;//static variable

**void** method(){

**int** n=90;//local variable

}

}//end of class

In Java, there are several points to remember about identifiers. They are as follows −

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.

Java Variable Example: Add Two Numbers

**class** Simple{

**public** **static** **void** main(String[] args){

**int** a=10;

**int** b=10;

**int** c=a+b;

System.out.println(c);

}}

Output:

20

Java Variable Example: Widening

**class** Simple{

**public** **static** **void** main(String[] args){

**int** a=10;

**float** f=a;

System.out.println(a);

System.out.println(f);

}}

Output:

10

10.0

Java Variable Example: Narrowing (Typecasting)

**class** Simple{

**public** **static** **void** main(String[] args){

**float** f=10.5f;

//int a=f;//Compile time error

**int** a=(**int**)f;

System.out.println(f);

System.out.println(a);

}}

Output:

10.5

10

Java Variable Example: Overflow

**class** Simple{

**public** **static** **void** main(String[] args){

//Overflow

**int** a=130;

**byte** b=(**byte**)a;

System.out.println(a);

System.out.println(b);

}}

Output:

130

-126

Java Variable Example: Adding Lower Type

**class** Simple{

**public** **static** **void** main(String[] args){

**byte** a=10;

**byte** b=10;

//byte c=a+b;//Compile Time Error: because a+b=20 will be int

**byte** c=(**byte**)(a+b);

System.out.println(c);

}}

Output:

20

1. **Data Types in Java**

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

* Primitive data types
* Non-primitive data types



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| boolean | false | 1 bit |
| Char | '\u0000' | 2 byte |
| Byte | 0 | 1 byte |
| Short | 0 | 2 byte |
| Int | 0 | 4 byte |
| Long | 0L | 8 byte |
| Float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

# How to find the size of a primitive data type in Java?

Java has no sizeof operator to find the size of primitive data types but all Java primitive wrappers except Boolean

provide a SIZE constant in bits that could be divided by eight to get the size of a data type in bytes.

Moreover, since Java 8, all primitive wrapper classes (except Boolean) have a BYTES constant, which gives data type's size in bytes. So you can use that also once you have been moved to Java 8. Following is a trivial Java program demonstrating the size of primitive data types through their primitive wrappers.

**class** SizePrimitiveTypes

{

**public** **static** **void** main ([String](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+string)[] args)

{

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of byte: " + ([Byte](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+byte).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of short: " + ([Short](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+short).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of int: " + ([Integer](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+integer).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of long: " + ([Long](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+long).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of char: " + ([Character](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+character).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of float: " + ([Float](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+float).SIZE/8) + " bytes.");

[System](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+system).out.println("Size of double: " + ([Double](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+double).SIZE/8) + " bytes.");

}

}

OUTPUT

======

D:\JavaPrograms>javac SizePrimitiveTypes.java

D:\JavaPrograms>java SizePrimitiveTypes

Size of **byte**: 1 bytes.

Size of **short**: 2 bytes.

Size of **int**: 4 bytes.

Size of **long**: 8 bytes.

Size of **char**: 2 bytes.

Size of **float**: 4 bytes.

Size of **double**: 8 bytes.

Note that size of primitive types in Java is always the same. It is not platform dependent. Also, all primitive data types in Java are signed. Java does not support unsigned types.

**Primitive data types**

The eight primitive data types in Java are:

* boolean, the type whose values are either true or false
* char, the character type whose values are 16-bit Unicode characters
* the arithmetic types:
  + the integral types:
    - byte
    - short
    - int
    - long
  + the floating-point types:
    - float
    - double

Values of class type are references. Strings are references to an instance of class String.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Primitive data types in Java | | | | |
| **Type** | **Description** | **Default** | **Size** | **Example Literals** |
| boolean | true or false | false | 1 bit | true, false |
| byte | twos complement integer | 0 | 8 bits | (none) |
| char | Unicode character | \u0000 | 16 bits | 'a', '\u0041', '\101', '\\', '\'', '\n', 'ß' |
| short | twos complement integer | 0 | 16 bits | (none) |
| int | twos complement integer | 0 | 32 bits | -2, -1, 0, 1, 2 |
| long | twos complement integer | 0 | 64 bits | -2L, -1L, 0L, 1L, 2L |
| float | IEEE 754 floating point | 0.0 | 32 bits | 1.23e100f, -1.23e-100f, .3f, 3.14F |
| double | IEEE 754 floating point | 0.0 | 64 bits | 1.23456e300d, -1.23456e-300d, 1e1d |

1. **Operators**

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator. | A + B will give 30 |
| - (Subtraction) | Subtracts right-hand operand from left-hand operand. | A - B will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | A \* B will give 200 |
| / (Division) | Divides left-hand operand by right-hand operand. | B / A will give 2 |
| % (Modulus) | Divides left-hand operand by right-hand operand and returns remainder. | B % A will give 0 |
| ++ (Increment) | Increases the value of operand by 1. | B++ gives 21 |
| -- (Decrement) | Decreases the value of operand by 1. | B-- gives 19 |

Assume integer variable A holds 10 and variable B holds 20, then

## Example

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

int c = 25;

int d = 25;

System.out.println("a + b = " + (a + b) );

System.out.println("a - b = " + (a - b) );

System.out.println("a \* b = " + (a \* b) );

System.out.println("b / a = " + (b / a) );

System.out.println("b % a = " + (b % a) );

System.out.println("c % a = " + (c % a) );

System.out.println("a++ = " + (a++) );

System.out.println("b-- = " + (a--) );

// Check the difference in d++ and ++d

System.out.println("d++ = " + (d++) );

System.out.println("++d = " + (++d) );

}

}

This will produce the following result −

## Output

a + b = 30

a - b = -10

a \* b = 200

b / a = 2

b % a = 0

c % a = 5

a++ = 10

b-- = 11

d++ = 25

++d = 27

**The Relational Operators**

There are following relational operators supported by Java language.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == (equal to) | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != (not equal to) | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > (greater than) | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < (less than) | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= (greater than or equal to) | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= (less than or equal to) | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

Assume variable A holds 10 and variable B holds 20, then −

## Example

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

System.out.println("a == b = " + (a == b) );

System.out.println("a != b = " + (a != b) );

System.out.println("a > b = " + (a > b) );

System.out.println("a < b = " + (a < b) );

System.out.println("b >= a = " + (b >= a) );

System.out.println("b <= a = " + (b <= a) );

}

}

This will produce the following result −

## Output

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false

**The Bitwise Operators**

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & (bitwise and) | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | (bitwise or) | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ (bitwise XOR) | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ (bitwise compliment) | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << (left shift) | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> (right shift) | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> (zero fill right shift) | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

Assume integer variable A holds 60 and variable B holds 13 then −

## Example

public class Test {

public static void main(String args[]) {

int a = 60; /\* 60 = 0011 1100 \*/

int b = 13; /\* 13 = 0000 1101 \*/

int c = 0;

c = a & b; /\* 12 = 0000 1100 \*/

System.out.println("a & b = " + c );

c = a | b; /\* 61 = 0011 1101 \*/

System.out.println("a | b = " + c );

c = a ^ b; /\* 49 = 0011 0001 \*/

System.out.println("a ^ b = " + c );

c = ~a; /\*-61 = 1100 0011 \*/

System.out.println("~a = " + c );

c = a << 2; /\* 240 = 1111 0000 \*/

System.out.println("a << 2 = " + c );

c = a >> 2; /\* 15 = 1111 \*/

System.out.println("a >> 2 = " + c );

c = a >>> 2; /\* 15 = 0000 1111 \*/

System.out.println("a >>> 2 = " + c );

}

}

This will produce the following result −

## Output

a & b = 12

a | b = 61

a ^ b = 49

~a = -61

a << 2 = 240

a >> 15

a >>> 15

**The Logical Operators**

The following table lists the logical operators −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && (logical and) | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false |
| || (logical or) | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true |
| ! (logical not) | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true |

Assume Boolean variables A holds true and variable B holds false, then −

## Example

public class Test {

public static void main(String args[]) {

boolean a = true;

boolean b = false;

System.out.println("a && b = " + (a&&b));

System.out.println("a || b = " + (a||b) );

System.out.println("!(a && b) = " + !(a && b));

}

}

This will produce the following result −

## Output

a && b = false

a || b = true

!(a && b) = true

**The Assignment Operators**

Following are the assignment operators supported by Java language −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C – A |
| \*= | Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

## Example

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

int c = 0;

c = a + b;

System.out.println("c = a + b = " + c );

c += a ;

System.out.println("c += a = " + c );

c -= a ;

System.out.println("c -= a = " + c );

c \*= a ;

System.out.println("c \*= a = " + c );

a = 10;

c = 15;

c /= a ;

System.out.println("c /= a = " + c );

a = 10;

c = 15;

c %= a ;

System.out.println("c %= a = " + c );

c <<= 2 ;

System.out.println("c <<= 2 = " + c );

c >>= 2 ;

System.out.println("c >>= 2 = " + c );

c >>= 2 ;

System.out.println("c >>= 2 = " + c );

c &= a ;

System.out.println("c &= a = " + c );

c ^= a ;

System.out.println("c ^= a = " + c );

c |= a ;

System.out.println("c |= a = " + c );

}

}

This will produce the following result −

## Output

c = a + b = 30

c += a = 40

c -= a = 30

c \*= a = 300

c /= a = 1

c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

c >>= 2 = 1

c &= a = 0

c ^= a = 10

c |= a = 10

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator ( ? : )

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false

Following is an example −

**Example**

public class Test {

public static void main(String args[]) {

int a, b;

a = 10;

b = (a == 1) ? 20: 30;

System.out.println( "Value of b is : " + b );

b = (a == 10) ? 20: 30;

System.out.println( "Value of b is : " + b );

}

}

This will produce the following result −

**Output**

Value of b is : 30

Value of b is : 20

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as −

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example −

**Example**

public class Test {

public static void main(String args[]) {

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example −

**Example**

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]) {

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3 \* 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | >() [] . (dot operator) | Left toright |
| Unary | >++ - - ! ~ | Right to left |
| Multiplicative | >\* / | Left to right |
| Additive | >+ - | Left to right |
| Shift | >>> >>> << | Left to right |
| Relational | >> >= < <= | Left to right |
| Equality | >== != | Left to right |
| Bitwise AND | >& | Left to right |
| Bitwise XOR | >^ | Left to right |
| Bitwise OR | >| | Left to right |
| Logical AND | >&& | Left to right |
| Logical OR | >|| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | >= += -= \*= /= %= >>= <<= &= ^= |= | Right to left |

1. **Decision making**

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



ava programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [**if statement**](https://www.tutorialspoint.com/java/if_statement_in_java.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [**if...else statement**](https://www.tutorialspoint.com/java/if_else_statement_in_java.htm)  An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| 3 | [**nested if statement**](https://www.tutorialspoint.com/java/nested_if_statements_in_java.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| 4 | [**switch statement**](https://www.tutorialspoint.com/java/switch_statement_in_java.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |

**if statement**

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

Following is the syntax of an if statement −

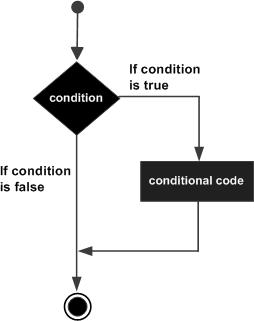
if(Boolean\_expression) {

// Statements will execute if the Boolean expression is true

}

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

public class Test {

public static void main(String args[]) {

int x = 10;

if( x < 20 ) {

System.out.print("This is if statement");

}

}

}

This will produce the following result −

Output

This is if statement.

[**if...else statement**](https://www.tutorialspoint.com/java/if_else_statement_in_java.htm)

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

Following is the syntax of an if...else statement −

if(Boolean\_expression) {

// Executes when the Boolean expression is true

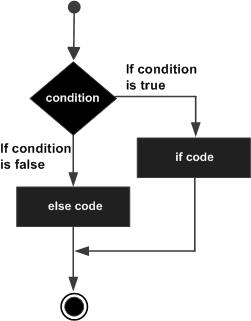
}else {

// Executes when the Boolean expression is false

}

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

public class Test {

public static void main(String args[]) {

int x = 30;

if( x < 20 ) {

System.out.print("This is if statement");

}else {

System.out.print("This is else statement");

}

}

}

This will produce the following result −

Output

This is else statement

The if...else if...else Statement

An if statement can be followed by an optional *else if...else*statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

Following is the syntax of an if...else statement −

if(Boolean\_expression 1) {

// Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2) {

// Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3) {

// Executes when the Boolean expression 3 is true

}else {

// Executes when the none of the above condition is true.

}

Example

public class Test {

public static void main(String args[]) {

int x = 30;

if( x == 10 ) {

System.out.print("Value of X is 10");

}else if( x == 20 ) {

System.out.print("Value of X is 20");

}else if( x == 30 ) {

System.out.print("Value of X is 30");

}else {

System.out.print("This is else statement");

}

}

}

This will produce the following result −

Output

Value of X is 30

# nested if statement in java

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

## Syntax

The syntax for a nested if...else is as follows −

if(Boolean\_expression 1) {

// Executes when the Boolean expression 1 is true

if(Boolean\_expression 2) {

// Executes when the Boolean expression 2 is true

}

}

You can nest **else if...else** in the similar way as we have nested *if* statement.

## Example

public class Test {

public static void main(String args[]) {

int x = 30;

int y = 10;

if( x == 30 ) {

if( y == 10 ) {

System.out.print("X = 30 and Y = 10");

}

}

}

}

This will produce the following result −

## Output

X = 30 and Y = 10

# switch statement in java

Advertisements

[Previous Page](https://www.tutorialspoint.com/java/java_decision_making.htm)

[Next Page](https://www.tutorialspoint.com/java/java_decision_making.htm)

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

## Syntax

The syntax of enhanced for loop is −

switch(expression) {

case value :

// Statements

break; // optional

case value :

// Statements

break; // optional

// You can have any number of case statements.

default : // Optional

// Statements

}

The following rules apply to a **switch** statement −

* The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
* When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will *fall through*to subsequent cases until a break is reached.
* A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

// char grade = args[0].charAt(0);

char grade = 'C';

switch(grade) {

case 'A' :

System.out.println("Excellent!");

break;

case 'B' :

case 'C' :

System.out.println("Well done");

break;

case 'D' :

System.out.println("You passed");

case 'F' :

System.out.println("Better try again");

break;

default :

System.out.println("Invalid grade");

}

System.out.println("Your grade is " + grade);

}

}

Compile and run the above program using various command line arguments. This will produce the following result −

## Output

Well done

Your grade is C

The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

To determine the value of the whole expression, initially exp1 is evaluated.

* If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
* If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

**Java Arrays**

* Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

**Java Enums**

* Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.
* With the use of enums it is possible to reduce the number of bugs in your code.
* For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.
* Example
* class FreshJuice {
* enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
* FreshJuiceSize size;
* }
* public class FreshJuiceTest {
* public static void main(String args[]) {
* FreshJuice juice = new FreshJuice();
* juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
* System.out.println("Size: " + juice.size);
* }
* }
* The above example will produce the following result −
* Output
* Size: MEDIUM
* **Note** − Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

**Inheritance**

* In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.
* This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

**Interfaces**

* In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.
* An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

# Java - Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop & Description** |
| 1 | [**while loop**](https://www.tutorialspoint.com/java/java_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [**for loop**](https://www.tutorialspoint.com/java/java_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [**do...while loop**](https://www.tutorialspoint.com/java/java_do_while_loop.htm)  Like a while statement, except that it tests the condition at the end of the loop body. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [**break statement**](https://www.tutorialspoint.com/java/java_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [**continue statement**](https://www.tutorialspoint.com/java/java_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

# while Loop in java

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

## Syntax

The syntax of a while loop is −

while(Boolean\_expression) {

// Statements

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

## Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

# do while loop in java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

## Syntax

Following is the syntax of a do...while loop −

do {

// Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int x = 10;

do {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}while( x < 20 );

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

# for loop in java

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

## Syntax

The syntax of a for loop is −

for(initialization; Boolean\_expression; update) {

// Statements

}

Here is the flow of control in a **for** loop −

* The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
* Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
* After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

## Flow Diagram



## Example

Following is an example code of the for loop in Java.

public class Test {

public static void main(String args[]) {

for(int x = 10; x < 20; x = x + 1) {

System.out.print("value of x : " + x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

## Enhanced for loop in Java

This is mainly used to traverse collection of elements including arrays.

### Syntax

Following is the syntax of enhanced for loop −

for(declaration : expression)   
{

// Statements

}

* **Declaration** − The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression** − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

### Example

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers )

{

System.out.print( x );

System.out.print(",");

}

System.out.print("\n");

String [] names = {"James", "Larry", "Tom", "Lacy"};

for( String name : names )

{

System.out.print( name );

System.out.print(",");

}

}

}

This will produce the following result −

### Output

10, 20, 30, 40, 50,

James, Larry, Tom, Lacy,

# Break statement in java

The **break** statement in Java programming language has the following two usages −

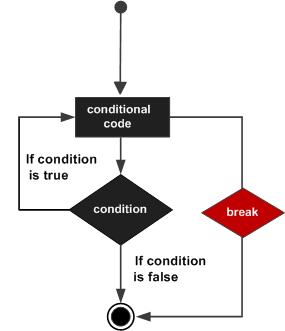
* When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
* It can be used to terminate a case in the **switch**statement (covered in the next chapter).

## Syntax

The syntax of a break is a single statement inside any loop −

break;

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers )

{

if( x == 30 )

{

break;

}

System.out.print( x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

10

20

# Continue statement in java

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

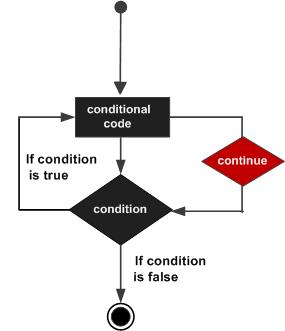
* In a for loop, the continue keyword causes control to immediately jump to the update statement.
* In a while loop or do/while loop, control immediately jumps to the Boolean expression.

## Syntax

The syntax of a continue is a single statement inside any loop −

continue;

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers ) {

if( x == 30 ) {

continue;

}

System.out.print( x );

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

10

20

40

50

GOTO STATEMENT IN JAVA WITH EXAMPLE

[3 comments](http://www.aboutcodes.com/2012/07/goto-statement-in-java-with-example.html#comment-form)

In java goto keyword is not used. In place of goto we used break in java.  
  
Why we using break in place of goto.  
goto provides a way to branch in an arbitrary and unstructured manner. This usually makes goto ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimization so break is used in place of break.  
  
**Syntax:-**  
  
break label;  
  
label can be any variable but not java keyword. label is followed by colon.  
  
**Execution:-**  
  
Label break statement is used to transfer control unconditionally from one place to another place in program.   
  
To use break as goto statement, we have to name a block, block are set of java statement. To name a block, put label at start of it.  
  
Example:-  
  
go:{   
System.out.println("Hey");  
.  
.  
block  
.  
.  
}  
  
Here go is label.  
  
Once you labeled a block, you can use this label as the target of break statement.   
When compiler see break statement with label than compiler transfer execution to at end of labeled block.  
  
Label break statement is used exit from nested loops.  
  
**Example:-**  
  
Program to find sum of N number

import java.util.Scanner;  
  
class group{  
public static void main(String arng[]){  
int num, i,sum=0;  
go:{  
Scanner data = new Scanner(System.in);  
System.out.println("Enter a number");  
num=data.nextInt();  
for(i=0;i<100;i++)  
{  
sum=sum+i;  
if(i==num)  
break go;  
}  
}  
System.out.println("Sum of odd number:"+sum);  
}  
}  
  
 Here go is label of break statement.  
 Following block is go labeled block.  
 go:{  
Scanner data = new Scanner(System.in);  
System.out.println("Enter a number");  
num=data.nextInt();  
for(i=0;i<100;i++)  
{  
sum=sum+i;  
if(i==num)  
break go;  
}  
}

# Java Array

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

**Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



### Advantage of Java Array

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

### Disadvantage of Java Array

* **Size Limit:** We can store only 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.

An array can be either primitive or reference type. It gets memory in heap area. Index of array starts from zero to size-1.

#### Features of Array

* It is always indexed. Index begins from 0.
* It is a collection of similar data types.
* It occupies a contiguous memory location.

#### Array Declaration

**Syntax :**

datatype[ ] *identifier*;

or

**datatype** identifier[ ];

Both are valid syntax for array declaration. But the former is more readable.

**Example :**

int[ ] arr;

char[ ] arr;

short[ ] arr;

long[ ] arr;

int[ ][ ] arr; // two dimensional array.

#### Initialization of Array

new operator is used to initialize an array.

**Example :**

int[ ] *arr* = new int[10]; //this creates an empty array named arr of integer type whose size is 10.

or

int[ ] *arr* = {10,20,30,40,50}; //this creates an array named arr whose elements are given.

#### Accessing array element

As mention ealier array index starts from 0. To access nth element of an array. Syntax

arrayname[n-1];

*Example :* To access 4th element of a given array

int[ ] arr = {10,20,30,40};

System.out.println("Element at 4th place" + **arr**[3]);

The above code will print the 4th element of array arr on console.

**Note:** To find the length of an array, we can use the following syntax: array\_name.length. There are no braces infront of length. Its not length().

#### foreach or enhanced for loop

J2SE 5 introduces special type of for loop called foreach loop to access elements of array. Using foreach loop you can access complete array sequentially without using index of array. Let us see an example of foreach loop.

class **Test**

{

public static void main(String[] args)

{

int[] **arr** = {10, 20, 30, 40};

for(int *x* : **arr**)

{

System.out.println(x);

}

}

}

**Output :**

10

20

30

40

### Types of Array in java

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

### Single Dimensional Array in java

### Syntax to Declare an Array in java

1. dataType[] arr; (or)
2. dataType arr[];

### Instantiation of an Array in java

1. arrayRefVar=new datatype[size];

### Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

class Testarray

{

public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//printing array

for(int i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

Output: 10

20

70

40

50

## Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. int a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

class Testarray1{

public static void main(String args[]){

  int a[]={33,3,4,5};//declaration, instantiation and initialization

  //printing array

for(int i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

Output:33

3

4

5

### Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

class Testarray2

{

static void min(int arr[])

{

int small=arr[0];

for(int i=1;i<arr.length;i++)

 if(small>arr[i])

  small=arr[i];

  System.out.println(small);

}

public static void main(String args[])

{

  int a[]={33,3,4,5};

min(a);//passing array to method

 }

}

Output:3

### Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

### Syntax to Declare Multidimensional Array in java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

### Example to instantiate Multidimensional Array in java

1. int[][] arr=new int[3][3];//3 row and 3 column

### Example to initialize Multidimensional Array in java

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

### Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

class Testarray3{

public static void main(String args[]){

//declaring and initializing 2D array

int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array

for(int i=0;i<3;i++){

 for(int j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

Output:1 2 3

2 4 5

4 4 5

### What is the class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

class Testarray4{

public static void main(String args[]){

int arr[]={4,4,5};

Class c=arr.getClass();

String name=c.getName();

System.out.println(name);

}}

Output:I

### Copying a java array

We can copy an array to another by the arraycopy method of System class.

### Syntax of arraycopy method

public static void arraycopy(  Object src, int srcPos,Object dest, int destPos, int length  )

### Example of arraycopy method

class TestArrayCopyDemo

{

    public static void main(String[] args) {

        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };

        char[] copyTo = new char[7];

          System.arraycopy(copyFrom, 2, copyTo, 0, 7);

        System.out.println(new String(copyTo));

    }

}

Output:caffein

### Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

class Testarray5{

public static void main(String args[]){

//creating two matrices

int a[][]={{1,3,4},{3,4,5}};

int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices

int c[][]=new int[2][3];

//adding and printing addition of 2 matrices

for(int i=0;i<2;i++){

for(int j=0;j<3;j++){

c[i][j]=a[i][j]+b[i][j];

System.out.print(c[i][j]+" ");

}

System.out.println();//new line

}

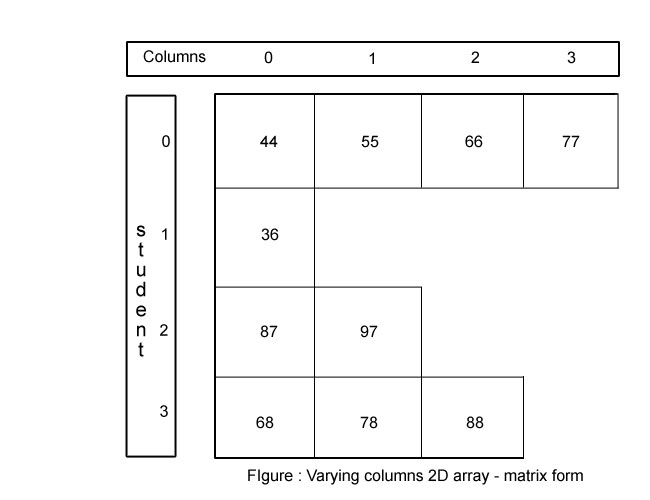
}}

Output:2 6 8

6 8 10

### Jagged Array

Jagged means to have an uneven edge or surface. In java, a jagged array means to have a multi-dimensional array with uneven size of rows in it.



#### Initialization of Jagged Array

new operator is used to initialize an array.

**Example:**

int[ ][ ] arr = new int[3][ ]; //there will be 10 arrays whose size is variable

arr[0] = new int[3];

arr[1] = new int[4];

arr[2] = new int[5];

# Jagged Array in Java

Prerequisite : [Arrays in Java](http://www.geeksforgeeks.org/arrays-in-java/)

[Jagged array](https://en.wikipedia.org/wiki/Jagged_array) is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row. These type of arrays are also known as Jagged arrays.

Following are Java programs to demonstrate the above concept.

|  |
| --- |
| // Program to demonstrate 2-D jagged array in Java  class Main  {      public static void main(String[] args)      {          // Declaring 2-D array with 2 rows          int arr[][] = new int[2][];            // Making the above array Jagged            // First row has 3 columns          arr[0] = new int[3];            // Second row has 2 columns          arr[1] = new int[2];          // Initializing array          int count = 0;          for (int i=0; i<arr.length; i++)              for(int j=0; j<arr[i].length; j++)                  arr[i][j] = count++;            // Displaying the values of 2D Jagged array          System.out.println("Contents of 2D Jagged Array");          for (int i=0; i<arr.length; i++)          {              for (int j=0; j<arr[i].length; j++)                  System.out.print(arr[i][j] + " ");              System.out.println();          }      }  } |

Output:

Contents of 2D Jagged Array

0 1 2

3 4

class Main

{

    public static void main(String[] args)

    {

        // Declaring 2-D array with 2 rows

        int arr[][] = new int[4][];

        // Making the above array Jagged

        // First row has 3 columns

        arr[0] = new int[1];

        // Second row has 2 columns

        arr[1] = new int[2];

        arr[2] = new int[3];

        arr[3] = new int[4];

        // Initializing array

       // int count = 1;

     /\*   for (int i=0; i<arr.length; i++)

            for(int j=0; j<arr[i].length; j++)

                arr[i][j] = count++;\*/

        // Displaying the values of 2D Jagged array

        System.out.println("Contents of 2D Jagged Array");

        for (int i=0; i<arr.length; i++)

        {

            for (int j=0; j<arr[i].length; j++)

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

System.out.print(“\*”);

            System.out.println();

        }

    }

}

Following is another example where i’th row has i columns, i.e., first row has 1 element, second row has two elements and so on.

|  |
| --- |
| // Another Java program to demonstrate 2-D jagged  // array such that first row has 1 element, second  // row has two elements and so on.  class Main  {      public static void main(String[] args)      {          int r = 5;            // Declaring 2-D array with 5 rows          int arr[][] = new int[r][];            // Creating a 2D array such that first row          // has 1 element, second row has two          // elements and so on.          for (int i=0; i<arr.length; i++)              arr[i] = new int[i+1];            // Initializing array          int count = 0;          for (int i=0; i<arr.length; i++)              for(int j=0; j<arr[i].length; j++)                  arr[i][j] = count++;            // Displaying the values of 2D Jagged array          System.out.println("Contents of 2D Jagged Array");          for (int i=0; i<arr.length; i++)          {              for (int j=0; j<arr[i].length; j++)                  System.out.print(arr[i][j] + " ");              System.out.println();          }      }  } |

Output:

Contents of 2D Jagged Array

0

1 2

3 4 5

6 7 8 9

10 11 12 13 14

{

public static void main(String args[])

{

   int student[][] = new int[4][];

   student[0] = new int[4];

   student[1] = new int[1];

   student[2] = new int[2];

   student[3] = new int[3];

   System.out.println("Row count: " + student.length);

   System.out.println("Third row size: : " + student[3].length);

                             // 1st row

   student[0][0] = 44;

   student[0][1] = 55;

   student[0][2] = 66;

   student[0][3] = 77;

             // 2nd row

   student[1][0] = 36;

             // 3rd row

   student[2][0] = 87;

   student[2][1] = 97;

             // 4th row

   student[3][0] = 68;

   student[3][1] = 78;

   student[3][2] = 88;

   System.out.println("student[3][1] marks: " + student[3][1]);

   System.out.println("\nMatrix Form");

   for(int i = 0; i < student.length; i++)

   {

     for(int j = 0; j < student[i].length; j++)

     {

       System.out.print(student[i][j] + "\t");

     }

     System.out.println();

   }

}

}

# 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:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=**new** String(ch);

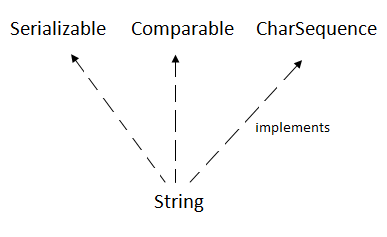
or

1. String s="javatpoint";

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

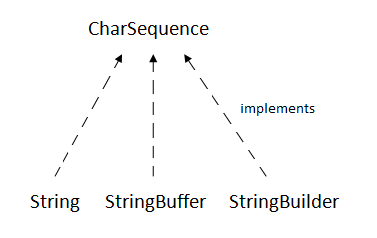
The java.lang.String class

implements *Serializable*, *Comparable* and *CharSequence* interfaces.



## CharSequence Interface

The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.



The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

We will discuss about immutable string later. Let's first understand what is string in java and how to create the string object.

### What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

### How to create String object?

|  |
| --- |
| There are two ways to create String object:   1. By string literal 2. By new keyword |

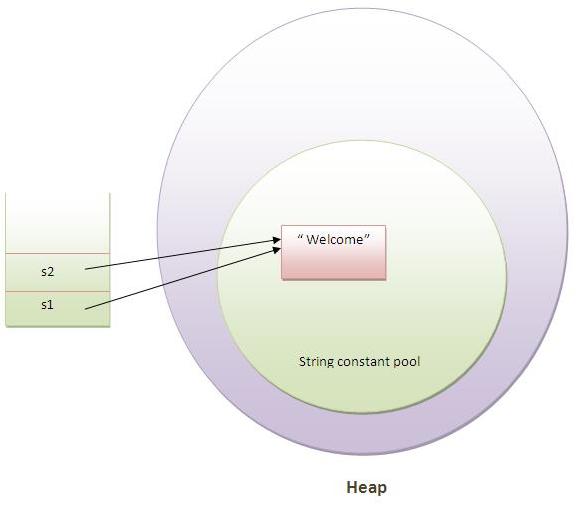
### 1) String Literal

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

1. String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

#### Note: String objects are stored in a special memory area known as string constant pool.

### Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

### 2) By new keyword

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

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

### 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);

}}

### 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)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatinates specified string |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns interned string |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. |

**1) Declaring a Java String array with an initial size**

If you know up front how large your array needs to be, you can (a) declare a String array, and (b) give it an initial size, like this:

public class JavaStringArrayTests

{

private String[] toppings = new String[20];

// more ...

}

In that example, I *declare* a String array named toppings, and then give it an initial size of 20 elements.

Later on, in a Java method in your class, you can populate the elements in the array like this:

private void populateStringArray()

{

toppings[0] = "Cheese";

toppings[1] = "Pepperoni";

toppings[2] = "Black Olives";

// ...

}

As this shows, Java arrays begins with an element numbered zero — they are *zero-based* — just like the C programming language.

**2) Declare a Java String array with no initial size**

If you know you need a String array, but don’t initially know how large the array needs to be, you can also declare an array variable without giving it an initial size, like this:

public class JavaStringArrayTests

{

private String[] toppings;

// more ...

}

Then somewhere later in your code you can (a) give the array a size and then (b) populate it as desired, like this:

private void populateStringArray()

{

// you'll probably determine this size based on some algorithm

toppings = new String[20];

toppings[0] = "Cheese";

toppings[1] = "Pepperoni";

toppings[2] = "Black Olives";

// ...

}

This array programming approach is very similar to the previous approach, but as you can see, I don't give the array a size until the populateStringArray method is called.

**3) Declaring and populating a Java String array**

You don’t have to declare a String array in two steps, you can do everything in one step, like this:

public class JavaStringArrayTests

{

private String[] toppings = {"Cheese", "Pepperoni", "Black Olives"};

}

This example is similar to the previous example, with the following differences:

1. The toppings array is defined and populated in one step.
2. This toppings array has only three elements in it.

**4) Iterating through a String array: Before Java 5**

Before *Java 5*, the way to loop through an array involved (a) getting the number of elements in the array, and then (b) looping through the array with a for loop. Here’s a complete source code example that demonstrates the syntax *prior* to Java 5:

public class JavaStringArrayTests1

{

private String[] toppings = {"Cheese", "Pepperoni", "Black Olives"};

// our constructor; print out the String array here

public JavaStringArrayTests1()

{

// old `for` loop

int size = toppings.length;

for (int i=0; i<size; i++)

{

System.out.println(toppings[i]);

}

}

// main kicks everything off.

// create a new instance of our class here.

public static void main(String[] args)

{

new JavaStringArrayTests1();

}

}

**5) Iterating through a String array: After Java 5**

With the advent of Java 5, you can make your for loops a little cleaner and easier to read, so looping through an array is even easier. Here’s a complete source code example that demonstrates the Java 5 syntax:

public class JavaStringArrayTests2

{

private String[] toppings = {"Cheese", "Pepperoni", "Black Olives"};

// our constructor; print out the String array here

public JavaStringArrayTests2()

{

// new `for` loop

for (String s: toppings)

{

System.out.println(s);

}

}

// main kicks everything off.

// create a new instance of our class here.

public static void main(String[] args)

{

new JavaStringArrayTests2();

}

}

I think you’ll agree that the Java 5 syntax for looping through an array is more concise, and easier to read.

## [Java String Array Examples](http://javadevnotes.com/java-string-array-examples)

A Java String Array is an object that holds a fixed number of String values. Arrays in general is a very useful and important data structure that can help solve many types of problems. It's simplicity on how to access contents through index makes it powerful yet user-friendly. Here are some examples on how to use String Array in Java. 

### String Array Declaration

Square brackets is used to declare a String array. There are two ways of using it. The first one is to put square brackets after the String reserved word. For example: 

String[] thisIsAStringArray;

Another way of declaring a String Array is to put the square brackers after the name of the variable. For example: 

String thisIsAStringArray[];

Both statements will declare the variable "thisIsAStringArray" to be a String Array. Note that this is just a declaration, the variable "thisIsAStringArray" will have the value null. And since there is only one square brackets, this means that the variable is only a one-dimensional String Array. Examples will be shown later on how to declare multi-dimensional String Arrays. 

### String Array Declaration With Initial Size

Arrays are usually used when we know how many objects are needed. Hence, arrays are usually declared with an initial size. Here is an example: 

String[] thisIsAStringArray = new String[5];

This will declare a String Array with 5 elements. Each element can be accessed using index that starts with 0. The fist element is "thisIsAStringArray[0]", the second element is "thisIsAStringArray[1]", and so on. Here is an example of declaring a String Array with size 5 and assigning values to each element: 

String[] thisIsAStringArray = new String[5];

thisIsAStringArray[0] = "AAA";

thisIsAStringArray[1] = "BBB";

thisIsAStringArray[2] = "CCC";

thisIsAStringArray[3] = "DDD";

thisIsAStringArray[4] = "EEE";

Note, since there are only 5 elements and index started from 0, the last index will be 4. Or we could use the formula ( array length - 1). This means if we access the index greater than 4, an Exception will be raised. Example:

String[] thisIsAStringArray = new String[5];

thisIsAStringArray[5] = "FFF";

The code will throw a java.lang.ArrayIndexOutOfBoundsException. 

### String Array Initialization on Declaration

There are multiple ways to initialize a String Array. Initialization can also be done at the same time as the declaration. Here is an example: 

String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};

This will create a String Array of length 5. Element at index 0 will have the value "AAA", element at index 1 will have the value "BBB", and so on. Hence, when we run this code: 

String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};

System.out.println( thisIsAStringArray[0] );

System.out.println( thisIsAStringArray[1] );

System.out.println( thisIsAStringArray[2] );

System.out.println( thisIsAStringArray[3] );

System.out.println( thisIsAStringArray[4] );

It will produce this output:

AAA

BBB

CCC

DDD

EEE

Another syntax to declare and initialize a String array together is by using the new operator. Here is an example:

String[] thisIsAStringArray = new String[]{"AAA", "BBB", "CCC", "DDD", "EEE"};

The behaviour is the same as the example above: *"String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};"* 

### String Array Initialization After Declaration

The code for initializing an array can be separated from the declaration code. This is useful when values are not known during declaration. Here is an example: 

String[] thisIsAStringArray;

if (fruits) {

thisIsAStringArray = new String[] {"Apple", "Banana", "Orange"};

} else {

thisIsAStringArray = new String[] {"Asparagus", "Carrot", "Tomato"};

}

Note that the value of the array depends on the value of fruits variable. The array can have the values *{"Apple", "Banana", "Orange"}* if fruits is true, or *{"Asparagus", "Carrot", "Tomato"}* if fruits is false.

Note that initializing an array will override the old contents of the array. For example

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

thisIsAStringArray = new String[] {"Asparagus", "Carrot", "Tomato"};

System.out.println( thisIsAStringArray[0] );

System.out.println( thisIsAStringArray[1] );

System.out.println( thisIsAStringArray[2] );

The code will have the output below. This is because the old contents *{"Apple", "Banana", "Orange"}*, will be discarded and replaced with the new contents.

Asparagus

Carrot

Tomato

Also note that even the size of array will be changed if re-initialized. For example: 

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

thisIsAStringArray = new String[] {"Asparagus", "Carrot"};

System.out.println( thisIsAStringArray[0] );

System.out.println( thisIsAStringArray[1] );

System.out.println( thisIsAStringArray[2] );

Will have the following output:

Asparagus

Carrot

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2

at ArrayTest.main(ArrayTest.java:10)

The exception is because the thisIsAStringArray variable will only have 2 elements after the second initialization. 

### String Array Length/Size

The property length of a String Array can be used to determine the number of elements in an Array. Here is an example usage: 

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

System.out.println( thisIsAStringArray.length );

This will have the output: 

3

Because there are 3 elements in the declared array. 

### Iterate Through String Array

Since the size and contents of a String Array can vary, it is useful to iterate through all the values. Here is an example code using style that can run prior to Java 5.

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

for( int i = 0; i < thisIsAStringArray.length; i++)

{

String element = thisIsAStringArray[i];

System.out.println( element );

}

The code will start from index 0, and continue upto *length - 1*, which is the last element of the array. This code will run on any version of Java, before of after Java 5

The code will have the output:

Apple

Banana

Orange

Another way is to use the enhanced for loop of Java 5. For example:

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

for( String element:thisIsAStringArray)

{

System.out.println( element );

}

The code will have the same output as the previous example: 

### Test if String Array Contains a Value

When we have an array of Strings, it is usually common to have a scenario where we wish to know if the array contains a specific value. One way to do it is to have custom code: 

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

String stringToSearch = "Banana";

boolean found = false;

for (String element:thisIsAStringArray) {

if ( element.equals( stringToSearch )) {

found = true;

}

}

if (found) {

System.out.println( "The array contains the string: " + stringToSearch );

} else {

System.out.println( "The array does not contains the string: " + stringToSearch );

}

This code will loop through the array and check one by one if an element is equal to the item being searched for. The output of the code will be:

The array contains the string: Banana

We can place a break when we found the item, to make it run faster.

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

String stringToSearch = "Banana";

boolean found = false;

for (String element:thisIsAStringArray) {

if ( element.equals( stringToSearch )) {

found = true;

break;

}

}

if (found) {

System.out.println( "The array contains the string: " + stringToSearch );

} else {

System.out.println( "The array does not contains the string: " + stringToSearch );

}

The break will stop the loop when we found an element match. We can also factor this out as a separate method, if this will be a commonly used routine:

public static boolean contains(String[] stringArray, String stringToSearch)

{

boolean result = false;

for (String element:stringArray) {

if ( element.equals( stringToSearch )) {

result = true;

break;

}

}

return result;

}

public static void main( String[] args )

{

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

System.out.println( "The array contains the string Apple: " + contains(thisIsAStringArray, "Apple") );

System.out.println( "The array contains the string Carrots: " + contains(thisIsAStringArray, "Carrots") );

}

As shown, the function can be called multiple times because it is re-factored as a utility method. Here is the output of the code:

The array contains the string Apple: true

The array contains the string Carrots: false

### Add element to String Array

If you have a String array with length 5 and want to add a 6th element, this is not possible because the size of an array is fixed when created. only possible way is to re-create a new String Array with larger capacity to be able to put the new value at the end. Here is an example code: 

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

String[] tempArray = new String[ thisIsAStringArray.length + 1 ];

for (int i=0; i<thisIsAStringArray.length; i++)

{

tempArray[i] = thisIsAStringArray[i];

}

tempArray[thisIsAStringArray.length] = "Carrots";

thisIsAStringArray = tempArray;

for (String element:thisIsAStringArray) {

System.out.println( element );

}

The example code above started with a String Array of size 3. Another array was created with size larger than 1. The 3 elements were then copied and the new value is added on the last index. The code will print the following result: 

Apple

Banana

Orange

Carrots

We can refactor the logic to a separate function. For example:

public static String[] add(String[] stringArray, String newValue)

{

String[] tempArray = new String[ stringArray.length + 1 ];

for (int i=0; i<stringArray.length; i++)

{

tempArray[i] = stringArray[i];

}

tempArray[stringArray.length] = newValue;

return tempArray;

}

public static void main( String[] args )

{

String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};

thisIsAStringArray = add(thisIsAStringArray, "Carrots");

for (String element:thisIsAStringArray) {

System.out.println( element );

}

}

This will have the same output as the previous example. 

### Sort String Array

Another common situation in Java is the need to sort elements of an array. One way to do this is to code our own routine based on some well known algorithm. Here is an example of how to Sort String Array in Java using Bubble Sort: 

public static void bubbleSort( String[] arr ) {

int j = 0;

String tmp;

boolean sorted = false;

while ( !sorted ) {

sorted = true;

j++;

for ( int i = 0; i < arr.length - j; i++ ) {

if (arr[i].compareTo( arr[i + 1] ) > 0) {

tmp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = tmp;

sorted = false;

}

}

}

}

public static void main( String[] args ) {

String[] thisIsAStringArray = { "Banana", "Orange", "Cherry", "Apple", "Pineapple", "Melon", "Plum" };

bubbleSort( thisIsAStringArray );

for ( String element : thisIsAStringArray ) {

System.out.println( element );

}

}

The code will output the Strings in the array in order alphabetically.

Apple

Banana

Cherry

Melon

Orange

Pineapple

Plum

But since sorting is a common programming problem, Java has a built-in solution for this. We can use the sort method in java.util.Arrays class. Here is a shortened example of our code above using Arrays.sort() method: 

public static void main( String[] args ) {

String[] thisIsAStringArray = { "Banana", "Orange", "Cherry", "Apple", "Pineapple", "Melon", "Plum" };

Arrays.sort(thisIsAStringArray );

for ( String element : thisIsAStringArray ) {

System.out.println( element );

}

}

And the output will be the same, the alphabetically sorted strings in the array. 

### Convert String Array to a String

For display purposes, it is sometimes required to convert a String Array to a String. The utility class java.util.Arrays has a convenience method for this. Here is an example code for the conversion: 

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

String theString = Arrays.toString( thisIsAStringArray );

System.out.println( theString );

And this will be the output:

[Apple, Banana, Orange]

The elements are separated by comma in the converted String, and enclosed in square brackets. To implement a custom behaviour, we may implement our own code for that:

We may implement our own that uses a custom delimiter without the square brackets. Here is an example of converting String Array to String with custom delimiter:

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

String delimiter = "-";

StringBuilder sb = new StringBuilder();

for ( String element : thisIsAStringArray ) {

if (sb.length() > 0) {

sb.append( delimiter );

}

sb.append( element );

}

String theString = sb.toString();

System.out.println( theString );

The output will use the delimiter dash without square brackets:

Apple-Banana-Orange

We can refactor the logic to a separate method. Here is an example of converting a String Array to String using comma delimiter or any other delimiter: 

public static String stringArrayToString( String[] stringArray, String delimiter ) {

StringBuilder sb = new StringBuilder();

for ( String element : stringArray ) {

if (sb.length() > 0) {

sb.append( delimiter );

}

sb.append( element );

}

return sb.toString();

}

public static void main( String[] args ) {

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

String theString = stringArrayToString(thisIsAStringArray, ",");

System.out.println( theString );

String theString2 = stringArrayToString(thisIsAStringArray, "=");

System.out.println( theString2 );

}

The output will be:

Apple,Banana,Orange

Apple=Banana=Orange

### Convert String Array to List

One dis-advantage of arrays is that the size is fixed. If we need the size of the array to grow, it is better to use List. Fortunately, this is simple - again by using java.util.Arrays. Here is an example code:

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

List<String> stringList = Arrays.asList( thisIsAStringArray );

Note that reading the Javadoc of asList says:

*Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs, in combination with Collection.toArray. The returned list is serializable and implements RandomAccess.*

This means that you can not add items to the List returned by Arrays.asList method. Hence, this code will raise an Exception:

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

List<String> stringList = Arrays.asList( thisIsAStringArray );

stringList.add( "WaterMelon" );

When you run this, you will encounter a *java.lang.UnsupportedOperationException* on the line where "WaterMelon" was being added to the list.

To avoid this problem, we can specifically convert the String Array to an ArrayList. Here is a sample code on how to do that:

String[] thisIsAStringArray = { "Apple", "Banana", "Orange" };

List<String> fixedList = Arrays.asList( thisIsAStringArray );

List<String> stringList = new ArrayList<String>( fixedList );

stringList.add( "WaterMelon" );

The code will construct a new ArrayList based on the returned value of Arrays.asList. All items from fixedList will be added to the stringList. And the ArrayList instance has no limitation of being fixed size. Hence, the code that adds "WaterMelon" will not throw any exception. 

### Convert String Array to Set

The difference between a Set and a list is that a Set cannot contain duplicate elements while a List can. So if we only want elements of a collection without duplicates, Set is a more appropriate data structure. Here is a sample code on how to convert a String Array to a Set:

String[] thisIsAStringArray = { "Apple", "Banana", "Orange", "Banana" };

List<String> stringList = Arrays.asList( thisIsAStringArray );

Set<String> stringSet = new HashSet<String>( stringList );

System.out.println( "Size of the list is: " + stringList.size() );

System.out.println( "Size of the set is: " + stringSet.size() );

The output will be: 

Size of the list is: 4

Size of the set is: 3

The list will have four items, as enumerated in declaration. But since there are two Banana's, the set will only have 3 elements. 

### Convert list to String Array

If we want the opposite behavior of the sample above, it is also possible to convert a List back to String Array. Here is a sample code that converts a List to String Array:

List<String> stringList = new ArrayList<String>();

stringList.add( "Apple" );

stringList.add( "Banana" );

stringList.add( "Orange" );

String[] stringArr = stringList.toArray( new String[] {} );

for ( String element : stringArr ) {

System.out.println( element );

}

Note that we need to pass an instance of array to the *toArray* method. This is to tell the list what type of array it should return. 

### Two Dimensional String Array in Java

In Java, it is possible to declare an array of arrays - or simply a two-dimensional arrays. This is very useful for storing more complex information. For example, a collection of couples (husband and wife). A couple is composed of two information, the name of the husband, and the name of the wife. If we want to represent an array of couples, then two-dimensional String Array can be used. Here is an example code: 

String[][] coupleArray = new String[4][];

coupleArray[0] = new String[] {"John", "Rose"};

coupleArray[1] = new String[] {"Peter", "Abigail"};

coupleArray[2] = new String[] {"Robert", "Josephine"};

coupleArray[3] = new String[] {"Timothy", "Angelina"};

Here, we declared an array of arrays with size 4. We initialize each element with a String Array that represents a pair of husband and wife names.

To access a specific item, we need two index values. For example, to output "Robert", the code should be:

System.out.println( coupleArray[2][0] );

And to access "Josephine", the code should be:

System.out.println( coupleArray[2][1] );

Here is an alternative way of creating a two-dimensional String array and initializing it's values: 

String[][] coupleArray = new String[4][2];

coupleArray[0][0] = "John";

coupleArray[0][1] = "Rose";

coupleArray[1][0] = "Peter";

coupleArray[1][1] = "Abigail";

coupleArray[2][0] = "Robert";

coupleArray[2][1] = "Josephine";

coupleArray[3][0] = "Timothy";

coupleArray[3][1] = "Angelina";

Here, we can think of it as creating a multi-dimensional array where the first dimension is of size 4, and the second dimension is of size 2. The effect is the same as the first example. Which means accessing elements is also the same.

public class ReplaceExample1{

public static void main(String args[]){

String s1="Java Programming is good for developer";

String replaceString=s1.replace('o','e');//replaces all occurrences of 'o' to 'e'

System.out.println(s1);

System.out.println(replaceString);

}

}

public class StringJoinExample

{

public static void main(String args[])

{

String  joinString1=String.join("-","welcome","to","javaprogram");

System.out.println(joinString1);

}

}

1. public class CharAtExample{
2. public static void main(String args[]){
3. String name="weolcome";
4. char ch=name.charAt(4);//returns the char value at the 4th index
5. System.out.println(ch);
6. }}

  public class SplitExample{

public static void main(String args[]){

String s1="java string split method by javatpoint";

String[] words=s1.split("\\s ");//splits the string based on string delimeter

//or

//String[] words=s1.split(" ");//splits the string based on string

//using java foreach loop to print elements of string array

for(String w:words){

System.out.println(w);

}

}}

public class InternExample{

public static void main(String args[]){

String s1=new String("hello");

String s2="hello";

String s4=”hello”;

String s3=s1.intern();//returns string from pool, now it will be same as s2

System.out.println(s2==s4);//

System.out.println(s1==s2);//false because reference is different

System.out.println(s2==s3);//true because reference is same

}}

public class IndexOfExample{

public static void main(String args[]){

String s1="this is index of example";

//passing substring

int index1=s1.indexOf("t");//returns the index of is substring

System.out.println(index1);

int index2=s1.indexOf("index");//returns the index of index substring

System.out.println(index2);

System.out.println(index1+" "+index2);//2 8

//passing substring with from index

int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index

System.out.println(index3);//5 i.e. the index of another is

public class IndexOfExample{

public static void main(String args[]){

String s1="this is index of example";

//passing substring

int index1=s1.indexOf("is");//returns the index of is substring

int index2=s1.indexOf("index");//returns the index of index substring

System.out.println(index1+"  "+index2);//2 8

  //passing substring with from index

int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index

System.out.println(index3);//5 i.e. the index of another is

//passing char value

int index4=s1.indexOf('s');//returns the index of s char value

System.out.println(index4);//3

}}

**public class FormatExample{**

**public static void main(String args[]){**

**String name="sonoo";**

**String sf1=String.format("name is %s",name);**

**String sf2=String.format("value is %f",32.33434);**

**String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part**

**System.out.println(sf1);**

**System.out.println(sf2);**

**System.out.println(sf3);**

**}}**

# 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.

#### Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

### 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. |

### Important methods of StringBuffer class

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | is used to return the character at the specified position. |
| public int | length() | is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

# Wrapper class in Java

**Wrapper class in java** provides the mechanism to convert primitive into object and object into primitive.

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

The eight classes of java.lang package are known as wrapper classes in java. The list of eight 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 |

## Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements −

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Method & Description** |  |
| 1 | [**xxxValue()**](https://www.tutorialspoint.com/java/number_xxxvalue.htm)  Converts the value of *this* Number object to the xxx data type and returns it. | byte byteValue()  short shortValue()  int intValue()  long longValue()  float floatValue()  double doubleValue()  **Parameters**  Here is the detail of parameters −   * All these are default methods and accepts no parameter.   **Return Value**   * This method returns the primitive data type that is given in the signature.   **Example**  public class Test {  public static void main(String args[]) {  Integer x = 5;    // Returns byte primitive data type  System.out.println( x.byteValue() );  // Returns double primitive data type  System.out.println(x.doubleValue());  // Returns long primitive data type  System.out.println( x.longValue() );  }  }  This will produce the following result −  **Output**  5  5.0  5 |
| 2 | [**compareTo()**](https://www.tutorialspoint.com/java/number_compareto.htm)  Compares *this* Number object to the argument. |  |
| 3 | [**equals()**](https://www.tutorialspoint.com/java/number_equals.htm)  Determines whether *this* number object is equal to the argument. |  |
| 4 | [**valueOf()**](https://www.tutorialspoint.com/java/number_valueof.htm)  Returns an Integer object holding the value of the specified primitive. |  |
| 5 | [**toString()**](https://www.tutorialspoint.com/java/number_tostring.htm)  Returns a String object representing the value of a specified int or Integer. |  |
| 6 | [**parseInt()**](https://www.tutorialspoint.com/java/number_parseint.htm)  This method is used to get the primitive data type of a certain String. | public class Test {  public static void main(String args[]) {  int x =Integer.parseInt("9");  double c = Double.parseDouble("5");  int b = Integer.parseInt("444",16);  System.out.println(x);  System.out.println(c);  System.out.println(b);  }  }  This will produce the following result −  **Output**  9  5.0  1092 |
| 7 | [**abs()**](https://www.tutorialspoint.com/java/number_abs.htm)  Returns the absolute value of the argument. |  |
| 8 | [**ceil()**](https://www.tutorialspoint.com/java/number_ceil.htm)  Returns the smallest integer that is greater than or equal to the argument. Returned as a double. |  |
| 9 | [**floor()**](https://www.tutorialspoint.com/java/number_floor.htm)  Returns the largest integer that is less than or equal to the argument. Returned as a double. |  |
| 10 | [**rint()**](https://www.tutorialspoint.com/java/number_rint.htm)  Returns the integer that is closest in value to the argument. Returned as a double. |  |
| 11 | [**round()**](https://www.tutorialspoint.com/java/number_round.htm)  Returns the closest long or int, as indicated by the method's return type to the argument. |  |
| 12 | [**min()**](https://www.tutorialspoint.com/java/number_min.htm)  Returns the smaller of the two arguments. |  |
| 13 | [**max()**](https://www.tutorialspoint.com/java/number_max.htm)  Returns the larger of the two arguments. |  |
| 14 | [**exp()**](https://www.tutorialspoint.com/java/number_exp.htm)  Returns the base of the natural logarithms, e, to the power of the argument. |  |
| 15 | [**log()**](https://www.tutorialspoint.com/java/number_log.htm)  Returns the natural logarithm of the argument. |  |
| 16 | [**pow()**](https://www.tutorialspoint.com/java/number_pow.htm)  Returns the value of the first argument raised to the power of the second argument. |  |
| 17 | [**sqrt()**](https://www.tutorialspoint.com/java/number_sqrt.htm)  Returns the square root of the argument. |  |
| 18 | [**sin()**](https://www.tutorialspoint.com/java/number_sin.htm)  Returns the sine of the specified double value. |  |
| 19 | [**cos()**](https://www.tutorialspoint.com/java/number_cos.htm)  Returns the cosine of the specified double value. |  |
| 20 | [**tan()**](https://www.tutorialspoint.com/java/number_tan.htm)  Returns the tangent of the specified double value. |  |
| 21 | [**asin()**](https://www.tutorialspoint.com/java/number_asin.htm)  Returns the arcsine of the specified double value. |  |
| 22 | [**acos()**](https://www.tutorialspoint.com/java/number_acos.htm)  Returns the arccosine of the specified double value. |  |
| 23 | [**atan()**](https://www.tutorialspoint.com/java/number_atan.htm)  Returns the arctangent of the specified double value. |  |
| 24 | [**atan2()**](https://www.tutorialspoint.com/java/number_atan2.htm)  Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta. |  |
| 25 | [**toDegrees()**](https://www.tutorialspoint.com/java/number_todegrees.htm)  Converts the argument to degrees. |  |
| 26 | [**toRadians()**](https://www.tutorialspoint.com/java/number_toradians.htm)  Converts the argument to radians. |  |
| 27 | [**random()**](https://www.tutorialspoint.com/java/number_random.htm)  Returns a random number. |  |

## Wrapper class Example: Primitive to Wrapper

1. **public** **class** WrapperExample1{
2. **public** **static** **void** main(String args[]){
3. //Converting int into Integer
4. **int** a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

20 20 20

## Wrapper class Example: Wrapper to Primitive

1. **public** **class** WrapperExample2{
2. **public** **static** **void** main(String args[]){
3. //Converting Integer to int
4. Integer a=**new** Integer(3);
5. **int** i=a.intValue();//converting Integer to int
6. **int** j=a;//unboxing, now compiler will write a.intValue() internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

3 3 3