Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

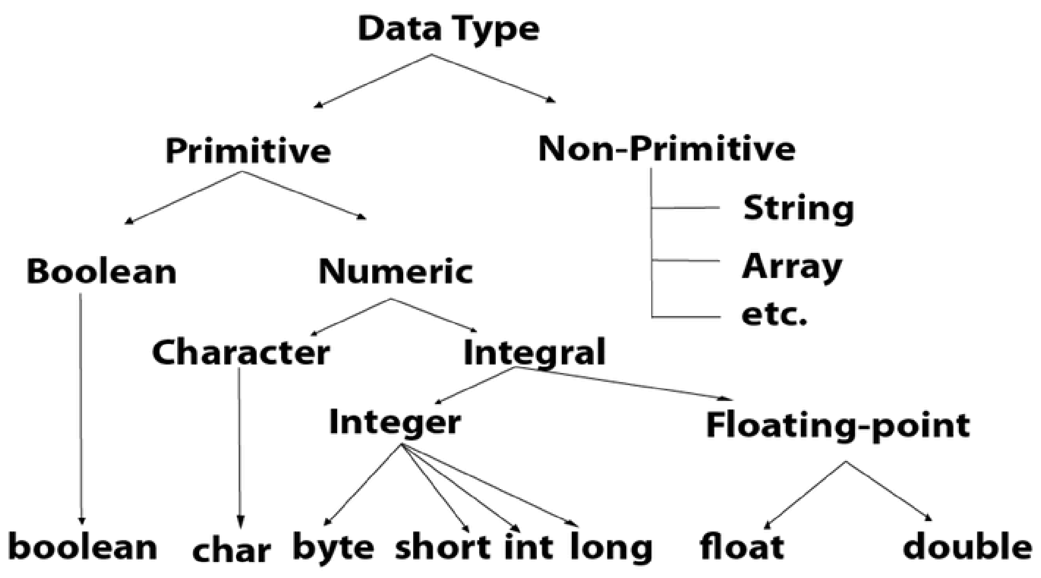
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



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

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

1. Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

1. byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

1. short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

1. int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between - 9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

1. long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

1. float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

1. double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

1. char letterA = 'A'

Literals in Java

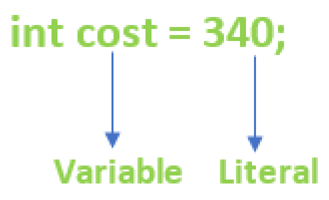
In Java

, **literal** is a notation that represents a fixed value in the source code. In lexical analysis, literals of a given type are generally known as **tokens**

. In this section, we will discuss the term **literals in Java**.

Literals

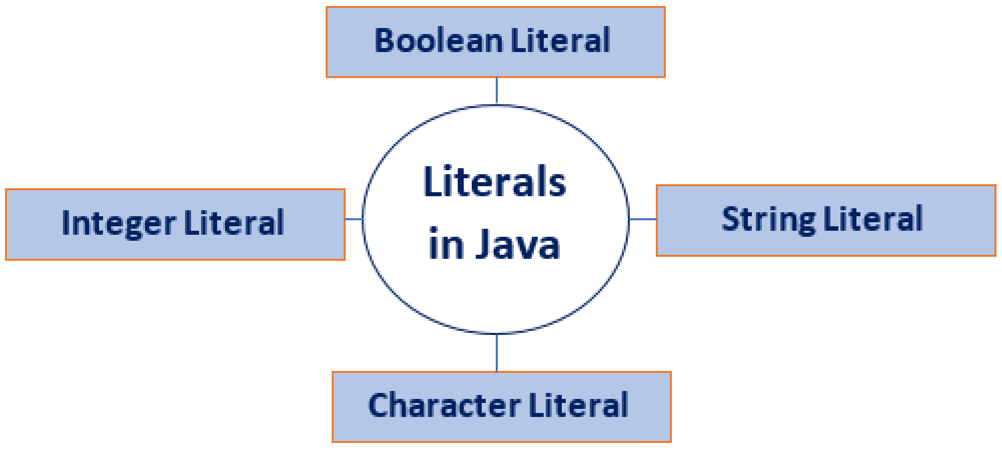
In Java, **literals** are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.



Types of Literals in Java

There are the majorly **four** types of literals in Java:

1. Integer Literal
2. Character Literal
3. Boolean Literal
4. String Literal



Integer Literals

Integer literals are sequences of digits. There are three types of integer literals:

- **Decimal Integer:** These are the set of numbers that consist of digits from 0 to 9. It may have a positive **(+)** or negative (**-**) Note that between numbers commas and non-digit characters are not permitted. For example, **5678, +657, -89,** etc.

1. int decVal = 26;

- **Octal Integer:** It is a combination of number have digits from 0 to 7 with a leading 0. For example, **045, 026,**

```
int octVal = 067;
```

- **Hexa-Decimal:** The sequence of digits preceded by **0x** or **0X** is considered as hexadecimal integers. It may also include a character from **a** to **f** or **A** to **F** that represents numbers from **10** to **15**, respectively. For example, **0xd, 0xf,**

```
int hexVal = 0x1a;
```

- **Binary Integer:** Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later). Prefix 0b represents the Binary system. For example, 0b11010.

```
int binVal = 0b11010;
```

Real Literals

The numbers that contain fractional parts are known as real literals. We can also represent real literals in exponent form. For example, **879.90, 99E-3,** etc.

Backslash Literals

Java supports some special backslash character literals known as backslash literals. They are used in formatted output. For example:

\n: It is used for a new line

\t: It is used for horizontal tab

\b: It is used for blank space

\v: It is used for vertical tab

\a: It is used for a small beep

\r: It is used for carriage return

\': It is used for a single quote

\": It is used for double quotes

Character Literals

A character literal is expressed as a character or an escape sequence, enclosed in a **single** quote ('') mark. It is always a type of char. For example, **'a', '%', '\u000d',** etc.

String Literals

String literal is a sequence of characters that is enclosed between **double** quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc. For example, "**Jack", "12345", "\n",** etc.

Floating Point Literals

The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals.

- Floating-point literals for float type end with F or f. For example, **6f, 8.354F**, etc. It is a **32**-bit float literal.
- Floating-point literals for double type end with D or d. It is optional to write D or d. For example, **6d, 8.354D,** etc. It is a **64**-bit double literal.
- It can also be represented in the form of the **exponent**.

Floating:

1. float length = 155.4f;

Decimal:

1. double interest = 99658.445;

Decimal in Exponent form:

1. double val= 1.234e2;

Boolean Literals

Boolean literals are the value that is either true or false. It may also have values 0 and 1. For example, **true, 0,** etc.

```
boolean isEven = true;
```

Null Literals

Null literal is often used in programs as a marker to indicate that reference type object is unavailable. The value **null** may be assigned to any variable, except variables of primitive types.

```
String stuName = null;
Student age = null;
```

Class Literals

Class literal formed by taking a type name and appending **.class** extension. For example, **Scanner.class**. It refers to the object (of type Class) that represents the type itself.

```
class classType = Scanner.class;
```

Invalid Literals

There is some invalid declaration of literals.

```
float g = 6_.674f;
float g = 6._674F;
long phoneNumber = 99_00_99_00_99_L;
int x = 77_;
int y = 0_x76;
int z = 0X_12;
int z = 0X12_;
```

Java Variables

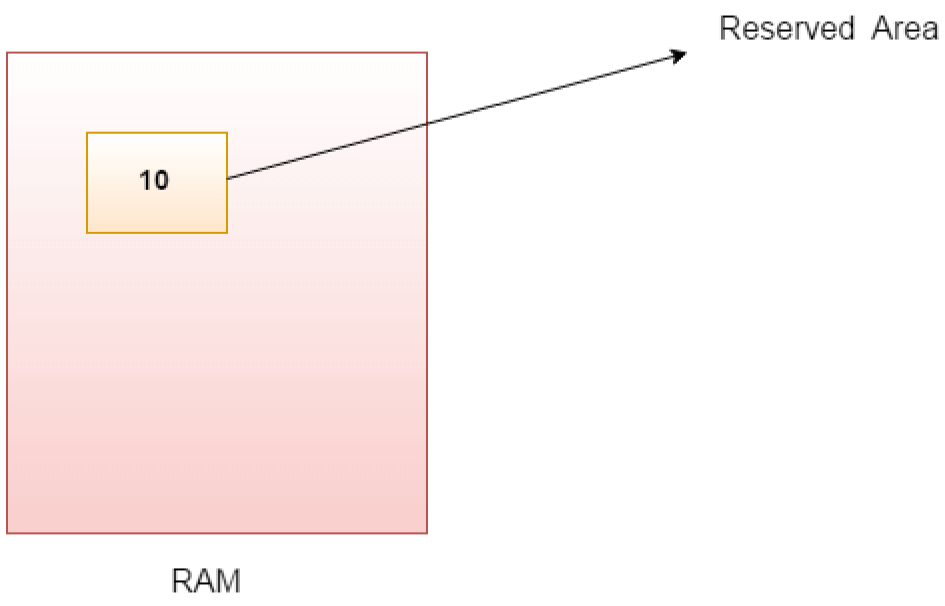
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

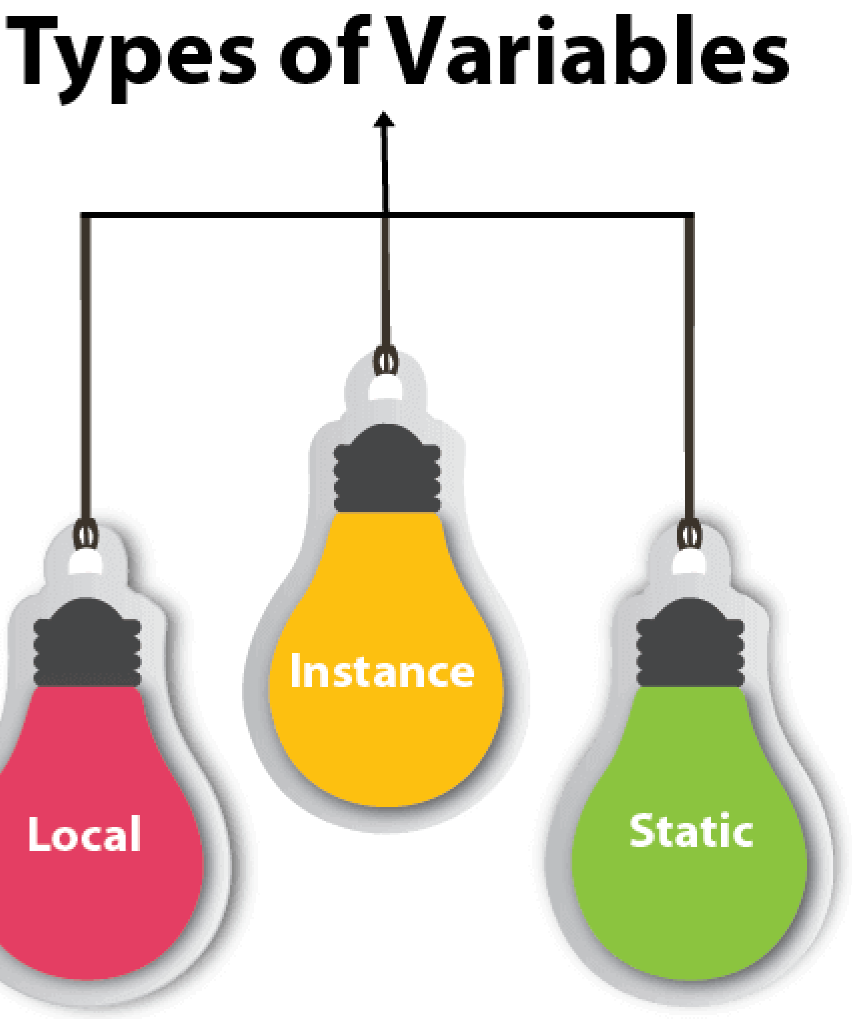


1. int data=50;//Here data is variable

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable



1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

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

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
}//end of class
```

Java Variable Example: Add Two Numbers

```
public 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

```
public 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)

```
public 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
```

--

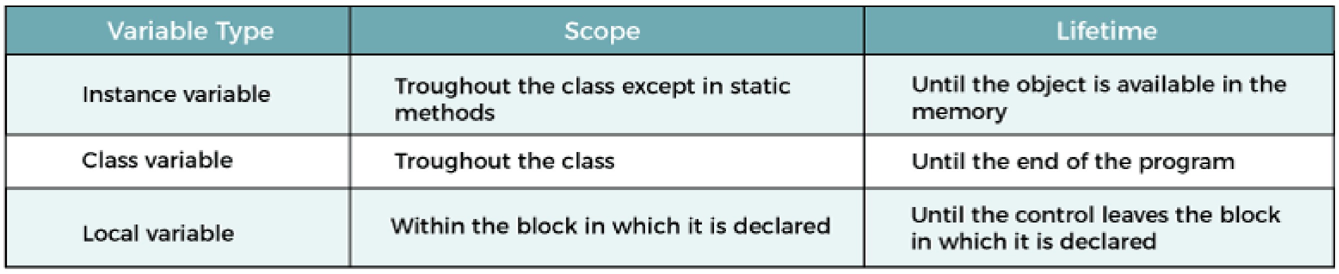
Scope of Variables in Java

In programming, **scope of variable** defines how a specific variable is accessible within the program or across classes. In this section, we will discuss the **scope of variables in Java**.

Scope of a Variable

In programming, a variable can be declared and defined inside a class, method, or block. It defines the scope of the variable i.e. the visibility or accessibility of a variable. Variable declared inside a block or method are not visible to outside. If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.

- We can declare variables anywhere in the program but it has limited scope.
- A variable can be a parameter of a method or constructor.
- A variable can be defined and declared inside the body of a method and constructor.
- It can also be defined inside blocks and loops.
- Variable declared inside main() function cannot be accessed outside the main() function



Variable Type	Scope	Lifetime
Instance variable	Troughout the class except in static methods	Until the object is available in the memory
Class variable	Troughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Demo.java

```
public class Demo
{
//instance variable
String name = "Andrew";
//class and static variable
static double height= 5.9;
public static void main(String args[])
{
//local variable
int marks = 72;
}
}
```

In Java, there are three types of variables based on their scope:

1. Member Variables (Class Level Scope)
2. Local Variables (Method Level Scope)

Member Variables (Class Level Scope)

These are the variables that are declared inside the class but outside any function have class-level scope. We can access these variables anywhere inside the class. Note that the access specifier of a member variable does not affect the scope within the class. Java allows us to access member variables outside the class with the following rules:

Access Modifier	Package	Subclass	Word
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	No	No	No
default	Yes	No	No

Syntax:

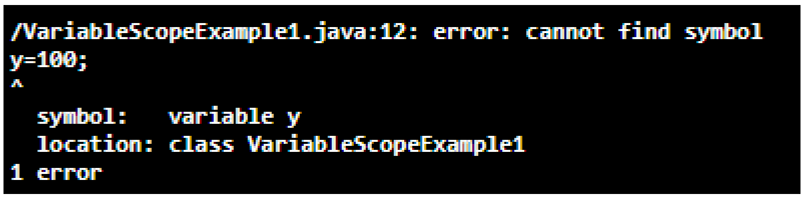
```
public class DemoClass
{
//variables declared inside the class have class level scope
int age;
private String name;
void displayName()
{
//statements
}
int dispalyAge()
{
//statements
}
char c;
}
```

Let's see an example.

VariableScopeExample1.java

```
public class VariableScopeExample1
{
public static void main(String args[])
{
int x=10;
{
//y has limited scope to this block only
int y=20;
System.out.println("Sum of x+y = " + (x+y));
}
//here y is unknown
y=100;
//x is still known
x=50;
}
}
```

Output:



```
/VariableScopeExample1.java:12: error: cannot find symbol
y=100;
^
  symbol:   variable y
  location: class VariableScopeExample1
1 error
```

We see that **y=100** is unknown. If you want to compile and run the above program remove or comment the statement **y=100.** After removing the statement, the above program runs successfully and shows the following output.

```
Sum of x+y = 30
```

There is another variable named an **instance** variable. These are declared inside a class but outside any method, constructor, or block. When an instance variable is declared using the keyword **static** is known as a static variable. Their scope is class level but visible to the method, constructor, or block that is defined inside the class.

Let's see an example.

Product.java

```
public class Product
{
//variable visible to any child class
public String pName;
//variable visible to product class only
private double pPrice;
//creating a constructor and parsed product name as a parameter
public Product (String pname)
{
pName = pname;
}
//function sets the product price
public void setPrice(double pprice)
{
pPrice= pprice;
}
//method prints all product info
public void getInfo()
{
System.out.println("Product Name: " +pName );
System.out.println("Product Price: " +pPrice);
}
public static void main(String args[])
{
Product pro = new Product("Mac Book");
pro.setPrice(65000);
pro.getInfo();
}
}
```

Output:

```
Product Name: Mac Book
Product Price: 65000.0
```

Let's see another example.

StaticVariableScope.java

```
public class StaticVariableScope
{
//declaring a private static variable
private static double pivalue;
//declaring a constant variable
public static final String piconstant = "PI";
public static void main(String args[])
{
pivalue = 3.14159265359;
System.out.println("The value of " + piconstant + " is: " + pivalue);
}
}
```

Output:

```
The value of PI is: 3.14159265359
```

Local Variables (Method Level Scope)

These are the variables that are declared inside a **method, constructor**, or **block** have a **method-level** or **block-level** scope and cannot be accessed outside in which it is defined. Variables declared inside a pair of curly braces **{}** have block-level scope.

Declaring a Variable Inside a Method

```
public class DemoClass1
{
void show()
{
//variable declared inside a method has method level scope
int x=10;
System.out.println("The value of x is: "+x);
}
public static void main(String args[])
{
DemoClass1 dc = new DemoClass1();
dc.show();
}
}
```

Output:

```
The value of x is: 10
```

Let's see another example of method-level scope.

DemoClass2.java

```
public class DemoClass2
{
private int a;
public void setNumber(int a)
{
this.a = a;
System.out.println("The value of a is: "+a);
}
public static void main(String args[])
{
DemoClass2 dc = new DemoClass2();
dc.setNumber(3);
}
}
```

Output:

```
The value of a is: 3
```

In the above example, we have passed a variable as a parameter. We have used **this** keyword that differentiates the class variable and local variable.

Declaring a Variable Inside a Constructor

VariableInsideConstructor.java

```
public class VariableInsideConstructor
{
//creating a default constructor
VariableInsideConstructor()
{
int age=24;
System.out.println("Age is: "+age);
}
//main() method
public static void main(String args[])
{
//calling a default constructor
VariableInsideConstructor vc=new VariableInsideConstructor();
}
}
```

Output:

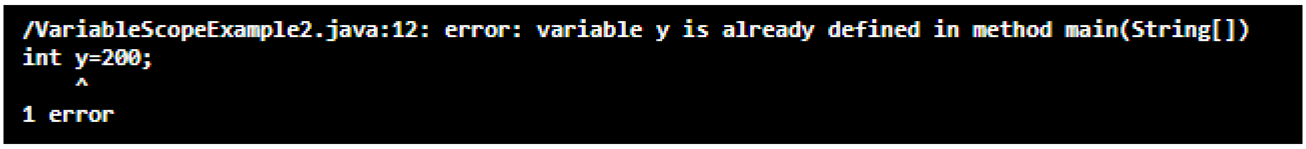
```
Age is: 24
```

Declaring a Variable Inside a Block

VariableInsideBlock.java

```
public class VariableInsideBlock
{
public static void main(String args[])
{
int x=4;
{
//y has limited scope to this block only
int y=100;
System.out.println("Sum of x+y = " + (x+y));
y=10;
//gives error, already defined
int y=200;
}
//creates a new variable
int y;
}
}
```

Output:



```
/VariableScopeExample2.java:12: error: variable y is already defined in method main(String[])
int y=200;
    ^
1 error
```

We see that **y=100** is unknown. If you want to compile and run the above program remove or comment the statement **y=100.** After removing the statement, the above program runs successfully and shows the following output.

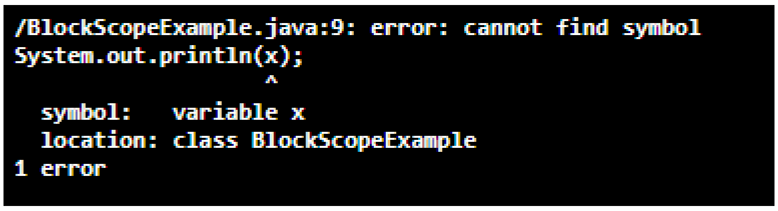
```
Sum of x+y = 30
```

Let's see another example.

BlockScopeExample1.java

```
public class BlockScopeExample1
{
public static void main(String args[])
{
for (int x = 0; x < 10; x++)
{
System.out.println(x);
}
System.out.println(x);
}
}
```

Output:



```
/BlockScopeExample.java:9: error: cannot find symbol
System.out.println(x);
                   ^
  symbol:   variable x
  location: class BlockScopeExample
1 error
```

When we run the above program, it shows an error at line 9, **cannot find symbol** because we have tried to print the variable x that is declared inside the loop. To resolve this error, we need to declare the variable x just before the for loop.

BlockScopeExample2.java

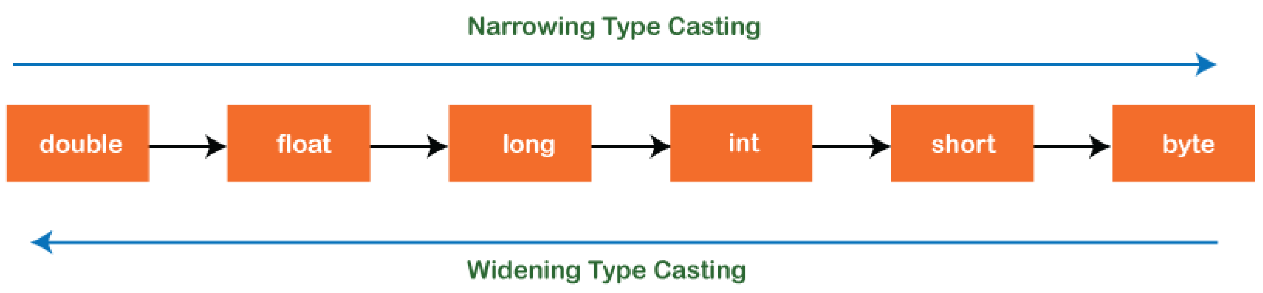
```
public class BlockScopeExample2
{
public static void main(String args[])
{
int x;
for (x = 0; x < 10; x++)
{
//prints 0 to 9
System.out.print(x+"\t");
}
//prints 10
System.out.println(x);
}
}
```

Output:

```
0       1       2       3       4       5       6       7       8       9       10
```

Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

```
byte -> short -> char -> int -> long -> float -> double
```

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

```
double -> float -> long -> int -> char -> short -> byte
```

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

Java Arrays

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

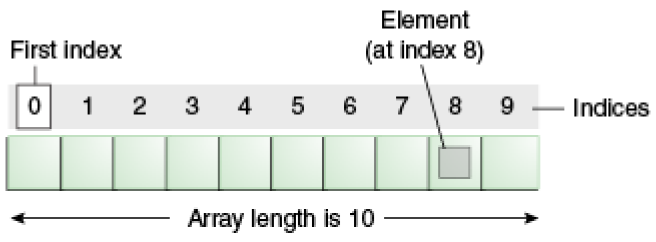
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

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

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

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

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



Advantages

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

Disadvantages

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

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

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

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

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java 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;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Test it Now

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

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

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

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

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
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]);
}}
```

Test it Now

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Let us see the example of print the elements of Java array using the for-each loop.

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

```
33
3
4
5
```