**Variables and Data Types in Java**

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.

variables in java

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

**Types of Variable**

There are three types of variables in java:

* local variable
* instance variable
* static variable

types of variables in java

**1) Local Variable**

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

**2) Instance Variable**

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

**3) Static variable**

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

We will have detailed learning of these variables in next chapters.

**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

**Data Types in Java**

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

* Primitive data types
* Non-primitive data types



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| boolean | false | 1 bit |
| char | '\u0000' | 2 byte |
| byte | 0 | 1 byte |
| short | 0 | 2 byte |
| int | 0 | 4 byte |
| long | 0L | 8 byte |
| float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

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

It is because java uses Unicode system than ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

**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 - Basic Datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java −

* Primitive Data Types
* Reference/Object Data Types

**Primitive Data Types**

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

**byte**

* Byte data type is an 8-bit signed two's complement integer
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
* Example: byte a = 100, byte b = -50

**short**

* Short data type is a 16-bit signed two's complement integer
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
* Default value is 0.
* Example: short s = 10000, short r = -20000

**int**

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648 (-2^31)
* Maximum value is 2,147,483,647(inclusive) (2^31 -1)
* Integer is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0
* Example: int a = 100000, int b = -200000

**long**

* Long data type is a 64-bit signed two's complement integer
* Minimum value is -9,223,372,036,854,775,808(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
* This type is used when a wider range than int is needed
* Default value is 0L
* Example: long a = 100000L, long b = -200000L

**float**

* Float data type is a single-precision 32-bit IEEE 754 floating point
* Float is mainly used to save memory in large arrays of floating point numbers
* Default value is 0.0f
* Float data type is never used for precise values such as currency
* Example: float f1 = 234.5f

**double**

* double data type is a double-precision 64-bit IEEE 754 floating point
* This data type is generally used as the default data type for decimal values, generally the default choice
* Double data type should never be used for precise values such as currency
* Default value is 0.0d
* Example: double d1 = 123.4

**boolean**

* boolean data type represents one bit of information
* There are only two possible values: true and false
* This data type is used for simple flags that track true/false conditions
* Default value is false
* Example: boolean one = true

**char**

* char data type is a single 16-bit Unicode character
* Minimum value is '\u0000' (or 0)
* Maximum value is '\uffff' (or 65,535 inclusive)
* Char data type is used to store any character
* Example: char letterA = 'A'

**Reference Datatypes**

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
* Class objects and various type of array variables come under reference datatype.
* Default value of any reference variable is null.
* A reference variable can be used to refer any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

**Java Literals**

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example −

byte a = 68;

char a = 'A'

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example −

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are −

**Example**

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example −

char a = '\u0001';

String a = "\u0001";

Java language supports few special escape sequences for String and char literals as well. They are −

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | tab |
| \" | Double quote |
| \' | Single quote |
| \\ | backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

# Escape Sequences in Java with Examples

Escape characters (also called escape sequences or escape codes) in general are used to signal an alternative interpretation of a series of characters. In Java, a character preceded by a backslash (\) is an escape sequence and has special meaning to the java compiler.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly. For example, if you want to put quotes within quotes you must use the escape sequence, **\"**, on the interior quotes. To print the sentence: **She said "Hello!" to me.**you should write:

System.out.println("She said **\"Hello!\"** to me.");

**Escape sequences available in java** are:

\t - Insert a tab in the text at this point.  
\b - Insert a backspace in the text at this point.  
\n - Insert a newline in the text at this point.  
\r - Insert a carriage return in the text at this point.  
\f - Insert a formfeed in the text at this point.  
\' - Insert a single quote character in the text at this point.  
\" - Insert a double quote character in the text at this point.  
\\ - Insert a backslash character in the text at this point.

An escape sequence is a single character. For example

**System.out.println("\n".length());**will print 1.

# Java Format Specifier

### Description

The format() method accepts a wide variety of format specifiers. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and lowercase specifiers perform the same conversion.

### Specifier List

The following table shows the format specifiers:

| **Format Specifier** | **Conversion Applied** |
| --- | --- |
| %a %A | Floating-point hexadecimal |
| %b %B | Boolean |
| %c | Character |
| %d | Decimal integer |
| %h %H | Hash code of the argument |
| %e %E | Scientific notation |
| %f | Decimal floating-point |
| %g %G | Uses %e or %f, whichever is shorter |
| %o | Octal integer |
| %n | Inserts a newline character |
| %s %S | String |
| %t %T | Time and date |
| %x %X | Integer hexadecimal |
| %% | Inserts a % sign |

If the argument doesn't match the type-checks, an IllegalFormatException is thrown.

### Example

Java Format Specifier

import java.util.Calendar;

import java.util.Formatter;

/\*w w w . j ava2 s . c om\*/

public class Main {

public static void main(String args[]) {

Formatter fmt = new Formatter();

fmt = new Formatter();

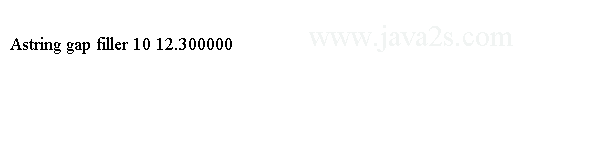
System.out.println(fmt.format("%s gap filler %d %f", "Astring", 10, 12.3));

}

}

You can obtain a reference to the underlying output buffer by calling out(). It returns a reference to an Appendable object.

The code above generates the following result.



### Example 2

Unknown Format Conversion Exception

import java.util.Date;

import java.util.Formatter;

/\*from w w w. java2s.c o m\*/

public class Main {

public static void main(String args[]) {

Formatter fmt = new Formatter();

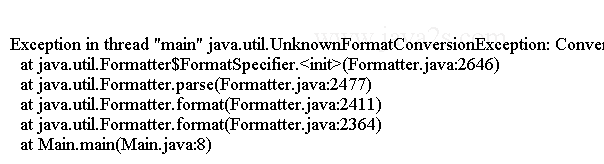
fmt.format("%t", new Date());

System.out.println(fmt);

}

}

The code above generates the following result.



### Example 3

The %n and %% format specifiers escape sequences. The %n inserts a newline. The %% inserts a percent sign.

You can use the standard escape sequence \n to embed a newline character. Here is an example that demonstrates the %n and %% format specifiers:

import java.util.Formatter;

/\*w w w . j a v a 2 s . c o m\*/

public class Main {

public static void main(String args[]) {

Formatter fmt = new Formatter();

fmt.format("line%nline %d%% complete", 88);

System.out.println(fmt);

}

}

The output:



### Example 4

The following code combines the %n, %d %% to display file copy progress information.

import java.util.Formatter;

//from w ww .jav a2 s.c om

public class Main {

public static void main(String args[]) {

Formatter fmt = new Formatter();

fmt.format("Copying file%nTransfer is %d%% complete", 88);

System.out.println(fmt);

}

}

The output:



**Operators in java**

**Operator** in java is a symbol that is used to perform operations. For example: +, -, \*, / etc.

There are many types of operators in java which are given below:

* Unary Operator,
* Arithmetic Operator,
* shift Operator,
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and
* Assignment Operator.

**Java Operator Precedence**

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | *expr*++ *expr*-- |
| prefix | ++*expr* --*expr* +*expr* ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**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 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 Shift Operator Example: Left Shift**

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 Shift Operator Example: Right Shift**

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 nagative 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 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 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

**Type Casting**

Assigning a value of one type to a variable of another type is known as **Type Casting**.

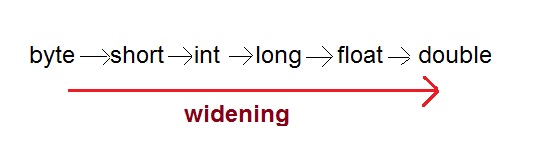
**Example :**

int x = 10;

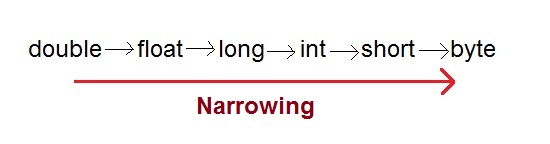
byte y = (byte)x;

In Java, type casting is classified into two types,

* Widening Casting(Implicit)



* Narrowing Casting(Explicitly done)



**Widening or Automatic type converion**

Automatic Type casting take place when,

* the two types are compatible
* the target type is larger than the source type

**Example :**

public class Test

{

public static void main(String[] args)

{

int i = 100;

long l = i; **//no explicit type casting required**

float f = l; **//no explicit type casting required**

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

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

System.out.println("Float value "+f);

}

}

**Output :**

Int value 100

Long value 100

Float value 100.0

**Narrowing or Explicit type conversion**

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**Example :**

public class Test

{

public static void main(String[] args)

{

double d = 100.04;

long l = (long)d; **//explicit type casting required**

int i = (int)l; **//explicit type casting required**

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

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

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

}

}

**Output :**

Double value 100.04

Long value 100

Int value 100