Contents

[Data Types 2](#_Toc161610529)

[Primitive Data Types 2](#_Toc161610530)

[Non-Primitive Data Types 3](#_Toc161610531)

[Call by value and call by reference 4](#_Toc161610532)

[Wrapper Classes in Java 7](#_Toc161610533)

[Need of Wrapper Classes 8](#_Toc161610534)

[Primitive Data types and their Corresponding Wrapper class 8](#_Toc161610535)

[Autoboxing 8](#_Toc161610536)

[Unboxing 9](#_Toc161610537)

[Variable Naming 10](#_Toc161610538)

[Java Literals 10](#_Toc161610539)

[Implicit type conversion 10](#_Toc161610540)

[Widening or Automatic Type Conversion 11](#_Toc161610541)

[Type promotion in Expressions 12](#_Toc161610542)

[Upcasting Vs Downcasting in Java 13](#_Toc161610543)

[JAVA operators 17](#_Toc161610544)

[Arithmetic Operators 17](#_Toc161610545)

[Relational Operators 17](#_Toc161610546)

[Bitwise Operators 19](#_Toc161610547)

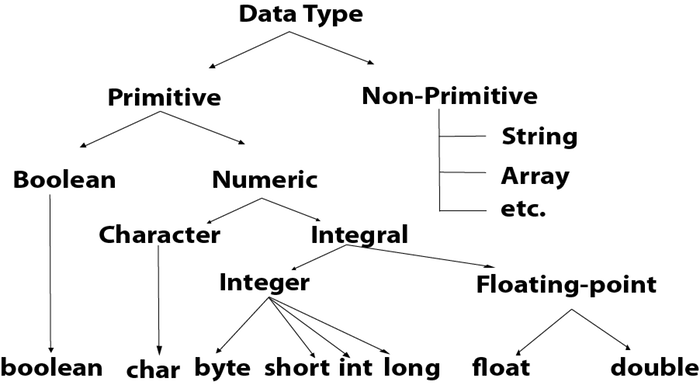
[Logical Operators 20](#_Toc161610548)

[Assignment Operators 20](#_Toc161610549)

[Misc Operators 22](#_Toc161610550)

[Conditional operator(?:) 22](#_Toc161610551)

# Data Types



Every variable in java has a data type. Data types specify the size and type of values that can be stored in an identifier. Java language is rich in its data types. The variety of data types available allow the programmer to select the type appropriate to the need of the application.

In java, data types are classified into two categories:

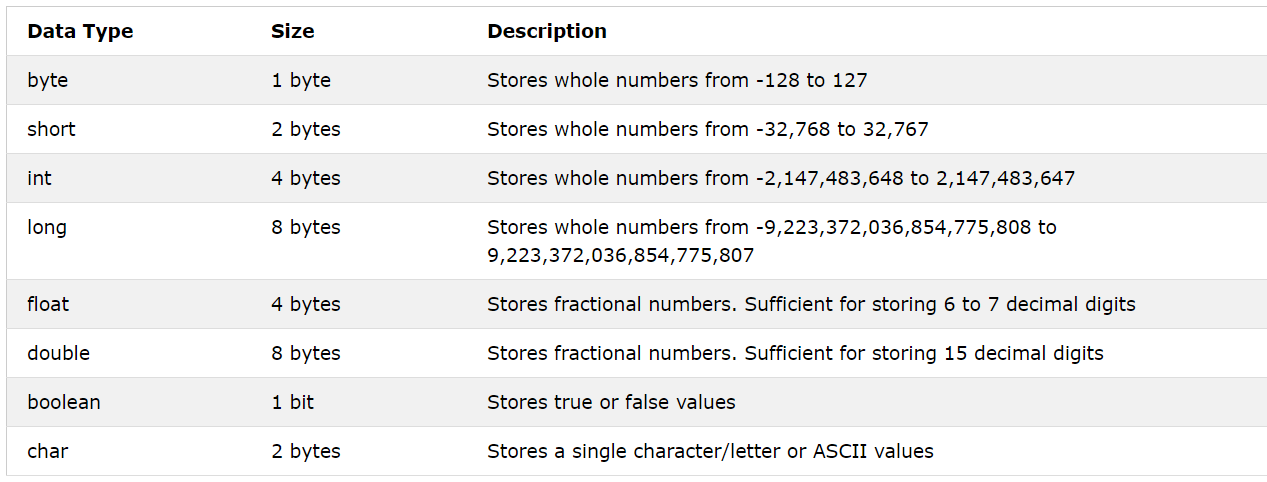
* Primitive Data type or Intrinsic or built-in data type
* Non-Primitive Data type or derived  or reference data type

## Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

the primitive data types are the predefined data types of Java. They specify the size and type of any standard values. Java has 8 primitive data types namely byte, short, int, long, float, double, char and boolean. When a primitive data type is stored, it is the stack that the values will be assigned. When a variable is copied then another copy of the variable is created and changes made to the copied variable will not reflect changes in the original variable

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/data-types-in-java.jpg)



//here if we dont use f at end of the value then it will automatically consider value as double and will give exception

## Non-Primitive Data Types

Non-primitive data types are called reference types because they refer to objects.

Unlike primitive data types, these are not predefined. These are user-defined data types created by programmers. These data types are used to store multiple values.

For example, consider an [array](https://www.javatpoint.com/array-in-java) that stores a group of values. Class is also a primitive type that stores different methods and variables. Therefore, these are also called as **advanced data types** in Java.

Whenever a non-primitive data type is defined, it refers a memory location where the data is stored in heap memory i.e., it refers to the memory location where an object is placed. Therefore, a non-primitive data type variable is also called **referenced data type** or **simply object reference variable**.

The main difference between primitive and non-primitive data types are:

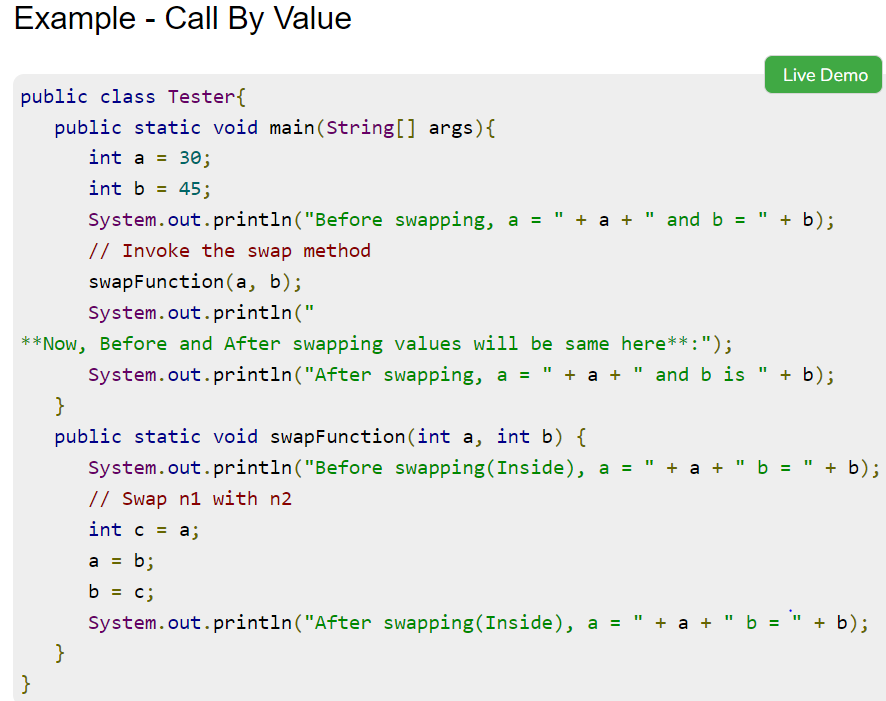
* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

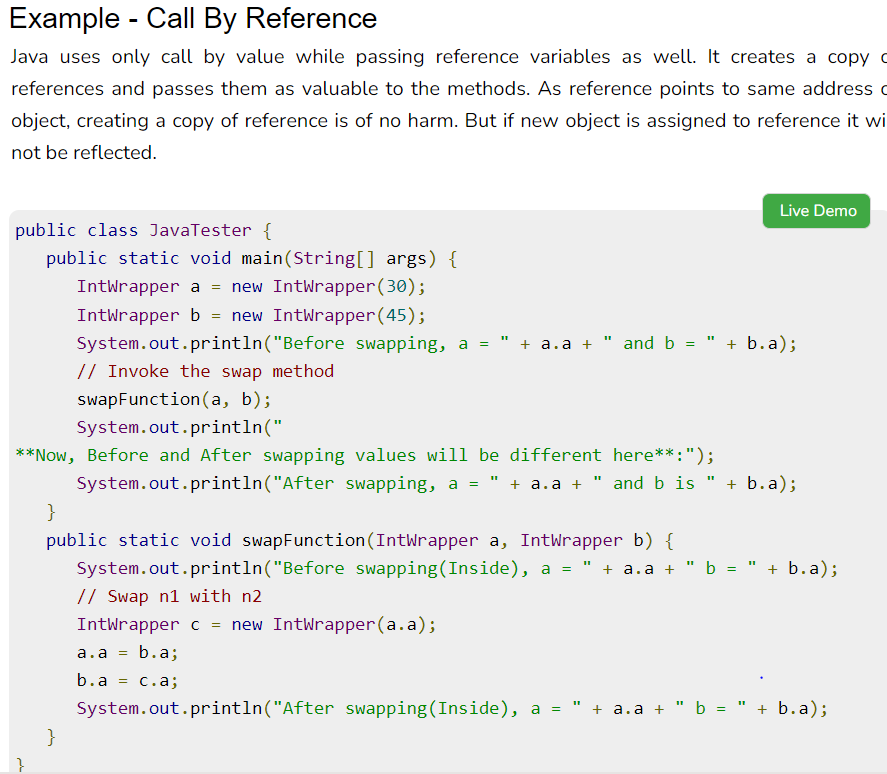
# Call by value and call by reference

Call by Value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.

While Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.

In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope.





<https://www.prepbytes.com/blog/java/call-by-value-and-call-by-reference-in-java/>

Some people will say incorrectly that objects are passed "by reference." In programming language design, the term pass by reference properly means that when an argument is passed to a function, the invoked function gets a reference to the original value, not a copy of its value. If the function modifies its parameter, the value in the calling code will be changed because the argument and parameter use the same slot in memory.... The Java programming language does not pass objects by reference; it passes object references by value. Because two copies of the same reference refer to the same actual object, changes made through one reference variable are visible through the other. There is exactly one parameter passing mode -- pass by value -- and that helps keep things simple.

Now what is interesting is that, the pass by value, is a theoretical Java language specification term, which has no ambiguities. But when it comes to how human beings perceive different instances of parameters, object values in a Java program, the experienced notion is a little different.

Java always passes by value. The value of the variable.

For a primitive variable like

1. private int mePromitive = 10;

* The value of the variable is 10.
* So when you pass this to a method, Java makes a copy of this variable (i.e another variable with value 10) and sends it.
* This new variable has no relation to the original variable.
* Any changes done by the method on that new variable has no effect on the original variable.

Now for a non-primitive variable like:

1. private Employee meNoPrimitve;

* The value of the variable ‘meNotPrimitive’ is a value which is the address of the location of a chunk of memory which has details about the contents of this Employee instance. (name field value, salary field value etc.)
* So when you pass such a variable to a method, Java still makes a copy by value on this, by creating a new variable with a copied value of the same address.
* Now if the new method changes the name or salary of the passed in object, then the original object’s name and salary field value will be changed.
* This is because, for a non-primitve variable, the value of the variable itself (which is an address location) is never changed. What it points to is changed.
* And even for copied variable address values, both unrelated variables end up pointing to the same address location, represented by the original employee instance.

So this is kind of a workaround to modify non-primitve object values using helper methods.

E. G.

1. Employee employee = new Employee();
2. populateNameAndSalary(employee);

Please not that the called method cannot swap the object or anything. It can only set its field values etc.

In all this, please remember that Java is pass-by-value only. Always.

# Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

The wrapper classes in Java are used to convert primitive types (int, char, float, etc) into corresponding objects.

## Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

## Primitive Data types and their Corresponding Wrapper class

[](https://media.geeksforgeeks.org/wp-content/uploads/Wrapper-Class.png)

Autoboxing

 Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.  
Example:

|  |
| --- |
| // Java program to demonstrate Autoboxing    import java.util.ArrayList;  class Autoboxing  {      public static void main(String[] args)      {          char ch = 'a';            // Autoboxing- primitive to Character object conversion          Character a = ch;            ArrayList<Integer> arrayList = new ArrayList<Integer>();            // Autoboxing because ArrayList stores only objects          arrayList.add(25);            // printing the values from object          System.out.println(arrayList.get(0));      }  } |

Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

|  |
| --- |
| // Java program to demonstrate Unboxing  import java.util.ArrayList;    class Unboxing  {      public static void main(String[] args)      {          Character ch = 'a';            // unboxing - Character object to primitive conversion          char a = ch;            ArrayList<Integer> arrayList = new ArrayList<Integer>();          arrayList.add(24);            // unboxing because get method returns an Integer object          int num = arrayList.get(0);            // printing the values from primitive data types          System.out.println(num);      }  } |

# Variable Naming

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

* Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "\_". The convention, however, is to always begin your variable names with a letter, not "$" or "\_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "\_", this practice is discouraged. White space is not permitted.
* Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g. Also keep in mind that the name you choose must not be a [keyword or reserved word](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html).
* If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention. If your variable stores a constant value, such as static final int NUM\_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

# Java Literals

Java Literals are syntactic representations of boolean, character, numeric, or string data. Literals provide a means of expressing specific values in your program. For example, in the following statement, an integer variable named count is declared and assigned an integer value. The literal 0 represents, naturally enough, the value zero.

Int count =0;

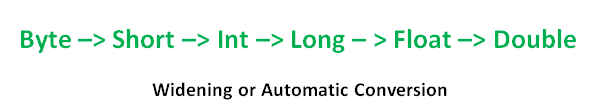
# Implicit type conversion

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

## Widening or Automatic Type Conversion

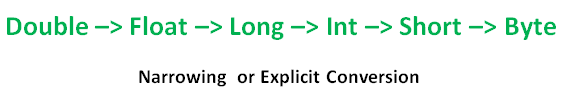
Widening conversion takes place when two data types are automatically converted. This happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.  
[](https://media.geeksforgeeks.org/wp-content/uploads/Widening-or-Automatic-Type-Conversion.png)

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

* This is useful for incompatible data types where automatic conversion cannot be done.
* Here, target-type specifies the desired type to convert the specified value to.

[](https://media.geeksforgeeks.org/wp-content/uploads/Narrowing-or-Explicit-Conversion.png)

|  |
| --- |
| //Java program to illustrate incompatible data  // type for explicit type conversion  public class Test  {    public static void main(String[] argv)    {      char ch = 'c';      int num = 88;      ch = num;    }  } |

Error:

7: error: incompatible types: possible lossy conversion from int to char

ch = num;

^

1 error

class Test

{

    public static void main(String[] args)

    {

        double d = 100.04;

        //explicit type casting

        long l = (long)d;

        //explicit type casting

        int i = (int)l;

        System.out.println("Double value "+d);

        //fractional part lost

        System.out.println("Long value "+l);

        //fractional part lost

        System.out.println("Int value "+i);

    }

}

Output:

Double value 100.04

Long value 100

Int value 100

## Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

|  |
| --- |
| //Java program to illustrate Type promotion in Expressions  class Test  {      public static void main(String args[])      {          byte b = 42;          char c = 'a';          short s = 1024;          int i = 50000;          float f = 5.67f;          double d = .1234;            // The Expression          double result = (f \* b) + (i / c) - (d \* s);            //Result after all the promotions are done          System.out.println("result = " + result);      }  } |

Output:

Result = 626.7784146484375

While evaluating expressions, the result is automatically updated to larger data type  of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.  
Example:

|  |
| --- |
| //Java program to illustrate type casting int to byte  class Test  {      public static void main(String args[])      {          byte b = 50;            //type casting int to byte          b = (byte)(b \* 2);          System.out.println(b);      }  } |

Output

100

## Upcasting Vs Downcasting in Java

Just like the datatypes, the objects can also be typecasted. However, in objects, there are only two types of objects (i.e.) parent object and child object. Therefore, typecasting of objects basically mean that one type of object (i.e.) child or parent to another. There are two types of typecasting. They are:

Upcasting: Upcasting is the [typecasting](https://www.geeksforgeeks.org/type-conversion-java-examples/) of a child object to a parent object. Upcasting can be done implicitly. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.

Downcasting: Similarly, downcasting means the typecasting of a parent object to a child object. Downcasting cannot be implicitly.

Upcasting is always allowed, but downcasting involves a type check and can throw a ClassCastException.

Upcasting example:

class Parent{

void PrintData() {

System.out.println("method of parent class");

}

}

class Child extends Parent {

void PrintData() {

System.out.println("method of child class");

}

}

class UpcastingExample{

public static void main(String args[]) {

Parent obj1 = (Parent) new Child();

Parent obj2 = (Parent) new Child();

obj1.PrintData();

obj2.PrintData();

}

}

Downcasting example:

Strictly typed languages enforce typing on all data being interacted with.

For example

int i = 3

string s = "4"

From here on out, whenever you use i, you can only interact with it as an integer type. That means you are restricted to using with methods that work with integers.

As for string s you can only interact with it as a string type. You can concatenate it with other string, print it out, etc. However, even though it contains that character "4", you cannot add to an integer without using some function to convert the string to an integer type.

In a dynamically typed language, you have a lot more flexibility:

//Parent class

class Parent {

String name;

// A method which prints the data of the parent class

void showMessage()

{

System.out.println("Parent method is called");

}

}

// Child class

class Child extends Parent {

int age;

// Performing overriding

@Override

void showMessage()

{

System.out.println("Child method is called");

}

}

public class Downcasting{

public static void main(String[] args)

{

Parent p = new Child();

p.name = "Shubham";

// Performing Downcasting Implicitly

//Child c = new Parent(); // it gives compile-time error

// Performing Downcasting Explicitly

Child c = (Child)p;

c.age = 18;

System.out.println(c.name);

System.out.println(c.age);

c.showMessage();

}

}

# JAVA operators

## Arithmetic Operators

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

|  |  |  |
| --- | --- | --- |
| * **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 |

## Relational Operators

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

|  |  |  |
| --- | --- | --- |
| **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. |

## Bitwise Operators

a = 0011 1100

b = 0000 1101

|  |  |  |
| --- | --- | --- |
| **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 |

## Logical Operators

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

|  |  |  |
| --- | --- | --- |
| **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 |

## Assignment Operators

|  |  |  |
| --- | --- | --- |
| **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 |

## Misc Operators

Conditional operator(?:)

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