Java syntax

From Wikipedia, the free encyclopedia

[Jump to navigation](https://en.wikipedia.org/wiki/Java_syntax#mw-head)[Jump to search](https://en.wikipedia.org/wiki/Java_syntax#searchInput)

|  |  |
| --- | --- |
|  | This article includes a [list of references](https://en.wikipedia.org/wiki/Wikipedia%3ACiting_sources), but **its sources remain unclear because it has insufficient**[**inline citations**](https://en.wikipedia.org/wiki/Wikipedia%3ACiting_sources#Inline_citations). Please help to [improve](https://en.wikipedia.org/wiki/Wikipedia%3AWikiProject_Fact_and_Reference_Check) this article by [introducing](https://en.wikipedia.org/wiki/Wikipedia%3AWhen_to_cite) more precise citations. *(January 2014) (*[*Learn how and when to remove this template message*](https://en.wikipedia.org/wiki/Help%3AMaintenance_template_removal)*)* |

*See also:*[*Java (programming language) § Syntax*](https://en.wikipedia.org/wiki/Java_%28programming_language%29#Syntax)



A snippet of Java code with keywords highlighted in bold blue font

The [**syntax**](https://en.wikipedia.org/wiki/Syntax)**of**[**Java**](https://en.wikipedia.org/wiki/Java_%28programming_language%29) refers to [the set of rules](https://en.wikipedia.org/wiki/Syntax) defining how a Java program is written and interpreted.

The syntax is mostly derived from [C](https://en.wikipedia.org/wiki/C_%28programming_language%29) and [C++](https://en.wikipedia.org/wiki/C%2B%2B). Unlike in C++, in Java there are no global functions or variables, but there are data members which are also regarded as global variables. All code belongs to [classes](https://en.wikipedia.org/wiki/Class_%28computer_science%29) and all values are [objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29). The only exception is the [primitive types](https://en.wikipedia.org/wiki/Primitive_types), which are not represented by a class instance for performance reasons (though can be automatically converted to objects and vice versa via [autoboxing](https://en.wikipedia.org/wiki/Java_syntax#Boxing_and_unboxing)). Some features like [operator overloading](https://en.wikipedia.org/wiki/Operator_overloading) or [unsigned integer](https://en.wikipedia.org/wiki/Unsigned_integer) types are omitted to simplify the language and to avoid possible programming mistakes.

The Java syntax has been gradually extended in the course of numerous major [JDK](https://en.wikipedia.org/wiki/JDK) [releases](https://en.wikipedia.org/wiki/Java_version_history), and now supports capabilities such as [generic programming](https://en.wikipedia.org/wiki/Generic_programming) and [function literals](https://en.wikipedia.org/wiki/Function_literals) (called lambda expressions in Java). Since 2017, a new JDK version is released twice a year, with each release bringing incremental improvements to the language.



**Contents**

* [1Basics](https://en.wikipedia.org/wiki/Java_syntax#Basics)
	+ [1.1Identifier](https://en.wikipedia.org/wiki/Java_syntax#Identifier)
	+ [1.2Keywords](https://en.wikipedia.org/wiki/Java_syntax#Keywords)
	+ [1.3Literals](https://en.wikipedia.org/wiki/Java_syntax#Literals)
	+ [1.4Variables](https://en.wikipedia.org/wiki/Java_syntax#Variables)
	+ [1.5Code blocks](https://en.wikipedia.org/wiki/Java_syntax#Code_blocks)
	+ [1.6Comments](https://en.wikipedia.org/wiki/Java_syntax#Comments)
	+ [1.7Universal types](https://en.wikipedia.org/wiki/Java_syntax#Universal_types)
* [2Program structure](https://en.wikipedia.org/wiki/Java_syntax#Program_structure)
	+ [2.1main method](https://en.wikipedia.org/wiki/Java_syntax#main_method)
	+ [2.2Packages](https://en.wikipedia.org/wiki/Java_syntax#Packages)
	+ [2.3Import declaration](https://en.wikipedia.org/wiki/Java_syntax#Import_declaration)
		- [2.3.1Type import declaration](https://en.wikipedia.org/wiki/Java_syntax#Type_import_declaration)
		- [2.3.2Static import declaration](https://en.wikipedia.org/wiki/Java_syntax#Static_import_declaration)
* [3Operators](https://en.wikipedia.org/wiki/Java_syntax#Operators)
* [4Control structures](https://en.wikipedia.org/wiki/Java_syntax#Control_structures)
	+ [4.1Conditional statements](https://en.wikipedia.org/wiki/Java_syntax#Conditional_statements)
		- [4.1.1if statement](https://en.wikipedia.org/wiki/Java_syntax#if_statement)
		- [4.1.2switch statement](https://en.wikipedia.org/wiki/Java_syntax#switch_statement)
			* [4.1.2.1switch expressions](https://en.wikipedia.org/wiki/Java_syntax#switch_expressions)
	+ [4.2Iteration statements](https://en.wikipedia.org/wiki/Java_syntax#Iteration_statements)
		- [4.2.1while loop](https://en.wikipedia.org/wiki/Java_syntax#while_loop)
		- [4.2.2do ... while loop](https://en.wikipedia.org/wiki/Java_syntax#do_..._while_loop)
		- [4.2.3for loop](https://en.wikipedia.org/wiki/Java_syntax#for_loop)
		- [4.2.4Enhanced for loop](https://en.wikipedia.org/wiki/Java_syntax#Enhanced_for_loop)
	+ [4.3Jump statements](https://en.wikipedia.org/wiki/Java_syntax#Jump_statements)
		- [4.3.1Labels](https://en.wikipedia.org/wiki/Java_syntax#Labels)
		- [4.3.2break statement](https://en.wikipedia.org/wiki/Java_syntax#break_statement)
		- [4.3.3continue statement](https://en.wikipedia.org/wiki/Java_syntax#continue_statement)
		- [4.3.4return statement](https://en.wikipedia.org/wiki/Java_syntax#return_statement)
	+ [4.4Exception handling statements](https://en.wikipedia.org/wiki/Java_syntax#Exception_handling_statements)
		- [4.4.1try-catch-finally statements](https://en.wikipedia.org/wiki/Java_syntax#try-catch-finally_statements)
		- [4.4.2try-with-resources statements](https://en.wikipedia.org/wiki/Java_syntax#try-with-resources_statements)
		- [4.4.3throw statement](https://en.wikipedia.org/wiki/Java_syntax#throw_statement)
	+ [4.5Thread concurrency control](https://en.wikipedia.org/wiki/Java_syntax#Thread_concurrency_control)
	+ [4.6assert statement](https://en.wikipedia.org/wiki/Java_syntax#assert_statement)
* [5Primitive types](https://en.wikipedia.org/wiki/Java_syntax#Primitive_types)
	+ [5.1Boxing and unboxing](https://en.wikipedia.org/wiki/Java_syntax#Boxing_and_unboxing)
* [6Reference types](https://en.wikipedia.org/wiki/Java_syntax#Reference_types)
	+ [6.1Arrays](https://en.wikipedia.org/wiki/Java_syntax#Arrays)
		- [6.1.1Initializers](https://en.wikipedia.org/wiki/Java_syntax#Initializers)
		- [6.1.2Multi-dimensional arrays](https://en.wikipedia.org/wiki/Java_syntax#Multi-dimensional_arrays)
	+ [6.2Classes](https://en.wikipedia.org/wiki/Java_syntax#Classes)
		- [6.2.1Declaration](https://en.wikipedia.org/wiki/Java_syntax#Declaration)
		- [6.2.2Instantiation](https://en.wikipedia.org/wiki/Java_syntax#Instantiation)
		- [6.2.3Accessing members](https://en.wikipedia.org/wiki/Java_syntax#Accessing_members)
		- [6.2.4Modifiers](https://en.wikipedia.org/wiki/Java_syntax#Modifiers)
			* [6.2.4.1Access modifiers](https://en.wikipedia.org/wiki/Java_syntax#Access_modifiers)
		- [6.2.5Constructors and initializers](https://en.wikipedia.org/wiki/Java_syntax#Constructors_and_initializers)
		- [6.2.6Methods](https://en.wikipedia.org/wiki/Java_syntax#Methods)
			* [6.2.6.1Modifiers](https://en.wikipedia.org/wiki/Java_syntax#Modifiers_2)
			* [6.2.6.2Varargs](https://en.wikipedia.org/wiki/Java_syntax#Varargs)
		- [6.2.7Fields](https://en.wikipedia.org/wiki/Java_syntax#Fields)
			* [6.2.7.1Modifiers](https://en.wikipedia.org/wiki/Java_syntax#Modifiers_3)
		- [6.2.8Inheritance](https://en.wikipedia.org/wiki/Java_syntax#Inheritance)
			* [6.2.8.1Overriding methods](https://en.wikipedia.org/wiki/Java_syntax#Overriding_methods)
			* [6.2.8.2Abstract classes](https://en.wikipedia.org/wiki/Java_syntax#Abstract_classes)
		- [6.2.9Enumerations](https://en.wikipedia.org/wiki/Java_syntax#Enumerations)
	+ [6.3Interfaces](https://en.wikipedia.org/wiki/Java_syntax#Interfaces)
		- [6.3.1Implementing an interface](https://en.wikipedia.org/wiki/Java_syntax#Implementing_an_interface)
		- [6.3.2Functional interfaces and lambda expressions](https://en.wikipedia.org/wiki/Java_syntax#Functional_interfaces_and_lambda_expressions)
		- [6.3.3Method references](https://en.wikipedia.org/wiki/Java_syntax#Method_references)
		- [6.3.4Inheritance](https://en.wikipedia.org/wiki/Java_syntax#Inheritance_2)
		- [6.3.5Default methods](https://en.wikipedia.org/wiki/Java_syntax#Default_methods)
		- [6.3.6Static methods](https://en.wikipedia.org/wiki/Java_syntax#Static_methods)
		- [6.3.7Private methods](https://en.wikipedia.org/wiki/Java_syntax#Private_methods)
		- [6.3.8Annotations](https://en.wikipedia.org/wiki/Java_syntax#Annotations)
			* [6.3.8.1Annotation types](https://en.wikipedia.org/wiki/Java_syntax#Annotation_types)
			* [6.3.8.2Usage of annotations](https://en.wikipedia.org/wiki/Java_syntax#Usage_of_annotations)
* [7Generics](https://en.wikipedia.org/wiki/Java_syntax#Generics)
	+ [7.1Generic classes](https://en.wikipedia.org/wiki/Java_syntax#Generic_classes)
	+ [7.2Generic methods and constructors](https://en.wikipedia.org/wiki/Java_syntax#Generic_methods_and_constructors)
	+ [7.3Generic interfaces](https://en.wikipedia.org/wiki/Java_syntax#Generic_interfaces)
* [8See also](https://en.wikipedia.org/wiki/Java_syntax#See_also)
* [9References](https://en.wikipedia.org/wiki/Java_syntax#References)
* [10External links](https://en.wikipedia.org/wiki/Java_syntax#External_links)

Basics[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=1)]

**Identifier**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=2)]

An [identifier](https://en.wikipedia.org/wiki/Identifier#In_computer_languages) is the name of an element in the [code](https://en.wikipedia.org/wiki/Source_code). There are certain standard [naming conventions](https://en.wikipedia.org/wiki/Naming_conventions_%28programming%29) to follow when selecting names for elements. Identifiers in Java are [case-sensitive](https://en.wikipedia.org/wiki/Case_sensitivity).

An identifier can contain:

* Any Unicode character that is a letter (including numeric letters like [Roman numerals](https://en.wikipedia.org/wiki/Roman_numerals)) or digit.
* [Currency sign](https://en.wikipedia.org/wiki/Currency_sign) (such as ¥).
* Connecting punctuation character (such as [\_](https://en.wikipedia.org/wiki/Underscore)).

An identifier cannot:

* Start with a digit.
* Be equal to a reserved keyword, null literal or boolean literal.

**Keywords**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=3)]

*Main article:*[*Java keywords*](https://en.wikipedia.org/wiki/Java_keywords)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | var |
| class | finally | long | strictfp | void |
| const | float | native | super | volatile |
| while |  |  |  |  |

**Literals**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=4)]

|  |
| --- |
| **Integers** |
| [**binary**](https://en.wikipedia.org/wiki/Binary_numeral_system)**(introduced in Java SE 7)** | 0b11110101 (0b followed by a binary number) |
| [**octal**](https://en.wikipedia.org/wiki/Octal) | 0365 (0 followed by an octal number) |
| [**hexadecimal**](https://en.wikipedia.org/wiki/Hexadecimal) | 0xF5 (0x followed by a hexadecimal number) |
| [**decimal**](https://en.wikipedia.org/wiki/Decimal) | 245 (decimal number) |
| [**Floating-point**](https://en.wikipedia.org/wiki/Floating-point)**values** |
| **float** | 23.5F, .5f, 1.72E3F (decimal fraction with an optional exponent indicator, followed by F) |
| 0x.5FP0F, 0x.5P-6f (0x followed by a hexadecimal fraction with a mandatory exponent indicator and a suffix F) |
| **double** | 23.5D, .5, 1.72E3D (decimal fraction with an optional exponent indicator, followed by optional D) |
| 0x.5FP0, 0x.5P-6D (0x followed by a hexadecimal fraction with a mandatory exponent indicator and an optional suffix D) |
| **Character literals** |
| **char** | 'a', 'Z', '\u0231' (character or a character escape, enclosed in single quotes) |
| **Boolean literals** |
| **boolean** | true, false |
| **null literal** |
| **null reference** | null |
| **String literals** |
| **String** | "Hello, World" (sequence of characters and character escapes enclosed in double quotes) |
| **Characters escapes in strings** |
| [**Unicode**](https://en.wikipedia.org/wiki/Unicode)**character** | \u3876 (\u followed by the hexadecimal unicode code point up to U+FFFF) |
| [**Octal**](https://en.wikipedia.org/wiki/Octal)**escape** | \352 (octal number not exceeding 377, preceded by backslash) |
| [**Line feed**](https://en.wikipedia.org/wiki/Line_feed) | \n |
| [**Carriage return**](https://en.wikipedia.org/wiki/Carriage_return) | \r |
| [**Form feed**](https://en.wikipedia.org/wiki/Form_feed) | \f |
| [**Backslash**](https://en.wikipedia.org/wiki/Backslash) | \\ |
| [**Single quote**](https://en.wikipedia.org/wiki/Single_quote) | \' |
| [**Double quote**](https://en.wikipedia.org/wiki/Double_quote) | \" |
| [**Tab**](https://en.wikipedia.org/wiki/Tab_character) | \t |
| [**Backspace**](https://en.wikipedia.org/wiki/Backspace) | \b |

Integer literals are of int type by default unless long type is specified by appending L or l suffix to the literal, e.g. 367L. Since Java SE 7, it is possible to include underscores between the digits of a number to increase readability; for example, a number 145608987 can be written as 145\_608\_987.

**Variables**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=5)]

[Variables](https://en.wikipedia.org/wiki/Variable_%28programming%29) are identifiers associated with values. They are declared by writing the variable's type and name, and are optionally initialized in the same statement by assigning a value.

int count; *//Declaring an uninitialized variable called 'count', of type 'int'*

count = 35; *//Initializing the variable*

int count = 35; *//Declaring and initializing the variable at the same time*

Multiple variables of the same type can be declared and initialized in one statement using comma as a delimiter.

int a, b; *//Declaring multiple variables of the same type*

int a = 2, b = 3; *//Declaring and initializing multiple variables of the same type*

Since Java 10 it has become possible to infer types for the variables automatically by using var.

*// stream will have the FileOutputStream type as inferred from its initializer*

**var** stream = **new** FileOutputStream("file.txt");

*// An equivalent declaration with an explicit type*

FileOutputStream stream = **new** FileOutputStream("file.txt");

**Code blocks**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=6)]

The separators { and } signify a code block and a new scope. Class members and the body of a method are examples of what can live inside these braces in various contexts.

Inside of method bodies, braces may be used to create new scopes, as follows:

void doSomething() {

 int a;

 {

 int b;

 a = 1;

 }

 a = 2;

 b = 3; *// Illegal because the variable b is declared in an inner scope..*

}

**Comments**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=7)]

Java has three kinds of comments: *traditional comments*, *end-of-line comments* and *documentation comments*.

Traditional comments, also known as block comments, start with /\* and end with \*/, they may span across multiple lines. This type of comment was derived from C and C++.

*/\* This is a multi-line comment.*

*It may occupy more than one line. \*/*

End-of-line comments start with // and extend to the end of the current line. This comment type is also present in C++ and in modern C.

*// This is an end-of-line comment*

Documentation comments in the source files are processed by the [Javadoc](https://en.wikipedia.org/wiki/Javadoc) tool to generate documentation. This type of comment is identical to traditional comments, except it starts with /\*\* and follows conventions defined by the Javadoc tool. Technically, these comments are a special kind of traditional comment and they are not specifically defined in the language specification.

*/\*\**

 *\* This is a documentation comment.*

 *\**

 *\* @author John Doe*

 *\*/*

**Universal types**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=8)]

Classes in the package java.lang are implicitly imported into every program, as long as no explicitly-imported types have the same names. Important ones include:

**java.lang.Object**

Java's [top type](https://en.wikipedia.org/wiki/Top_type). Superclass of all classes that do not declare a parent class. All values can be converted to this type, although for primitive values this involves [autoboxing](https://en.wikipedia.org/wiki/Object_type_%28object-oriented_programming%29#Autoboxing).

**java.lang.String**

Java's basic string type. [Immutable](https://en.wikipedia.org/wiki/Immutable_object). Some methods treat each [UTF-16](https://en.wikipedia.org/wiki/UTF-16) code unit as a "character", but methods to convert to an int[] that is effectively [UTF-32](https://en.wikipedia.org/wiki/UTF-32) are also available.

**java.lang.Throwable**

supertype of everything that can be [thrown or caught](https://en.wikipedia.org/wiki/Exception_handling) with Java's throw and catch statements.

Program structure[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=9)]

Java applications consist of collections of classes. Classes exist in packages but can also be nested inside other classes.

**main method**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=10)]

Every Java application must have an entry point. This is true of both graphical interface applications and console applications. The entry point is the main method. There can be more than one class with a main method, but the main class is always defined externally (for example, in a [manifest file](https://en.wikipedia.org/wiki/Manifest_file)). The method must be static and is passed command-line arguments as an array of strings. Unlike [C++](https://en.wikipedia.org/wiki/C%2B%2B) or [C#](https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29), it never returns a value and must return void.

**public** **static** void main(String[] args) {

}

**Packages**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=11)]

Packages are a part of a class name and they are used to group and/or distinguish named entities from other ones. Another purpose of packages is to govern code access together with access modifiers. For example, java.io.InputStream is a fully qualified class name for the class InputStream which is located in the package java.io.

A package is declared at the start of the file with the package declaration:

**package** **myapplication.mylibrary**;

**public** **class** **MyClass** {

}

Classes with the public modifier must be placed in the files with the same name and java extension and put into nested folders corresponding to the package name. The above class myapplication.mylibrary.MyClass will have the following path: myapplication/mylibrary/MyClass.java.

**Import declaration**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=12)]

**Type import declaration**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=13)]

A type import declaration allows a named type to be referred to by a simple name rather than the full name that includes the package. Import declarations can be *single type import declarations* or *import-on-demand declarations*. Import declarations must be placed at the top of a code file after the package declaration.

**package** **myPackage**;

**import** **java.util.Random**; *// Single type declaration*

**public** **class** **ImportsTest** {

 **public** **static** void main(String[] args) {

 */\* The following line is equivalent to*

 *\* java.util.Random random = new java.util.Random();*

 *\* It would've been incorrect without the import.*

 *\*/*

 Random random = **new** Random();

 }

}

Import-on-demand declarations are mentioned in the code. A "type import" imports all the types of the package. A "static import" imports members of the package.

**import** **java.util.\***; */\*This form of importing classes makes all classes*

 *in package java.util available by name, could be used instead of the*

 *import declaration in the previous example. \*/*

**import** **java.\***; */\*This statement is legal, but does nothing, since there*

 *are no classes directly in package java. All of them are in packages*

 *within package java. This does not import all available classes.\*/*

**Static import declaration**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=14)]

This type of declaration has been available since [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). [Static import](https://en.wikipedia.org/wiki/Static_imports) declarations allow access to static members defined in another class, interface, annotation, or enum; without specifying the class name:

**import static** **java.lang.System.out**; *//'out' is a static field in java.lang.System*

**public** **class** **HelloWorld** {

 **public** **static** void main(String[] args) {

 */\* The following line is equivalent to:*

 *System.out.println("Hi World!");*

 *and would have been incorrect without the import declaration. \*/*

 out.println("Hello World!");

 }

}

Import-on-demand declarations allow to import all the fields of the type:

**import static** **java.lang.System.\***;

 */\* This form of declaration makes all*

 *fields in the java.lang.System class available by name, and may be used instead*

 *of the import declaration in the previous example. \*/*

Enum constants may also be used with static import. For example, this enum is in the package called screen:

**public** **enum** ColorName {

 RED, BLUE, GREEN

};

It is possible to use static import declarations in another class to retrieve the enum constants:

**import** **screen.ColorName**;

**import static** **screen.ColorName.\***;

**public** **class** **Dots** {

 */\* The following line is equivalent to 'ColorName foo = ColorName.RED',*

 *and it would have been incorrect without the static import. \*/*

 ColorName foo = RED;

 void shift() {

 */\* The following line is equivalent to:*

 *if (foo == ColorName.RED) foo = ColorName.BLUE; \*/*

 **if** (foo == RED) foo = BLUE;

 }

}

Operators[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=15)]

Operators in Java are similar to those in [C++](https://en.wikipedia.org/wiki/C%2B%2B). However, there is no delete operator due to [garbage collection](https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29) mechanisms in Java, and there are no operations on [pointers](https://en.wikipedia.org/wiki/Pointer_%28computer_programming%29) since Java does not support them. Another difference is that Java has an unsigned right shift operator (>>>), while C's right shift operator's signedness is type-dependent. Operators in Java cannot be [overloaded](https://en.wikipedia.org/wiki/Operator_overloading).

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| **1** | () | Method invocation | Left-to-right |
| [] | Array access |
| . | Class member selection |
| **2** | ++ -- | Postfix increment and decrement[[1]](https://en.wikipedia.org/wiki/Java_syntax#cite_note-1) |
| **3** | ++ -- | Prefix increment and decrement | Right-to-left |
| + - | Unary plus and minus |
| ! ~ | Logical NOT and bitwise NOT |
| (*type*) val | Type cast |
| new | Class instance or array creation |
| **4** | \* / % | Multiplication, division, and modulus (remainder) | Left-to-right |
| **5** | + - | Addition and subtraction |
| + | String concatenation |
| **6** | << >> >>> | [Bitwise](https://en.wikipedia.org/wiki/Bitwise_operation) left shift, signed right shift and unsigned right shift |
| **7** | < <= | [Relational](https://en.wikipedia.org/wiki/Relational_operator) “less than” and “less than or equal to” |
| > >= | Relational “greater than” and “greater than or equal to” |
| instanceof | Type comparison |
| **8** | == != | Relational “equal to” and “not equal to” |
| **9** | & | Bitwise and logical AND |
| **10** | ^ | Bitwise and logical XOR (exclusive or) |
| **11** | | | Bitwise and logical OR (inclusive or) |
| **12** | && | Logical conditional-AND |
| **13** | || | Logical conditional-OR |
| **14** | *c* ? *t* : *f* | [Ternary](https://en.wikipedia.org/wiki/Ternary_operator) conditional (see [?:](https://en.wikipedia.org/wiki/%3F%3A)) | Right-to-left |
| **15** | = | Simple assignment |
| += -= | Assignment by sum and difference |
| \*= /= %= | Assignment by product, quotient, and remainder |
| <<= >>= >>>= | Assignment by bitwise left shift, signed right shift and unsigned right shift |
| &= ^= |= | Assignment by bitwise AND, XOR, and OR |

Control structures[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=16)]

**Conditional statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=17)]

**if statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=18)]

[if statements](https://en.wikipedia.org/wiki/Conditional_%28programming%29) in Java are similar to those in C and use the same syntax:

**if** (i == 3) doSomething();

if statement may include optional else block, in which case it becomes an if-then-else statement:

**if** (i == 2) {

 doSomething();

} **else** {

 doSomethingElse();

}

Like C, else-if construction does not involve any special keywords, it is formed as a sequence of separate if-then-else statements:

**if** (i == 3) {

 doSomething();

} **else** **if** (i == 2) {

 doSomethingElse();

} **else** {

 doSomethingDifferent();

}

Also, note that the [?:](https://en.wikipedia.org/wiki/%3F%3A) operator can be used in place of simple if statement, for example

int a = 1;

int b = 2;

int minVal = (a < b) ? a : b;

**switch statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=19)]

[Switch statements](https://en.wikipedia.org/wiki/Switch_statement) in Java can use byte, short, char, and int (note: not long) primitive data types or their corresponding wrapper types. Starting with J2SE 5.0, it is possible to use [enum types](https://en.wikipedia.org/wiki/Enumerated_type%22%20%5Co%20%22Enumerated%20type). Starting with Java SE 7, it is possible to use Strings. Other reference types cannot be used in switch statements.

Possible values are listed using case labels. These labels in Java may contain only constants (including enum constants and string constants). Execution will start after the label corresponding to the expression inside the brackets. An optional default label may be present to declare that the code following it will be executed if none of the case labels correspond to the expression.

Code for each label ends with the break keyword. It is possible to omit it causing the execution to proceed to the next label, however, a warning will usually be reported during compilation.

**switch** (ch) {

 **case** 'A':

 doSomething(); *// Triggered if ch == 'A'*

 **break**;

 **case** 'B':

 **case** 'C':

 doSomethingElse(); *// Triggered if ch == 'B' or ch == 'C'*

 **break**;

 **default**:

 doSomethingDifferent(); *// Triggered in any other case*

 **break**;

}

**switch expressions**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=20)]

Since Java 14 it has become possible to use switch expressions, which use the new arrow syntax:

**var** result = **switch** (ch) {

 **case** 'A' -> Result.GREAT;

 **case** 'B', 'C' -> Result.FINE;

 **default** -> **throw** **new** ThisIsNoGoodException();

};

Alternatively, there is a possibility to express the same with the yield statement, although it is recommended to prefer the arrow syntax because it avoids the problem of accidental fall throughs.

**var** result = **switch** (ch) {

 **case** 'A':

 yield Result.GREAT;

 **case** 'B':

 **case** 'C':

 yield Result.FINE;

 **default**:

 **throw** **new** ThisIsNoGoodException();

};

**Iteration statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=21)]

Iteration statements are statements that are repeatedly executed when a given condition is evaluated as true. Since [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0), Java has four forms of such statements.

**while loop**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=22)]

In the while loop, the test is done before each iteration.

**while** (i < 10) {

 doSomething();

}

**do ... while loop**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=23)]

In the do ... while loop, the test is done after each iteration. Consequently, the code is always executed at least once.

*// doSomething() is called at least once*

**do** {

 doSomething();

} **while** (i < 10);

**for loop**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=24)]

for loops in Java include an initializer, a condition and a counter expression. It is possible to include several expressions of the same kind using comma as delimiter (except in the condition). However, unlike C, the comma is just a delimiter and not an operator.

**for** (int i = 0; i < 10; i++) {

 doSomething();

}

*// A more complex loop using two variables*

**for** (int i = 0, j = 9; i < 10; i++, j -= 3) {

 doSomething();

}

Like C, all three expressions are optional. The following loop is infinite:

**for** (;;) {

 doSomething();

}

**Enhanced for loop**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=25)]

[Enhanced for loops](https://en.wikipedia.org/wiki/Enhanced_for_loop) have been available since [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). This type of loop uses built-in iterators over arrays and collections to return each item in the given collection. Every element is returned and reachable in the context of the code block. When the block is executed, the next item is returned until there are no items remaining. Unlike [C#](https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29), this kind of loop does not involve a special keyword, but instead uses a different notation style.

**for** (int i : intArray) {

 doSomething(i);

}

**Jump statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=26)]

**Labels**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=27)]

Labels are given points in code used by break and continue statements. Note that the Java goto keyword cannot be used to jump to specific points in the code.

start:

someMethod();

**break statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=28)]

The break statement breaks out of the closest loop or switch statement. Execution continues in the statement after the terminated statement, if any.

**for** (int i = 0; i < 10; i++) {

 **while** (**true**) {

 **break**;

 }

 *// Will break to this point*

}

It is possible to break out of the outer loop using labels:

outer:

**for** (int i = 0; i < 10; i++) {

 **while** (**true**) {

 **break** outer;

 }

}

*// Will break to this point*

**continue statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=29)]

The continue statement discontinues the current iteration of the current control statement and begins the next iteration. The following while loop in the code below reads characters by calling getChar(), skipping the statements in the body of the loop if the characters are spaces:

int ch;

**while** (ch == getChar()) {

 **if** (ch == ' ') {

 **continue**; *// Skips the rest of the while-loop*

 }

 *// Rest of the while-loop, will not be reached if ch == ' '*

 doSomething();

}

Labels can be specified in continue statements and break statements:

outer:

**for** (String str : stringsArr) {

 char[] strChars = str.toCharArray();

 **for** (char ch : strChars) {

 **if** (ch == ' ') {

 */\* Continues the outer cycle and the next*

 *string is retrieved from stringsArr \*/*

 **continue** outer;

 }

 doSomething(ch);

 }

}

**return statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=30)]

The return statement is used to end method execution and to return a value. A value returned by the method is written after the return keyword. If the method returns anything but void, it must use the return statement to return some value.

void doSomething(boolean streamClosed) {

 *// If streamClosed is true, execution is stopped*

 **if** (streamClosed) {

 **return**;

 }

 readFromStream();

}

int calculateSum(int a, int b) {

 int result = a + b;

 **return** result;

}

return statement ends execution immediately, except for one case: if the statement is encountered within a try block and it is complemented by a finally, control is passed to the finally block.

void doSomething(boolean streamClosed) {

 **try** {

 **if** (streamClosed) {

 **return**;

 }

 readFromStream();

 } **finally** {

 */\* Will be called last even if*

 *readFromStream() was not called \*/*

 freeResources();

 }

}

**Exception handling statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=31)]

**try-catch-finally statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=32)]

Exceptions are managed within try ... catch blocks.

**try** {

 *// Statements that may throw exceptions*

 methodThrowingExceptions();

} **catch** (Exception ex) {

 *// Exception caught and handled here*

 reportException(ex);

} **finally** {

 *// Statements always executed after the try/catch blocks*

 freeResources();

}

The statements within the try block are executed, and if any of them throws an exception, execution of the block is discontinued and the exception is handled by the catch block. There may be multiple catch blocks, in which case the first block with an exception variable whose type matches the type of the thrown exception is executed.

Java SE 7 also introduced multi-catch clauses besides uni-catch clauses. This type of catch clauses allows Java to handle different types of exceptions in a single block provided they are not subclasses of each other.

**try** {

 methodThrowingExceptions();

} **catch** (IOException | IllegalArgumentException ex) {

 *//Both IOException and IllegalArgumentException will be caught and handled here*

 reportException(ex);

}

If no catch block matches the type of the thrown exception, the execution of the outer block (or method) containing the try ... catch statement is discontinued, and the exception is passed up and outside the containing block (or method). The exception is propagated upwards through the [call stack](https://en.wikipedia.org/wiki/Call_stack) until a matching catch block is found within one of the currently active methods. If the exception propagates all the way up to the top-most main method without a matching catch block being found, a textual description of the exception is written to the standard output stream.

The statements within the finally block are always executed after the try and catch blocks, whether or not an exception was thrown and even if a return statement was reached. Such blocks are useful for providing clean-up code that is guaranteed to always be executed.

The catch and finally blocks are optional, but at least one or the other must be present following the try block.

**try-with-resources statements**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=33)]

try-with-resources statements are a special type of try-catch-finally statements introduced as an implementation of the [dispose pattern](https://en.wikipedia.org/wiki/Dispose_pattern) in Java SE 7. In a try-with-resources statement the try keyword is followed by initialization of one or more resources that are released automatically when the try block execution is finished. Resources must implement java.lang.AutoCloseable. try-with-resources statements are not required to have a catch or finally block unlike normal try-catch-finally statements.

**try** (FileOutputStream fos = **new** FileOutputStream("filename");

 XMLEncoder xEnc = **new** XMLEncoder(fos)) {

 xEnc.writeObject(object);

} **catch** (IOException ex) {

 Logger.getLogger(Serializer.class.getName()).log(Level.SEVERE, **null**, ex);

}

Since Java 9 it is possible to use already declared variables:

FileOutputStream fos = **new** FileOutputStream("filename");

XMLEncoder xEnc = **new** XMLEncoder(fos);

**try** (fos; xEnc) {

 xEnc.writeObject(object);

} **catch** (IOException ex) {

 Logger.getLogger(Serializer.class.getName()).log(Level.SEVERE, **null**, ex);

}

**throw statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=34)]

The throw statement is used to throw an exception and end the execution of the block or method. The thrown exception instance is written after the throw statement.

void methodThrowingExceptions(Object obj) {

 **if** (obj == **null**) {

 *// Throws exception of NullPointerException type*

 **throw** **new** NullPointerException();

 }

 *// Will not be called, if object is null*

 doSomethingWithObject(obj);

}

**Thread concurrency control**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=35)]

Java has built-in tools for [multi-thread programming](https://en.wikipedia.org/wiki/Thread_%28computer_science%29). For the purposes of thread [synchronization](https://en.wikipedia.org/wiki/Synchronization_%28computer_science%29) the synchronized statement is included in Java language.

To make a code block synchronized, it is preceded by the synchronized keyword followed by the lock object inside the brackets. When the executing thread reaches the synchronized block, it acquires a [mutual exclusion](https://en.wikipedia.org/wiki/Mutual_exclusion) lock, executes the block, then releases the lock. No threads may enter this block until the lock is released. Any non-null reference type may be used as the lock.

*/\* Acquires lock on someObject. It must be of*

*a reference type and must be non-null \*/*

**synchronized** (someObject) {

 *// Synchronized statements*

}

**assert statement**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=36)]

assert statements have been available since [J2SE 1.4](https://en.wikipedia.org/wiki/J2SE_1.4). These types of statements are used to make [assertions](https://en.wikipedia.org/wiki/Assertion_%28computing%29) in the source code, which can be turned on and off during execution for specific classes or packages. To declare an assertion the assert keyword is used followed by a conditional expression. If it evaluates to false when the statement is executed, an exception is thrown. This statement can include a colon followed by another expression, which will act as the exception's detail message.

*// If n equals 0, AssertionError is thrown*

**assert** n != 0;

*/\* If n equals 0, AssertionError will be thrown*

*with the message after the colon \*/*

**assert** n != 0 : "n was equal to zero";

Primitive types[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=37)]

Primitive types in Java include integer types, floating-point numbers, [UTF-16](https://en.wikipedia.org/wiki/UTF-16) code units and a boolean type. There are no unsigned types in Java except char type, which is used to represent UTF-16 code units. The lack of unsigned types is offset by introducing unsigned right shift operation (>>>), which is not present in C++. Nevertheless, criticisms have been leveled about the lack of compatibility with C and C++ this causes.[[2]](https://en.wikipedia.org/wiki/Java_syntax#cite_note-2)

|  |
| --- |
| **Primitive Types** |
| **Type Name** | **Wrapper class** | **Value** | **Range** | **Size** | **Default Value** |
| byte | java.lang.Byte | integer | −128 through +127 | 8-bit (1-byte) | 0 |
| short | java.lang.Short | integer | −32,768 through +32,767 | 16-bit (2-byte) | 0 |
| int | java.lang.Integer | integer | −2,147,483,648 through +2,147,483,647 | 32-bit (4-byte) | 0 |
| long | java.lang.Long | integer | −9,223,372,036,854,775,808 through+9,223,372,036,854,775,807 | 64-bit (8-byte) | 0 |
| float | java.lang.Float | floating point number | ±1.401298E−45 through ±3.402823E+38 | 32-bit (4-byte) | 0.0f[[3]](https://en.wikipedia.org/wiki/Java_syntax#cite_note-3) |
| double | java.lang.Double | floating point number | ±4.94065645841246E−324 through±1.79769313486232E+308 | 64-bit (8-byte) | 0.0 |
| boolean | java.lang.Boolean | Boolean | true or false | 1-bit (1-bit) | false |
| char | java.lang.Character | [UTF-16](https://en.wikipedia.org/wiki/UTF-16) code unit ([BMP](https://en.wikipedia.org/wiki/Mapping_of_Unicode_character_planes#Basic_Multilingual_Plane) characteror a part of a surrogate pair) | '\u0000' through '\uFFFF' | 16-bit (2-byte) | '\u0000' |

char does not necessarily correspond to a single character. It may represent a part of a [surrogate pair](https://en.wikipedia.org/wiki/UTF-16#Encoding_of_characters_outside_the_BMP), in which case Unicode code point is represented by a sequence of two char values.

**Boxing and unboxing**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=38)]

This language feature was introduced in [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). *Boxing* is the operation of converting a value of a primitive type into a value of a corresponding reference type, which serves as a wrapper for this particular primitive type. *Unboxing* is the reverse operation of converting a value of a reference type (previously boxed) into a value of a corresponding primitive type. Neither operation requires an explicit conversion.

Example:

int foo = 42; *// Primitive type*

Integer bar = foo; */\* foo is boxed to bar, bar is of Integer type,*

 *which serves as a wrapper for int \*/*

int foo2 = bar; *// Unboxed back to primitive type*

Reference types[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=39)]

Reference types include class types, interface types, and array types. When the constructor is called, an object is created on the heap and a reference is assigned to the variable. When a variable of an object gets out of scope, the reference is broken and when there are no references left, the object gets marked as garbage. The garbage collector then collects and destroys it some time afterwards.

A reference variable is null when it does not reference any object.

**Arrays**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=40)]

Arrays in Java are created at runtime, just like class instances. Array length is defined at creation and cannot be changed.

int[] numbers = **new** int[5];

numbers[0] = 2;

numbers[1] = 5;

int x = numbers[0];

**Initializers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=41)]

*// Long syntax*

int[] numbers = **new** int[] {20, 1, 42, 15, 34};

*// Short syntax*

int[] numbers2 = {20, 1, 42, 15, 34};

**Multi-dimensional arrays**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=42)]

In Java, multi-dimensional arrays are represented as arrays of arrays. Technically, they are represented by arrays of references to other arrays.

int[][] numbers = **new** int[3][3];

numbers[1][2] = 2;

int[][] numbers2 = {{2, 3, 2}, {1, 2, 6}, {2, 4, 5}};

Due to the nature of the multi-dimensional arrays, sub-arrays can vary in length, so multi-dimensional arrays are not bound to be rectangular unlike C:

int[][] numbers = **new** int[2][]; *//Initialization of the first dimension only*

numbers[0] = **new** int[3];

numbers[1] = **new** int[2];

**Classes**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=43)]

[Classes](https://en.wikipedia.org/wiki/Class_%28computer_science%29) are fundamentals of an object-oriented language such as Java. They contain members that store and manipulate data. Classes are divided into *top-level* and *nested*. Nested classes are classes placed inside another class that may access the private members of the enclosing class. Nested classes include *member classes* (which may be defined with the *static* modifier for simple nesting or without it for inner classes), *local classes* and *anonymous classes*.

**Declaration**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=44)]

|  |  |
| --- | --- |
| **Top-level class** | **class** **Foo** { *// Class members*} |
| **Inner class** | **class** **Foo** { *// Top-level class* **class** **Bar** { *// Inner class* }} |
| **Nested class** | **class** **Foo** { *// Top-level class* **static** **class** **Bar** { *// Nested class* }} |
| **Local class** | **class** **Foo** { void bar() { **class** **Foobar** {*// Local class within a method* } }} |
| **Anonymous class** | **class** **Foo** { void bar() { **new** Object() {*// Creation of a new anonymous class extending Object* }; }} |

**Instantiation**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=45)]

Non-static members of a class define the types of the instance variables and methods, which are related to the objects created from that class. To create these objects, the class must be instantiated by using the new operator and calling the class constructor.

Foo foo = **new** Foo();

**Accessing members**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=46)]

Members of both instances and static classes are accessed with the . (dot) operator.

**Accessing an instance member**
Instance members can be accessed through the name of a variable.

String foo = "Hello";

String bar = foo.toUpperCase();

**Accessing a static class member**
Static members are accessed by using the name of the class or any other type. This does not require the creation of a class instance. Static members are declared using the static modifier.

**public** **class** **Foo** {

 **public** **static** void doSomething() {

 }

}

*// Calling the static method*

Foo.doSomething();

**Modifiers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=47)]

Modifiers are keywords used to modify declarations of types and type members. Most notably there is a sub-group containing the access modifiers.

* **abstract** - Specifies that a class only serves as a base class and cannot be instantiated.
* **static** - Used only for member classes, specifies that the member class does not belong to a specific instance of the containing class.
* **final** - Classes marked as final cannot be extended from and cannot have any subclasses.
* **strictfp** - Specifies that all floating-point operations must be carried out conforming to [IEEE 754](https://en.wikipedia.org/wiki/IEEE_754) and forbids using enhanced precision to store intermediate results.

**Access modifiers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=48)]

The *access modifiers*, or *inheritance modifiers*, set the accessibility of classes, methods, and other members. Members marked as public can be reached from anywhere. If a class or its member does not have any modifiers, default access is assumed.

**public** **class** **Foo** {

 int go() {

 **return** 0;

 }

 **private** **class** **Bar** {

 }

}

The following table shows whether code within a class has access to the class or method depending on the accessing class location and the modifier for the accessed class or class member:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | **Same class or nested class** | **Other class inside the same package** | **Extended Class inside another package** | **Non-extended inside another package** |
| **private** | yes | no | no | no |
| **default (package private)** | yes | yes | no | no |
| **protected** | yes | yes | yes | no |
| **public** | yes | yes | yes | yes |



This image describes the class member scope within classes and packages.

**Constructors and initializers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=49)]

A [constructor](https://en.wikipedia.org/wiki/Constructor_%28computer_science%29) is a special method called when an object is initialized. Its purpose is to initialize the members of the object. The main differences between constructors and ordinary methods are that constructors are called only when an instance of the class is created and never return anything. Constructors are declared as common methods, but they are named after the class and no return type is specified:

**class** **Foo** {

 String str;

 Foo() { *// Constructor with no arguments*

 *// Initialization*

 }

 Foo(String str) { *// Constructor with one argument*

 **this**.str = str;

 }

}

Initializers are blocks of code that are executed when a class or an instance of a class is created. There are two kinds of initializers, *static initializers* and *instance initializers*.

Static initializers initialize static fields when the class is created. They are declared using the static keyword:

**class** **Foo** {

 **static** {

 *// Initialization*

 }

}

A class is created only once. Therefore, static initializers are not called more than once. On the contrary, instance initializers are automatically called before the call to a constructor every time an instance of the class is created. Unlike constructors instance initializers cannot take any arguments and generally they cannot throw any [checked exceptions](https://en.wikipedia.org/wiki/Exception_handling#Checked_exceptions) (except in several special cases). Instance initializers are declared in a block without any keywords:

**class** **Foo** {

 {

 *// Initialization*

 }

}

Since Java has a garbage collection mechanism, there are no [destructors](https://en.wikipedia.org/wiki/Destructor_%28computer_science%29). However, every object has a finalize() method called prior to garbage collection, which can be [overridden](https://en.wikipedia.org/wiki/Method_overriding) to implement finalization.

**Methods**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=50)]

All the statements in Java must reside within methods. Methods are similar to functions except they belong to classes. A method has a return value, a name and usually some parameters initialized when it is called with some arguments. Similar to C++, methods returning nothing have return type declared as void. Unlike in C++, methods in Java are not allowed to have default argument values and methods are usually overloaded instead.

**class** **Foo** {

 int bar(int a, int b) {

 **return** (a\*2) + b;

 }

 */\* Overloaded method with the same name but different set of arguments \*/*

 int bar(int a) {

 **return** a\*2;

 }

}

A method is called using . notation on an object, or in the case of a static method, also on the name of a class.

Foo foo = **new** Foo();

int result = foo.bar(7, 2); *// Non-static method is called on foo*

int finalResult = Math.abs(result); *// Static method call*

The throws keyword indicates that a method throws an exception. All checked exceptions must be listed in a comma-separated list.

void openStream() **throws** IOException, myException { *// Indicates that IOException may be thrown*

}

**Modifiers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=51)]

* **abstract** - [Abstract methods](https://en.wikipedia.org/wiki/Abstract_method) can be present only in [abstract classes](https://en.wikipedia.org/wiki/Abstract_class), such methods have no body and must be overridden in a subclass unless it is abstract itself.
* **static** - Makes the method static and accessible without creation of a class instance. However static methods cannot access non-static members in the same class.
* **final** - Declares that the method cannot be overridden in a subclass.
* **native** - Indicates that this method is implemented through [JNI](https://en.wikipedia.org/wiki/JNI) in platform-dependent code. Actual implementation happens outside Java code, and such methods have no body.
* **strictfp** - Declares strict conformance to [IEEE 754](https://en.wikipedia.org/wiki/IEEE_754) in carrying out floating-point operations.
* **synchronized** - Declares that a thread executing this method must acquire monitor. For synchronized methods the monitor is the class instance or java.lang.Class if the method is static.
* Access modifiers - Identical to those used with classes.

**Varargs**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=52)]

This language feature was introduced in [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). The last argument of the method may be declared as a variable arity parameter, in which case the method becomes a variable arity method (as opposed to fixed arity methods) or simply [varargs](https://en.wikipedia.org/wiki/Variadic_function%22%20%5Co%20%22Variadic%20function) method. This allows one to pass a variable number of values, of the declared type, to the method as parameters - including no parameters. These values will be available inside the method as an array.

void printReport(String header, int... numbers) { *//numbers represents varargs*

 System.out.println(header);

 **for** (int num : numbers) {

 System.out.println(num);

 }

}

*// Calling varargs method*

printReport("Report data", 74, 83, 25, 96);

**Fields**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=53)]

Fields, or class variables, can be declared inside the class body to store data.

**class** **Foo** {

 double bar;

}

Fields can be initialized directly when declared.

**class** **Foo** {

 double bar = 2.3;

}

**Modifiers**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=54)]

* **static** - Makes the field a static member.
* **final** - Allows the field to be initialized only once in a constructor or inside initialization block or during its declaration, whichever is earlier.
* **transient** - Indicates that this field will not be stored during [serialization](https://en.wikipedia.org/wiki/Serialization).
* **volatile** - If a field is declared volatile, it is ensured that all threads see a consistent value for the variable.

**Inheritance**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=55)]

Classes in Java can only [inherit](https://en.wikipedia.org/wiki/Inheritance_%28computer_science%29) from *one* class. A class can be derived from any class that is not marked as final. Inheritance is declared using the extends keyword. A class can reference itself using the this keyword and its direct superclass using the super keyword.

**class** **Foo** {

}

**class** **Foobar** **extends** Foo {

}

If a class does not specify its superclass, it implicitly inherits from java.lang.Object class. Thus all classes in Java are subclasses of Object class.

If the superclass does not have a constructor without parameters the subclass must specify in its constructors what constructor of the superclass to use. For example:

**class** **Foo** {

 **public** Foo(int n) {

 *// Do something with n*

 }

}

**class** **Foobar** **extends** Foo {

 **private** int number;

 *// Superclass does not have constructor without parameters*

 *// so we have to specify what constructor of our superclass to use and how*

 **public** Foobar(int number) {

 **super**(number);

 **this**.number = number;

 }

}

**Overriding methods**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=56)]

Unlike C++, all non-final methods in Java are [virtual](https://en.wikipedia.org/wiki/Virtual_function) and can be overridden by the inheriting classes.

**class** **Operation** {

 **public** int doSomething() {

 **return** 0;

 }

}

**class** **NewOperation** **extends** Operation {

 @Override

 **public** int doSomething() {

 **return** 1;

 }

}

**Abstract classes**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=57)]

An [Abstract Class](http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.1.1.1) is a class that is incomplete, or to be considered incomplete. Normal classes may have abstract methods, that is, methods that are declared but not yet implemented, only if they are abstract classes. A class C has abstract methods if any of the following is true:

* C explicitly contains a declaration of an abstract method.
* Any of C's superclasses has an abstract method and C neither declares nor inherits a method that implements it.
* A direct superinterface of C declares or inherits a method (which is therefore necessarily abstract) and C neither declares nor inherits a method that implements it.
* A subclass of an abstract class that is not itself abstract may be instantiated, resulting in the execution of a constructor for the abstract class and, therefore, the execution of the field initializers for instance variables of that class.

**package** **org.dwwwp.test**;

*/\*\**

 *\* @author jcrypto*

 *\*/*

**public** **class** **AbstractClass** {

 **private** **static** **final** String hello;

 **static** {

 System.out.println(AbstractClass.class.getName() + ": static block runtime");

 hello = "hello from " + AbstractClass.class.getName();

 }

 {

 System.out.println(AbstractClass.class.getName() + ": instance block runtime");

 }

 **public** AbstractClass() {

 System.out.println(AbstractClass.class.getName() + ": constructor runtime");

 }

 **public** **static** void hello() {

 System.out.println(hello);

 }

}

**package** **org.dwwwp.test**;

*/\*\**

 *\* @author jcrypto*

 *\*/*

**public** **class** **CustomClass** **extends** AbstractClass {

 **static** {

 System.out.println(CustomClass.class.getName() + ": static block runtime");

 }

 {

 System.out.println(CustomClass.class.getName() + ": instance block runtime");

 }

 **public** CustomClass() {

 System.out.println(CustomClass.class.getName() + ": constructor runtime");

 }

 **public** **static** void main(String[] args) {

 CustomClass nc = **new** CustomClass();

 hello();

 *//AbstractClass.hello();//also valid*

 }

}

Output:

org.dwwwp.test.AbstractClass: static block runtime

org.dwwwp.test.CustomClass: static block runtime

org.dwwwp.test.AbstractClass: instance block runtime

org.dwwwp.test.AbstractClass: constructor runtime

org.dwwwp.test.CustomClass: instance block runtime

org.dwwwp.test.CustomClass: constructor runtime

hello from org.dwwwp.test.AbstractClass

**Enumerations**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=58)]

This language feature was introduced in [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). Technically enumerations are a kind of class containing enum constants in its body. Each enum constant defines an instance of the enum type. Enumeration classes cannot be instantiated anywhere except in the enumeration class itself.

**enum** Season {

 WINTER, SPRING, SUMMER, AUTUMN

}

Enum constants are allowed to have constructors, which are called when the class is loaded:

**public** **enum** Season {

 WINTER("Cold"), SPRING("Warmer"), SUMMER("Hot"), AUTUMN("Cooler");

 Season(String description) {

 **this**.description = description;

 }

 **private** **final** String description;

 **public** String getDescription() {

 **return** description;

 }

}

Enumerations can have class bodies, in which case they are treated like anonymous classes extending the enum class:

**public** **enum** Season {

 WINTER {

 String getDescription() {**return** "cold";}

 },

 SPRING {

 String getDescription() {**return** "warmer";}

 },

 SUMMER {

 String getDescription() {**return** "hot";}

 },

 FALL {

 String getDescription() {**return** "cooler";}

 };

}

**Interfaces**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=59)]

Interfaces are types which contain no fields and usually define a number of methods without an actual implementation. They are useful to define a contract with any number of different implementations. Every interface is implicitly abstract. Interface methods are allowed to have a subset of access modifiers depending on the language version, strictfp, which has the same effect as for classes, and also static since Java SE 8.

**interface** **ActionListener** {

 int ACTION\_ADD = 0;

 int ACTION\_REMOVE = 1;

 void actionSelected(int action);

}

**Implementing an interface**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=60)]

An interface is implemented by a class using the implements keyword. It is allowed to implement more than one interface, in which case they are written after implements keyword in a comma-separated list. Class implementing an interface must override all its methods, otherwise it must be declared as abstract.

**interface** **RequestListener** {

 int requestReceived();

}

**class** **ActionHandler** **implements** ActionListener, RequestListener {

 **public** void actionSelected(int action) {

 }

 **public** int requestReceived() {

 }

}

*//Calling method defined by interface*

RequestListener listener = **new** ActionHandler(); */\*ActionHandler can be*

 *represented as RequestListener...\*/*

listener.requestReceived(); */\*...and thus is known to implement*

 *requestReceived() method\*/*

**Functional interfaces and lambda expressions**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=61)]

These features were introduced with the release of Java SE 8. An interface automatically becomes a functional interface if it defines only one method. In this case an implementation can be represented as a lambda expression instead of implementing it in a new class, thus greatly simplifying writing code in the [functional style](https://en.wikipedia.org/wiki/Functional_programming). Functional interfaces can optionally be annotated with the @FunctionalInterface annotation, which will tell the compiler to check whether the interface actually conforms to a definition of a functional interface.

*// A functional interface*

@FunctionalInterface

**interface** **Calculation** {

 int calculate(int someNumber, int someOtherNumber);

}

*// A method which accepts this interface as a parameter*

int runCalculation(Calculation calculation) {

 **return** calculation.calculate(1, 2);

}

*// Using a lambda to call the method*

runCalculation((number, otherNumber) -> number + otherNumber);

*// Equivalent code which uses an anonymous class instead*

runCalculation(**new** Calculation() {

 @Override

 **public** int calculate(int someNumber, int someOtherNumber) {

 **return** someNumber + someOtherNumber;

 }

})

Lambda's parameters types don't have to be fully specified and can be inferred from the interface it implements. Lambda's body can be written without a body block and a return statement if it is only an expression. Also, for those interfaces which only have a single parameter in the method, round brackets can be omitted.

*// Same call as above, but with fully specified types and a body block*

runCalculation((int number, int otherNumber) -> {

 **return** number + otherNumber;

});

*// A functional interface with a method which has only a single parameter*

**interface** **StringExtender** {

 String extendString(String input);

}

*// Initializing a variable of this type by using a lambda*

StringExtender extender = input -> input + " Extended";

**Method references**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=62)]

It is not necessary to use lambdas when there already is a named method compatible with the interface. This method can be passed instead of a lambda using a method reference. There are several types of method references:

|  |  |  |
| --- | --- | --- |
| **Reference type** | **Example** | **Equivalent lambda** |
| Static | Integer::sum | (number, otherNumber) -> number + otherNumber |
| Bound | "LongString"::substring | index -> "LongString".substring(index) |
| Unbound | String::isEmpty | string -> string.isEmpty() |
| Class constructor | ArrayList<String>::new | capacity -> new ArrayList<String>(capacity) |
| Array constructor | String[]::new | size -> new String[size] |

The code above which calls runCalculation could be replaced with the following using the method references:

runCalculation(Integer::sum);

**Inheritance**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=63)]

Interfaces can inherit from other interfaces just like classes. Unlike classes it is allowed to inherit from multiple interfaces. However, it is possible that several interfaces have a field with the same name, in which case it becomes a single ambiguous member, which cannot be accessed.

*/\* Class implementing this interface must implement methods of both*

*ActionListener and RequestListener \*/*

**interface** **EventListener** **extends** ActionListener, RequestListener {

}

**Default methods**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=64)]

Java SE 8 introduced default methods to interfaces which allows developers to add new methods to existing interfaces without breaking compatibility with the classes already implementing the interface. Unlike regular interface methods, default methods have a body which will get called in the case if the implementing class doesn't override it.

**interface** **StringManipulator** {

 String extendString(String input);

 *// A method which is optional to implement*

 **default** String shortenString(String input) {

 **return** input.substring(1);

 }

}

*// This is a valid class despite not implementing all the methods*

**class** **PartialStringManipulator** **implements** StringManipulator {

 @Override

 **public** String extendString(String input) {

 **return** input + " Extended";

 }

}

**Static methods**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=65)]

Static methods is another language feature introduced in Java SE 8. They behave in exactly the same way as in the classes.

**interface** **StringUtils** {

 **static** String shortenByOneSymbol(String input) {

 **return** input.substring(1);

 }

}

StringUtils.shortenByOneSymbol("Test");

**Private methods**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=66)]

Private methods were added in the Java 9 release. An interface can have a method with a body marked as private, in which case it will not be visible to inheriting classes. It can be called from default methods for the purposes of code reuse.

**interface** **Logger** {

 **default** void logError() {

 log(Level.ERROR);

 }

 **default** void logInfo() {

 log(Level.INFO);

 }

 **private** void log(Level level) {

 SystemLogger.log(level.id);

 }

}

**Annotations**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=67)]

*Main article:*[*Java annotation*](https://en.wikipedia.org/wiki/Java_annotation)

Annotations in Java are a way to embed [metadata](https://en.wikipedia.org/wiki/Metadata) into code. This language feature was introduced in [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0).

**Annotation types**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=68)]

Java has a set of predefined annotation types, but it is allowed to define new ones. An annotation type declaration is a special type of an interface declaration. They are declared in the same way as the interfaces, except the interface keyword is preceded by the @ sign. All annotations are implicitly extended from java.lang.annotation.Annotation and cannot be extended from anything else.

@interface BlockingOperations {

}

Annotations may have the same declarations in the body as the common interfaces, in addition they are allowed to include enums and annotations. The main difference is that abstract method declarations must not have any parameters or throw any exceptions. Also they may have a default value, which is declared using the default keyword after the method name:

@interface BlockingOperations {

 boolean fileSystemOperations();

 boolean networkOperations() **default** **false**;

}

**Usage of annotations**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=69)]

Annotations may be used in any kind of declaration, whether it is package, class (including enums), interface (including annotations), field, method, parameter, constructor, or local variable. Also they can be used with enum constants. Annotations are declared using the @ sign preceding annotation type name, after which element-value pairs are written inside brackets. All elements with no default value must be assigned a value.

@BlockingOperations(*/\*mandatory\*/* fileSystemOperations,

*/\*optional\*/* networkOperations = **true**)

void openOutputStream() { *//Annotated method*

}

Besides the generic form, there are two other forms to declare an annotation, which are shorthands. *Marker annotation* is a short form, it is used when no values are assigned to elements:

@Unused *// Shorthand for @Unused()*

void travelToJupiter() {

}

The other short form is called *single element annotation*. It is used with annotations types containing only one element or in the case when multiple elements are present, but only one elements lacks a default value. In single element annotation form the element name is omitted and only value is written instead:

*/\* Equivalent for @BlockingOperations(fileSystemOperations = true).*

*networkOperations has a default value and*

*does not have to be assigned a value \*/*

@BlockingOperations(**true**)

void openOutputStream() {

}

Generics[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=70)]

*Main article:*[*Generics in Java*](https://en.wikipedia.org/wiki/Generics_in_Java)

[Generics](https://en.wikipedia.org/wiki/Generic_programming), or parameterized types, or [parametric polymorphism](https://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming#Parametric_Polymorphism) is one of the major features introduced in [J2SE 5.0](https://en.wikipedia.org/wiki/J2SE_5.0). Before generics were introduced, it was required to declare all the types explicitly. With generics it became possible to work in a similar manner with different types without declaring the exact types. The main purpose of generics is to ensure type safety and to detect runtime errors during compilation. Unlike C#, information on the used parameters is not available at runtime due to [type erasure](https://en.wikipedia.org/wiki/Type_erasure).[[4]](https://en.wikipedia.org/wiki/Java_syntax#cite_note-4)

**Generic classes**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=71)]

Classes can be parameterized by adding a type variable inside angle brackets (< and >) following the class name. It makes possible the use of this type variable in class members instead of actual types. There can be more than one type variable, in which case they are declared in a comma-separated list.

It is possible to limit a type variable to a subtype of some specific class or declare an interface that must be implemented by the type. In this case the type variable is appended by the extends keyword followed by a name of the class or the interface. If the variable is constrained by both class and interface or if there are several interfaces, the class name is written first, followed by interface names with & sign used as the delimiter.

*/\* This class has two type variables, T and V. T must be*

*a subtype of ArrayList and implement Formattable interface \*/*

**public** **class** **Mapper**<T **extends** ArrayList & Formattable, V> {

 **public** void add(T array, V item) {

 *// array has add method because it is an ArrayList subclass*

 array.add(item);

 }

}

When a variable of a parameterized type is declared or an instance is created, its type is written exactly in the same format as in the class header, except the actual type is written in the place of the type variable declaration.

*/\* Mapper is created with CustomList as T and Integer as V.*

*CustomList must be a subclass of ArrayList and implement Formattable \*/*

Mapper<CustomList, Integer> mapper = **new** Mapper<CustomList, Integer>();

Since Java SE 7, it is possible to use a diamond (<>) in place of type arguments, in which case the latter will be inferred. The following code in Java SE 7 is equivalent to the code in the previous example:

Mapper<CustomList, Integer> mapper = **new** Mapper<>();

When declaring a variable for a parameterized type, it is possible to use wildcards instead of explicit type names. Wildcards are expressed by writing ? sign instead of the actual type. It is possible to limit possible types to the subclasses or superclasses of some specific class by writing the extends keyword or the super keyword correspondingly followed by the class name.

*/\* Any Mapper instance with CustomList as the first parameter*

*may be used regardless of the second one.\*/*

Mapper<CustomList, ?> mapper;

mapper = **new** Mapper<CustomList, Boolean>();

mapper = **new** Mapper<CustomList, Integer>();

*/\* Will not accept types that use anything but*

*a subclass of Number as the second parameter \*/*

void addMapper(Mapper<?, ? **extends** Number> mapper) {

}

**Generic methods and constructors**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=72)]

Usage of generics may be limited to some particular methods, this concept applies to constructors as well. To declare a parameterized method, type variables are written before the return type of the method in the same format as for the generic classes. In the case of constructor, type variables are declared before the constructor name.

**class** **Mapper** {

 *// The class itself is not generic, the constructor is*

 <T, V> Mapper(T array, V item) {

 }

}

*/\* This method will accept only arrays of the same type as*

*the searched item type or its subtype\*/*

**static** <T, V **extends** T> boolean contains(T item, V[] arr) {

 **for** (T currentItem : arr) {

 **if** (item.equals(currentItem)) {

 **return** **true**;

 }

 }

 **return** **false**;

}

**Generic interfaces**[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=73)]

Interfaces can be parameterized in the similar manner as the classes.

**interface** **Expandable**<T **extends** Number> {

 void addItem(T item);

}

*// This class is parameterized*

**class** **Array**<T **extends** Number> **implements** Expandable<T> {

 void addItem(T item) {

 }

}

*// And this is not and uses an explicit type instead*

**class** **IntegerArray** **implements** Expandable<Integer> {

 void addItem(Integer item) {

 }

}

See also[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=74)]

* ******[***Computer programming portal***](https://en.wikipedia.org/wiki/Portal%3AComputer_programming)
* [Java Platform, Standard Edition](https://en.wikipedia.org/wiki/Java_Platform%2C_Standard_Edition)

References[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=75)]

* 1. [**^**](https://en.wikipedia.org/wiki/Java_syntax#cite_ref-1) [*"Operators (The Java™ Tutorials > Learning the Java Language > Language Basics)"*](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html)*. docs.oracle.com. Oracle and/or its affiliates. Retrieved 2015-06-16.*
	2. [**^**](https://en.wikipedia.org/wiki/Java_syntax#cite_ref-2) *Owens, Sean.*[*"Java and unsigned int, unsigned short, unsigned byte, unsigned long, etc. (Or rather, the lack thereof)"*](http://darksleep.com/player/JavaAndUnsignedTypes.html)*.*
	3. [**^**](https://en.wikipedia.org/wiki/Java_syntax#cite_ref-3) [*"Primitive Data Types"*](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)*.*
	4. [**^**](https://en.wikipedia.org/wiki/Java_syntax#cite_ref-4) [Generics in the Run Time (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/f4a6ta2h.aspx)
* [Patrick Naughton](https://en.wikipedia.org/wiki/Patrick_Naughton), [Herbert Schildt](https://en.wikipedia.org/wiki/Herbert_Schildt). *Java 2: The Complete Reference*, third edition. The McGraw-Hill Companies, 1999. [ISBN](https://en.wikipedia.org/wiki/ISBN_%28identifier%29) [0-07-211976-4](https://en.wikipedia.org/wiki/Special%3ABookSources/0-07-211976-4)
* *Vermeulen; Ambler; Bumgardner; Metz; Misfeldt; Shur; Thompson (2000).*[*The Elements of Java Style*](https://archive.org/details/elementsofjavast00verm)*. Cambridge University Press.*[*ISBN*](https://en.wikipedia.org/wiki/ISBN_%28identifier%29)[*0-521-77768-2*](https://en.wikipedia.org/wiki/Special%3ABookSources/0-521-77768-2)*.*
* [*Gosling, James*](https://en.wikipedia.org/wiki/James_Gosling)*;*[*Joy, Bill*](https://en.wikipedia.org/wiki/Bill_Joy)*;*[*Steele, Guy*](https://en.wikipedia.org/wiki/Guy_Steele)*; [Bracha, Gillad](https://en.wikipedia.org/wiki/Gilad_Bracha%22%20%5Co%20%22Gilad%20Bracha) (2005).*[*Java Language Specification*](http://java.sun.com/docs/books/jls/index.html)*(3rd ed.). Addison-Wesley Professional. Retrieved 2008-12-03.*

External links[[edit](https://en.wikipedia.org/w/index.php?title=Java_syntax&action=edit&section=76)]

|  |  |
| --- | --- |
|  | Wikibooks has a book on the topic of: [***Java Programming/Syntax***](https://en.wikibooks.org/wiki/Java_Programming/Syntax) |

* [The Java Language Specification, Third edition](http://docs.oracle.com/javase/specs/) Authoritative description of the Java language
* [Java SE 10 API Javadocs](https://docs.oracle.com/javase/10/docs/api/)

|  |
| --- |
| hide* [v](https://en.wikipedia.org/wiki/Template%3AJava_%28software_platform%29)
* [t](https://en.wikipedia.org/wiki/Template_talk%3AJava_%28software_platform%29)
* [e](https://en.wikipedia.org/w/index.php?title=Template:Java_(software_platform)&action=edit)

[**Java**](https://en.wikipedia.org/wiki/Java_%28software_platform%29) |
| [**Platforms**](https://en.wikipedia.org/wiki/Computing_platform) | * [Java language](https://en.wikipedia.org/wiki/Java_%28programming_language%29)
* [JVM](https://en.wikipedia.org/wiki/Java_virtual_machine)
* [Micro Edition](https://en.wikipedia.org/wiki/Java_Platform%2C_Micro_Edition)
* [Standard Edition](https://en.wikipedia.org/wiki/Java_Platform%2C_Standard_Edition)
* [Enterprise Edition](https://en.wikipedia.org/wiki/Jakarta_EE)
* [Java Card](https://en.wikipedia.org/wiki/Java_Card)
* [Android SDK](https://en.wikipedia.org/wiki/Android_software_development#SDK)
 |  |
| **Oracle technologies** | * [Squawk](https://en.wikipedia.org/wiki/Squawk_virtual_machine)
* [Java Development Kit](https://en.wikipedia.org/wiki/Java_Development_Kit)
* [OpenJDK](https://en.wikipedia.org/wiki/OpenJDK)
* [Java virtual machine](https://en.wikipedia.org/wiki/Java_virtual_machine)
* [JavaFX](https://en.wikipedia.org/wiki/JavaFX)
* [Maxine VM](https://en.wikipedia.org/wiki/Maxine_Virtual_Machine)
 |
| **Platform technologies** | * [Applets](https://en.wikipedia.org/wiki/Java_applet)
* [Servlets](https://en.wikipedia.org/wiki/Jakarta_Servlet)
* [MIDlets](https://en.wikipedia.org/wiki/MIDlet)
* [JSP](https://en.wikipedia.org/wiki/Jakarta_Server_Pages)
* [JSF](https://en.wikipedia.org/wiki/Jakarta_Server_Faces)
* [Web Start (JNLP)](https://en.wikipedia.org/wiki/Java_Web_Start)
 |
| **Major third-party technologies** | * [Blackdown](https://en.wikipedia.org/wiki/Blackdown_Java)
* [Eclipse](https://en.wikipedia.org/wiki/Eclipse_%28software%29)
* [GNU Classpath](https://en.wikipedia.org/wiki/GNU_Classpath)
* [GWT](https://en.wikipedia.org/wiki/Google_Web_Toolkit)
* [Harmony](https://en.wikipedia.org/wiki/Apache_Harmony)
* [Hibernate](https://en.wikipedia.org/wiki/Hibernate_%28framework%29)
* [IcedTea](https://en.wikipedia.org/wiki/IcedTea)
* [Jazelle](https://en.wikipedia.org/wiki/Jazelle)
* [Spring](https://en.wikipedia.org/wiki/Spring_Framework)
* [Struts](https://en.wikipedia.org/wiki/Apache_Struts_2)
* [TopLink](https://en.wikipedia.org/wiki/Oracle_TopLink)
* [WildFly](https://en.wikipedia.org/wiki/WildFly)
 |
| **History** | * [Java version history](https://en.wikipedia.org/wiki/Java_version_history)
* [Java Community Process](https://en.wikipedia.org/wiki/Java_Community_Process)
* [Sun Microsystems](https://en.wikipedia.org/wiki/Sun_Microsystems)
* [Free Java implementations](https://en.wikipedia.org/wiki/Free_Java_implementations)
 |
| [**Major JVM languages**](https://en.wikipedia.org/wiki/List_of_JVM_languages) | * [BeanShell](https://en.wikipedia.org/wiki/BeanShell)
* [Clojure](https://en.wikipedia.org/wiki/Clojure)
* [Groovy](https://en.wikipedia.org/wiki/Groovy_%28programming_language%29)
* [JRuby](https://en.wikipedia.org/wiki/JRuby)
* [Jython](https://en.wikipedia.org/wiki/Jython)
* [Kotlin](https://en.wikipedia.org/wiki/Kotlin_%28programming_language%29)
* [Processing](https://en.wikipedia.org/wiki/Processing_%28programming_language%29)
* [Rhino](https://en.wikipedia.org/wiki/Rhino_%28JavaScript_engine%29)
* [Scala](https://en.wikipedia.org/wiki/Scala_%28programming_language%29)
 |
| [**Community**](https://en.wikipedia.org/wiki/Community_of_practice) |

|  |  |
| --- | --- |
| [**Conferences**](https://en.wikipedia.org/wiki/Professional_conference) | * [JavaOne](https://en.wikipedia.org/wiki/JavaOne)
* [Devoxx](https://en.wikipedia.org/wiki/Devoxx)
 |
| [**Organizations**](https://en.wikipedia.org/wiki/Organization) | * [Apache Software Foundation](https://en.wikipedia.org/wiki/Apache_Software_Foundation)
* [Java Community Process](https://en.wikipedia.org/wiki/Java_Community_Process)
* [Oracle Corporation](https://en.wikipedia.org/wiki/Oracle_Corporation)
* [Sun Microsystems](https://en.wikipedia.org/wiki/Sun_Microsystems), [Sun Microsystems Laboratories](https://en.wikipedia.org/wiki/Sun_Microsystems_Laboratories)
 |
| **People** | * [James Gosling](https://en.wikipedia.org/wiki/James_Gosling)
* [Arthur van Hoff](https://en.wikipedia.org/wiki/Arthur_van_Hoff)
* [Urs Hölzle](https://en.wikipedia.org/wiki/Urs_H%C3%B6lzle)
* [Patrick Naughton](https://en.wikipedia.org/wiki/Patrick_Naughton)
 |

 |
| Category [**Category**](https://en.wikipedia.org/wiki/Category%3AJava_%28programming_language%29)    [Computer programming portal](https://en.wikipedia.org/wiki/Portal%3AComputer_programming) |

[Categories](https://en.wikipedia.org/wiki/Help%3ACategory):

* [Programming language syntax](https://en.wikipedia.org/wiki/Category%3AProgramming_language_syntax)
* [Java (programming language)](https://en.wikipedia.org/wiki/Category%3AJava_%28programming_language%29)