Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to [Java](https://www.javatpoint.com/java-tutorial). See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the differences between the JDK, JRE, and JVM.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](https://www.javatpoint.com/os-tutorial) is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and [applets](https://www.javatpoint.com/java-applet). It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



Internal Details of Hello Java Program

we have learnt about the first program, how to compile and run the first java program. Here, we are going to learn, what happens while compiling and running the java program. Moreover, we will see some question based on the first program.

What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

What happens at runtime?

At runtime, following steps are performed:



|  |
| --- |
| **Classloader:**is the subsystem of JVM that is used to load class files. |
| **Bytecode Verifier:**checks the code fragments for illegal code that can violate access right to objects. |
| **Interpreter:**read bytecode stream then execute the instructions. |

Q) Can you save a java source file by other name than the class name?

|  |
| --- |
| Yes, if the class is not public. It is explained in the figure given below: |
| how to save simple java program by another name |

|  |  |
| --- | --- |
| **To compile:** | javac Hard.java |
| **To execute:** | java Simple |

Q) Can you have multiple classes in a java source file?

|  |
| --- |
| Yes, like the figure given below illustrates: |
| how to contain multiple class in simple java program |

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:** 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:** 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:** 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:** 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:** 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:** 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:** 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:** char letterA = 'A'

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Unicode System

|  |
| --- |
| Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages. |

Why java uses Unicode System?

|  |
| --- |
| Before Unicode, there were many language standards: |
| * **ASCII** (American Standard Code for Information Interchange) for the United States. * **ISO 8859-1** for Western European Language. * **KOI-8** for Russian. * **GB18030 and BIG-5** for chinese, and so on. |

Problem

|  |
| --- |
| **This caused two problems:**   1. A particular code value corresponds to different letters in the various language standards. 2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte. |

Solution

|  |
| --- |
| To solve these problems, a new language standard was developed i.e. Unicode System. |
| In unicode, character holds 2 byte, so java also uses 2 byte for characters. |
| **lowest value:**\u0000 |
| **highest value:**\uFFFF |

# Java Variables

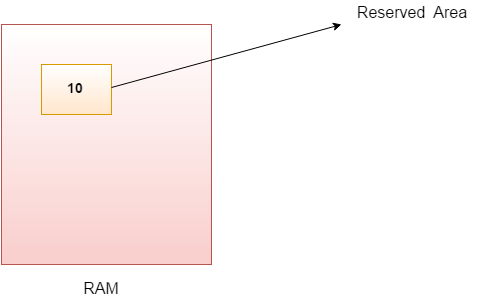
A variable is a container which holds the value while the java program is executed. A variable is assigned with a datatype.

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

**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.

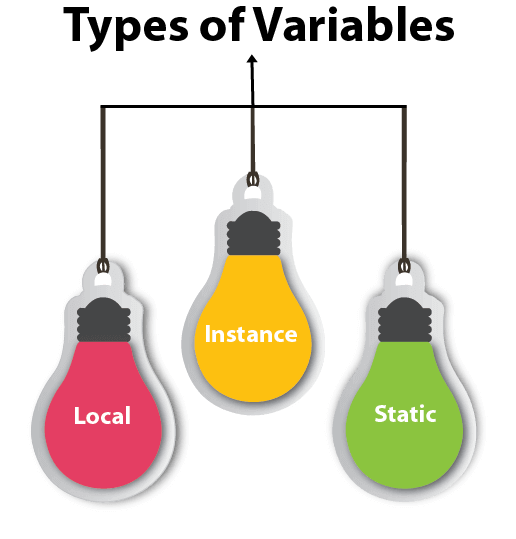


1. **int** data=50;//Here data is variable

### 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 instance variable. It is not declared as static.

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

#### 3) Static variable

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

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

1. **class** A{
2. **int** data=50;//instance variable
3. **static** **int** m=100;//static variable
4. **void** method(){
5. **int** n=90;//local variable
6. }
7. }//end of class

Java Variable Example: Add Two Numbers

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **int** b=10;
5. **int** c=a+b;
6. System.out.println(c);
7. }}

Output:

20

Java Variable Example: Widening

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **float** f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}

Output:

10

10.0

Java Variable Example: Narrowing (Typecasting)

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **float** f=10.5f;
4. //int a=f;//Compile time error
5. **int** a=(**int**)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}

Output:

10.5

10

Java Variable Example: Overflow

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. //Overflow
4. **int** a=130;
5. **byte** b=(**byte**)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}

Output:

130

-126

Java Variable Example: Adding Lower Type

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **byte** a=10;
4. **byte** b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. **byte** c=(**byte**)(a+b);
7. System.out.println(c);
8. }}

Output:

20

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

* incrementing/decrementing a value by one
* negating an expression
* inverting the value of a boolean

Java Unary Operator Example: ++ and --

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}

Output:

10

12

12

10

Java Unary Operator Example 2: ++ and --

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
8. }}

Output:

22

21

Java Unary Operator Example: ~ and !

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=-10;
5. **boolean** c=**true**;
6. **boolean** d=**false**;
7. System.out.println(~a);//-11 (minus of total positive value which starts from 0)
8. System.out.println(~b);//9 (positive of total minus, positive starts from 0)
9. System.out.println(!c);//false (opposite of boolean value)
10. System.out.println(!d);//true
11. }}

Output:

-11

9

false

true

Java Arithmetic Operators

Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a\*b);//50
8. System.out.println(a/b);//2
9. System.out.println(a%b);//0
10. }}

Output:

15

5

50

2

0

Java Arithmetic Operator Example: Expression

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10\*10/5+3-1\*4/2);
4. }}

Output:

21

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10<<2);//10\*2^2=10\*4=40
4. System.out.println(10<<3);//10\*2^3=10\*8=80
5. System.out.println(20<<2);//20\*2^2=20\*4=80
6. System.out.println(15<<4);//15\*2^4=15\*16=240
7. }}

Output:

40

80

80

240

Java Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}

Output:

2

5

2

Java Shift Operator Example: >> vs >>>

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. //For positive number, >> and >>> works same
4. System.out.println(20>>2);
5. System.out.println(20>>>2);
6. //For negative number, >>> changes parity bit (MSB) to 0
7. System.out.println(-20>>2);
8. System.out.println(-20>>>2);
9. }}

Output:

5

5

-5

1073741819

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}

Output:

false

false

Java AND Operator Example: Logical && vs Bitwise &

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10. }}

Output:

false

10

false

11

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}

Output:

true

true

true

10

true

11

Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in java programming. it is the only conditional operator which takes three operands.

Java Ternary Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=2;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

Output:

2

Another Example:

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=5;
5. **int** min=(a<b)?a:b;
6. System.out.println(min);
7. }}

Output:

5

Java Assignment Operator

Java assignment operator is one of the most common operator. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **int** a=10;
4. **int** b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}

Output:

14

16

Java Assignment Operator Example

1. **class** OperatorExample{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a\*=2;//9\*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}

Output:

13

9

18

9

Java Assignment Operator Example: Adding short

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. //a+=b;//a=a+b internally so fine
6. a=a+b;//Compile time error because 10+10=20 now int
7. System.out.println(a);
8. }}

Output:

Compile time error

After type cast:

1. **class** OperatorExample{
2. **public** **static** **void** main(String args[]){
3. **short** a=10;
4. **short** b=10;
5. a=(**short**)(a+b);//20 which is int now converted to short
6. System.out.println(a);
7. }}

Output:

20