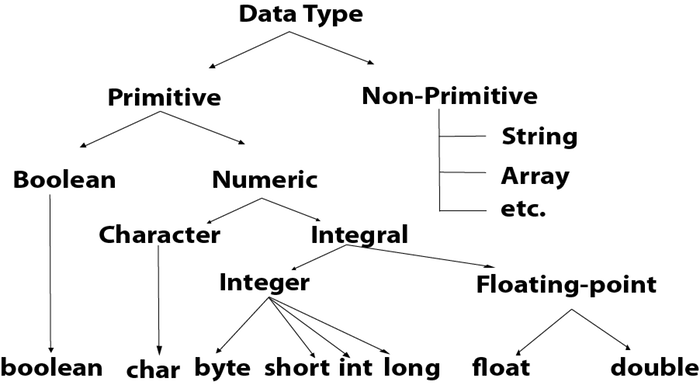
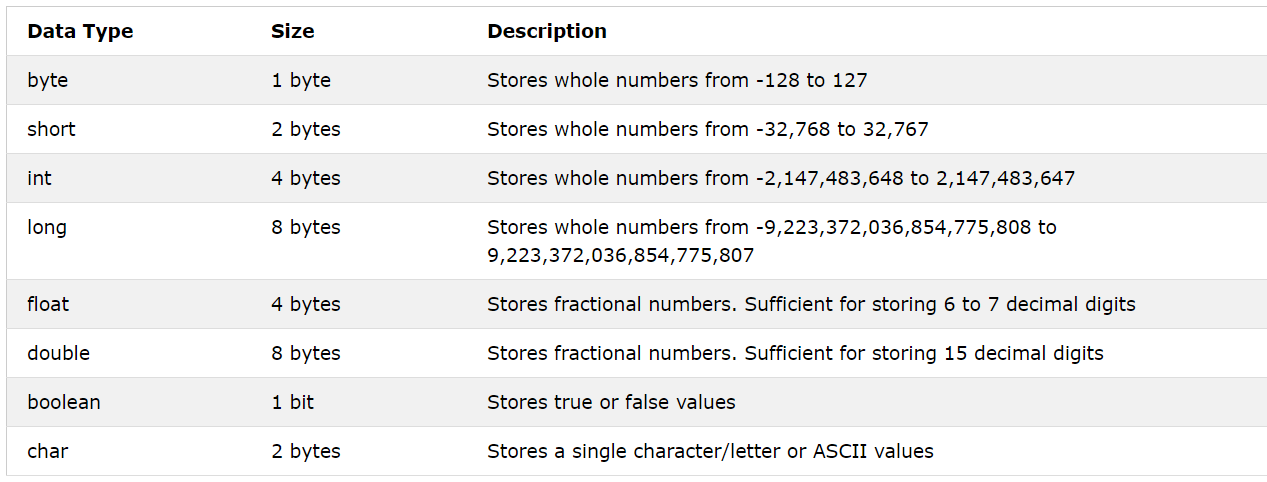
# Data Types



## Primitive Data Types

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

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



## Non-Primitive Data Types

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

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.

# 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;

# Enums

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).  
Enums are used when we know all possible values at **compile time**, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.

* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* According to [Java naming conventions](http://www.oracle.com/technetwork/java/codeconventions-135099.html), it is recommended that we name constant with all capital letters

Ex-1:

enum Color

{

    RED, GREEN, BLUE;

}

public class Test

{

    // Driver method

    public static void main(String[] args)

    {

        Color c1 = Color.RED;

        System.out.println(c1);

    }

}

EX -2:

public class Test

{

    enum Color

    {

        RED, GREEN, BLUE;

    }

    // Driver method

    public static void main(String[] args)

    {

        Color c1 = Color.RED;

        System.out.println(c1);

    }

}

Every enum internally implemented by using Class.

/\* internally above enum Color is converted to

class Color

{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}\*/

* Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name. Since it is final, we can’t create child enums.
* We can declare main() method inside enum. Hence we can invoke enum directly from the Command Prompt.
* All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
* toString() method is overridden in java.lang.Enum class,which returns enum constant name.
* enum can implement many interfaces.
* These methods are present inside java.lang.Enum.
* values() method can be used to return all values present inside enum.
* Order is important in enums.By using ordinal() method, each enum constant index can be found, just like array index.
* valueOf() method returns the enum constant of the specified string value, if exists

enum Color

{

    RED, GREEN, BLUE;

}

public class Test

{

    public static void main(String[] args)

    {

        // Calling values()

        Color arr[] = Color.values();

        // enum with loop

        for (Color col : arr)

        {

            // Calling ordinal() to find index

            // of color.

            System.out.println(col + " at index "

                             + col.ordinal());

        }

        // Using valueOf(). Returns an object of

        // Color with given constant.

        // Uncommenting second line causes exception

        // IllegalArgumentException

        System.out.println(Color.valueOf("RED"));

        // System.out.println(Color.valueOf("WHITE"));

    }

}

# 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 ;

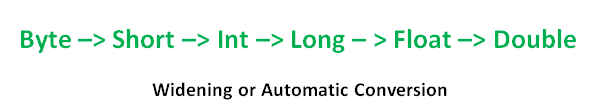
# Implicit type conversion

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

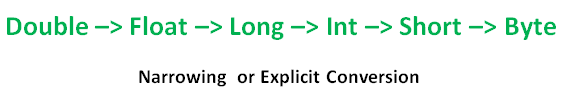
Widening conversion takes place when two data types are automatically converted. This happens when:

* 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.

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: