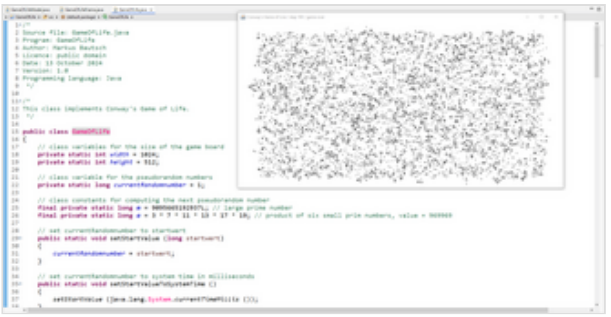
Java syntax

The **syntax of Java** is the set of rules defining how a Java program is written and interpreted.



A snippet of Java code.

The syntax is mostly derived from C and C++. Unlike C++, Java has no global functions or variables, but has data members which are also regarded as global variables. All code belongs to classes and all values are objects. The only exception is the primitive data types, which are not considered to be objects for performance reasons (though can be automatically converted to objects and vice versa via autoboxing). Some features like operator overloading or unsigned integer data types are omitted to simplify the language and avoid possible programming mistakes.

The Java syntax has been gradually extended in the course of numerous major JDK releases, and now supports abilities such as generic programming and anonymous functions (function literals, called lambda expressions in Java). Since 2017, a new JDK version is released twice a year.

Basics

The Java "Hello, World!" program program is as follows:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Since Java 25, a simplified Hello World program without an explicit class may be written:

```
void main() {
    IO.println("Hello World!");
}
```

Identifier

An identifier is the name of an element in the code. There are certain standard naming conventions to follow when selecting names for elements. Identifiers in Java are case-sensitive.

An identifier can contain:

- Any Unicode character that is a letter (including numeric letters like Roman numerals) or digit.

- Currency sign (such as ¥).
- Connecting punctuation character (such as _).

An identifier cannot:

- Start with a digit.
- Be equal to a reserved keyword, null literal or Boolean literal.

Keywords

Keywords

The following words are keywords and cannot be used as identifiers under any circumstances, of which there are 48.

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

Reserved identifiers

The following words are contextual keywords and are only restricted in certain contexts, of which there are 16.

- `exports`
- `module`
- `non-sealed`
- `open`
- `opens`
- `permits`
- `provides`
- `record`
- `requires`
- `sealed`
- `to`
- `transitive`
- `var`
- `when`
- `with`
- `yield`

Reserved words for literal values

The following words refer to literal values used by the language, of which there are 3.

- `true`
- `false`
- `null`

Unused

The following words are reserved as keywords, but currently have no use or purpose, of which there are 3.

- `const`
- `goto`
- `strictfp`

Literals

Integers	
binary (introduced in Java SE 7)	`0b11110101` (`0b` followed by a binary number)
octal	`0365` (`0` followed by an octal number)
hexadecimal	`0xF5` (`0x` followed by a hexadecimal number)
decimal	`245` (decimal number)
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`, `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 character	`\u3876` (`\u` followed by the hexadecimal unicode code point up to U+FFFF)
Octal escape	`\352` (octal number not exceeding 377, preceded by backslash)
Line feed	`\n`
Carriage return	`\r`
Form feed	`\f`
Backslash	`\\`
Single quote	`\'`
Double quote	`\"`
Tab	`\t`
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

Variables 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
```

Type inference

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

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

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 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

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.System`

`java.lang.System` is one of the most fundamental classes in Java. It is a utility class for interacting with the system, containing standard input, output, and error streams, system properties and environment variables, time, and more.

`java.lang.Object`

`java.lang.Object` is Java's top type. It is implicitly the superclass of all classes that do not declare any parent class (thus all classes in Java inherit from `Object`). All values can be converted to this type, although for primitive values this involves autoboxing.

`java.lang.Record`

All records implicitly extend `java.lang.Record`. Thus, a `record`, while treated as a `class`, cannot extend any other class.

`java.lang.Enum`

All enumerated types (enums) in Java implicitly extend `java.lang.Enum`. Thus, an `enum`, while treated as a `class`, cannot extend any other class.

`java.lang.Class<T>`

`java.lang.Class<T>` is a class that represents a `class` or `interface` in the application at runtime. It cannot be constructed via direct instantiation, but is rather created by the JVM when a class is derived from the bytes of a `.class` file. A class literal can be obtained through `.class` on such a class. For instance, `String.class` returns `Class<String>`. An unknown class being modeled can be represented as `Class<?>` (where `?` denotes a wildcard).

`java.lang.String`

`java.lang.String` is Java's basic string type. It is immutable. It does not implement `Iterable<Character>`, so it cannot be iterated over in a for-each loop, but can be converted to `char[]`. Some methods treat each UTF-16 code unit as a `char`, but methods to convert to an `int[]` that is effectively UTF-32 are also available. `String` implements `CharSequence`, so `char`s in the `String` can be accessed by the method `charAt()`.

`java.lang.Throwable`

`java.lang.Throwable` is the supertype of everything that can be thrown or caught with Java's `throw` and `catch` statements. Its direct known subclasses are `java.lang.Error` (for serious unrecoverable errors) and `java.lang.Exception` (for exceptions that may naturally occur in the execution of a program).

`java.lang.Error`

`java.lang.Error` is the supertype of all error classes and extends `Throwable`. It is used to indicate conditions that a reasonable application should not catch.

`java.lang.Exception`

`java.lang.Exception` is the supertype of all exception classes and extends `Throwable`. It is used to indicate conditions that a reasonable application may have reason to catch.

`java.lang.Math`

`java.lang.Math` is a utility class containing mathematical functions and mathematical constants (such as `Math.sin()`, `Math.pow()`, and `Math.PI`).

`java.lang.IO`

`java.lang.IO` is a class introduced in Java 25 (previously residing in `java.io` as `java.io.IO`). It allows simpler access to the standard input and output streams over `System.in` and `System.out`.

Primitives

Each primitive type has an associated wrapper class (see primitive types).

Program structure

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

`main` method

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). The `main` method along with the main class must be declared `public`. The method must be `static` and is passed command-line arguments as an array of strings. Unlike C++ or C#, it never returns a value and must return `void`. However, a return code can be specified to the operating system by calling `System.exit()`.

```
public static void main(String[] args) {
    // ...
}
```

Packages

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 essentially a namespace. Packages do not have hierarchies, even though the periods may suggest so. A package can be controlled whether it is accessible externally or internally of a project using modules.

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

```
package com.myapp.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 `com.myapp.mylibrary.MyClass` will have the following path: `com/myapp/mylibrary/MyClass.java`.

Modules

Modules are used to group packages and tightly control what packages belong to the public API. Contrary to Jar files, modules explicitly declare which modules they depend on, and what packages they export.[1] Explicit dependency declarations improve the integrity of the code, by making it easier to reason about

large applications and the dependencies between software components.

The module declaration is placed in a file named `module-info.java` at the root of the module's source-file hierarchy. The JDK will verify dependencies and interactions between modules both at compile-time and runtime.

For example, the following module declaration declares that the module `com.foo.bar` depends on another `com.foo.baz` module, and exports the following packages: `com.foo.bar.alpha` and `com.foo.bar.beta`:

```
module com.foo.bar {
    requires com.foo.baz;

    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
```

The public members of `com.foo.bar.alpha` and `com.foo.bar.beta` packages will be accessible by dependent modules. Private members are inaccessible even through a means such as reflection. Note that in Java versions 9 through 16, whether such 'illegal access' is *de facto* permitted depends on a command line setting.[2]

The JDK itself has been modularized in Java 9.[3] For example, the majority of the Java standard library is exported by the module `java.base`.

Import declaration

An import statement is used to resolve a type belonging to another package (namespace). It can be seen as similar to `using` in C++.

Type import declaration

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 com.myapp;

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 have been incorrect without the import.
         */
        Random random = new Random();
    }
}
```

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

```java
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 will not import all available classes. */
```

Static import declaration

This type of declaration has been available since J2SE 5.0. Static import declarations allow access to static members defined in another class, interface, annotation, or enum; without specifying the class name:

```java
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:

```java
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`:

```java
public enum ColorName {
    RED, BLUE, GREEN
};
```

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

```java
import org.example.screen.ColorName;
import static org.example.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;
        }
    }
}
```

Module import declaration

Since Java 25, modules can be used in import statements to automatically import all the packages exported by the module.[4] This is done using `import module`. For example, `import module java.sql;` is equivalent to

```
import java.sql.*;
import javax.sql.*;
// Remaining indirect exports from java.logging, java.transaction.xa, and java.xml
```

Similarly, `import module java.base;`, similarly, imports all 54 packages belonging to `java.base`.

```
import module java.base;

/**
 * importing module java.base allows us to avoid manually importing most classes
 * the following classes (outside of java.lang) are used:
 * java.text.MessageFormat
 * java.util.Date
 * java.util.List
 * java.util.concurrent.ThreadLocalRandom
 */
public class Example {
    public static void main(String[] args) {
        List<String> colours = List.of("Red", "Orange", "Yellow", "Green", "Blue", "Indigo",
"Violet");
        IO.println(MessageFormat.format("My favourite colour is {0} and today is {1,date,long}",
            colours.get(ThreadLocalRandom.current().nextInt(colours.size())),
            new Date()
        ));
    }
}
```

Operators

Operators in Java are similar to those in C++. However, there is no `delete` operator due to garbage collection mechanisms in Java, and there are no operations on pointers 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. The only overloaded operator is `operator+` for string concatenation.

Precedence	Operator	Description	Associativity
1	`()`	Method invocation	Left-to-right
	`[]`	Array access	
	`.`	Class member selection	
2	`++ --`	Postfix increment and decrement[5]	
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 left shift, signed right shift and unsigned right shift	
7	`< <=`	Relational "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 conditional (see ?:)	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

Conditional statements

`if` statement

if statements 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 == 3) {
    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, a ?: 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

Switch statements in Java can use `byte`, `short`, `char`, and `int` (not `long`) primitive data types or their corresponding wrapper types. Starting with J2SE 5.0, it is possible to use enum types. Starting with Java SE 7, it is possible to use Strings.[6] 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

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

```
enum Result {
    GREAT,
    FINE,
    // more enum values
}

Result result = switch (ch) {
    case 'A' -> Result.GREAT;
    case 'B', 'C' -> Result.FINE;
    default -> throw new Exception();
};
```

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.

```
Result result = switch (ch) {
    case 'A':
        yield Result.GREAT;
    case 'B':
    case 'C':
        yield Result.FINE;
    default:
        throw new Exception();
};
```

Iteration statements

Iteration statements are statements that are repeatedly executed when a given condition is evaluated as true. Since J2SE 5.0, Java has four forms of such statements. The condition must have type `boolean` or `java.lang.Boolean`. Java does not implicitly convert integers or class types to Boolean values.

For example, the following code is valid in C but results in a compilation error in Java.

```
while (1) {
    doSomething();
}
```

`while` loop

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

```
while (i < 10) {
    doSomething();
}
```

`do ... while` loop

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

`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 never terminates:

```
for (;;) {
    doSomething();
}
```

Foreach loop

Foreach loops have been available since 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. This for loop from Java was later added to C++11. Unlike C#, 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

Labels

Labels are given points in code used by `break` and `continue` statements. While `goto` is a reserved keyword in Java, it cannot be used to jump to specific points in code (in fact it has no use at all).

```
start:
someMethod();
```

`break` statement

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

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

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

`try-catch-finally` statements

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 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 cleanup 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

`try`-with-resources statements are a special type of `try-catch-finally` statements introduced as an implementation of the 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

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

Java has built-in tools for multi-thread programming. For the purposes of thread synchronization 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 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

`assert` statements have been available since J2SE 1.4. These types of statements are used to make assertions 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

Primitive types in Java include integer types, floating-point numbers, UTF-16 code units and a Boolean type. Unlike C++ and C#, 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++, methods such as `.toUnsignedInt()`. Nevertheless, criticisms have been leveled about the lack of compatibility with C and C++ this causes.[7]

Primitive types					
Type name	**Class Library equivalent**	**Value**	**Range**	**Size**	**Default value**
`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`
`byte`	`java.lang.Byte`	integer	-128 through 127	8-bit (1-byte)	`0`
`float`	`java.lang.Float`	floating point number	±1.401298E−45 through ±3.402823E+38	32-bit (4-byte)	`0.0`
`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`	8-bit (1-byte)	`false`
`char`	`java.lang.Character`	single Unicode character	`'\u0000'` through `'\uFFFF'`	16-bit (2-byte)	`'\u0000'`
`void`	`java.lang.Void`	N/A	N/A	N/A	N/A

`null` has no type, neither primitive nor class. Any object type may store `null`.

The class `Void` is used to hold a reference to the `Class` object representing the keyword `void`. It cannot be instantiated as `void` cannot be the type of any object. For example, `CompletableFuture<Void>` signifies that a `CompletableFuture` performs a task that does not return a value.

The class `java.lang.Number` represents all numeric values which can be converted to `byte`, `double`, `float`, `int`, `long`, and `short`.

`char` does not necessarily correspond to a single character. It may represent a part of a surrogate pair, in which case Unicode code point is represented by a sequence of two `char` values.

Boxing and unboxing

This language feature was introduced in 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

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

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

```
// Long syntax
int[] numbers = new int[] {20, 1, 42, 15, 34};
// Short syntax
int[] numbers2 = {20, 1, 42, 15, 34};
```

Multi-dimensional arrays

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

Classes 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

Top-level class	`class Foo {`
	`  // Class members`
	`}`

Inner class	`// Top-level class`
	`class Foo {`
	`  // Inner class`
	`  class Bar {`
	`  // ...`
	`  }`
	`}`

Nested class	`// Top-level class`
	`class Foo {`
	`  // Nested class`
	`  static class Bar {`
	`  // ...`
	`  }`
	`}`

Local class	`class Foo {`
	`  void bar() {`
	`  // Local class within a method`
	`  class Foobar {`
	`  // ...`
	`  }`
	`  }`
	`}`

Anonymous class	`class Foo {`
	`  void bar() {`
	`  // Creation of a new anonymous class extending Object`
	`  new Object() {};`
	`  }`
	`}`

Instantiation

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

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

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

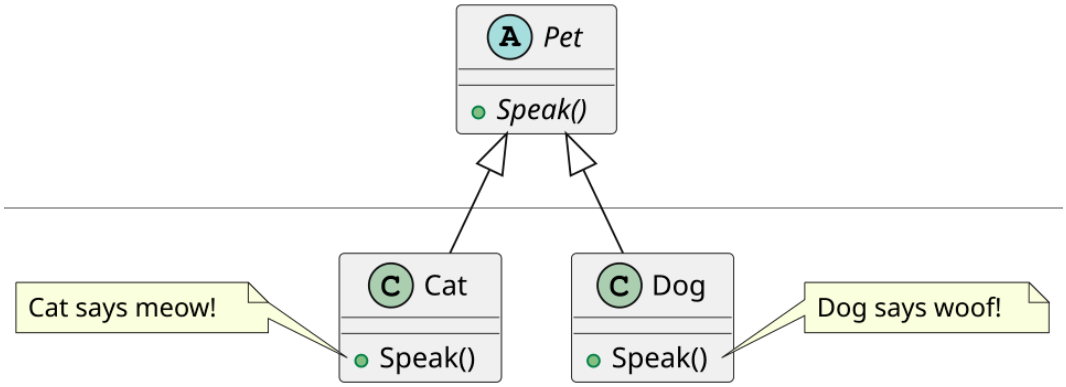
- **`abstract`** - Specifies that a class only serves as a base class and cannot be instantiated.
- **`static`** - Used only for member classes, specifies that the member class does not belong to a specific instance of the containing class.
- **`final`** - Classes marked as `final` cannot be extended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

Abstract class

By default, all methods in all classes are concrete, unless the abstract keyword is used. An abstract class may include abstract methods, which have no implementation. By default, all methods in all interfaces are abstract, unless the default keyword is used. The default keyword can be used to specify a concrete method in an interface.

```
//By default, all methods in all classes are concrete, unless the abstract keyword is used.
public abstract class Demo {
    // An abstract class may include abstract methods, which have no implementation.
    public abstract int sum(int x, int y);

    // An abstract class may also include concrete methods.
    public int product(int x, int y) {
        return x * y;
    }
}

//By default, all methods in all interfaces are abstract, unless the default keyword is used.
```

```java
interface DemoInterface {
    int getLength(); //The abstract keyword can be used here, though is completely useless

    //The default keyword can be used in this context to specify a concrete method in an interface
    default int product(int x, int y) {
        return x * y;
    }
}
```



Final class

A *final class* cannot be subclassed. As doing this can confer security and efficiency benefits, many of the Java standard library classes are final, such as `java.lang.System (https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/System.html)` and `java.lang.String (https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/String.html)`.

Example:

```java
public final class FinalClass {
    // ...
}

// Forbidden
public class DerivedClass extends FinalClass {
    // ...
}
```

Access modifiers

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.

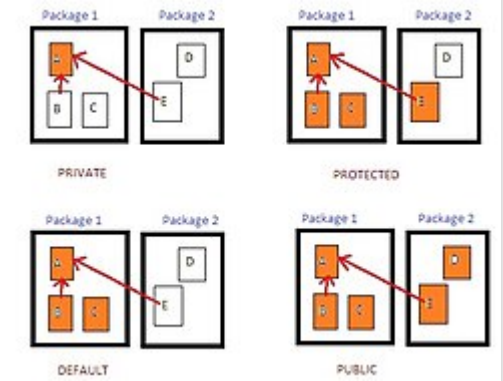
```java
public class Foo {
    int baz() {
        return 0;
    }

    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

Constructors and initializers

A constructor 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:



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

```
class Foo {
    String str;

    // Constructor with no arguments
    Foo() {}

    // Constructor with one argument
    Foo(String str) {
        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 (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. However, every object has a `finalize()` method called prior to garbage collection, which can be overridden to implement finalization.

Methods

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.

```
import java.io.IOException;
import java.util.zip.DataFormatException;

// Indicates that IOException and DataFormatException may be thrown
void operateOnFile(File f) throws IOException, DataFormatException {
    // ...
}
```

Modifiers

- **`abstract`** - Abstract methods can be present only in abstract classes, 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 in platform-dependent code. Actual implementation happens outside Java code, and such methods have no body.
- **`strictfp`** - Declares strict conformance to IEEE 754 in carrying out floating-point operations. Now obsolete.
- **`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.

Final methods

A final method cannot be overridden or hidden by subclasses.[8] This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.[9]

Example:

```
public class Base {
    public void m1() { ... }
    public final void m2() { ... }

    public static void m3() { ... }
    public static final void m4() { ... }
}

public class Derived extends Base {
    public void m1() { ... } // OK, overriding Base#m1()
    public void m2() { ... } // forbidden

    public static void m3() { ... } // OK, hiding Base#m3()
    public static void m4() { ... } // forbidden
}
```

A common misconception is that declaring a method as `final` improves efficiency by allowing the compiler to directly insert the method wherever it is called (see inline expansion). Because the method is loaded at runtime, compilers are unable to do this. Only the runtime environment and JIT compiler know exactly which classes have been loaded, and so only they are able to make decisions about when to inline, whether or not the method is final.[10]

Machine code compilers that generate directly executable, platform-specific machine code, are an exception. When using static linking, the compiler can safely assume that methods and variables computable at compile-time may be inlined.

Varargs

This language feature was introduced in 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 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.

```java
// numbers represents varargs
void printReport(String header, int... numbers) {
    System.out.println(header);
    for (int num : numbers) {
        System.out.println(num);
    }
}

// Calling varargs method
printReport("Report data", 74, 83, 25, 96);
```

Fields

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

```java
class Foo {
    double bar;
}
```

Fields can be initialized directly when declared.

```java
class Foo {
    double bar = 2.3;
}
```

Modifiers

- **`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.
- **`volatile`** - If a field is declared `volatile`, it is ensured that all threads see a consistent value for the variable.

Inheritance

Classes in Java can only inherit 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.

```java
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:

```java
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

Unlike C++, all non-`final` methods in Java are virtual and can be overridden by the inheriting classes.

```java
class Operation {
    public int doSomething() {
        return 0;
    }
}

class NewOperation extends Operation {
    @Override
    public int doSomething() {
        return 1;
    }
}
```

Abstract classes

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 is to be considered incomplete, so cannot be instantiated.

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.

```java
package org.example.test;

public class AbstractClass {
    private static final String hello;

    static {
        System.out.printf("%s: static block runtime%n", Abstract.class.getName());
        hello = String.format("hello from %s", AbstractClass.class.getName());
    }

    {
```

```
        System.out.printf("%s: instance block runtime%n", Abstract.class.getName());
    }

    public AbstractClass() {
        System.out.printf("%s: constructor runtime%n", Abstract.class.getName());
    }

    public static void hello() {
        System.out.println(hello);
    }
}
```

```
package org.example.test;

public class CustomClass extends AbstractClass {

    static {
        System.out.printf("%s: static block runtime%n", CustomClass.class.getName());
    }

    {
        System.out.printf("%s: instance block runtime%n", CustomClass.class.getName());
    }

    public CustomClass() {
        System.out.printf("%s: constructor runtime%n", CustomClass.class.getName());
    }

    public static void main(String[] args) {
        CustomClass nc = new CustomClass();
        hello();
        AbstractClass.hello(); //also valid
    }
}
```

Output:

```
org.example.test.AbstractClass: static block runtime
org.example.test.CustomClass: static block runtime
org.example.test.AbstractClass: instance block runtime
org.example.test.AbstractClass: constructor runtime
org.example.test.CustomClass: instance block runtime
org.example.test.CustomClass: constructor runtime
hello from org.example.test.AbstractClass
```

Enumerations

This language feature was introduced in 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

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

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. A 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
// ActionHandler can be represented as RequestListener...
RequestListener listener = new ActionHandler();
// ...and thus is known to implement requestReceived() method
listener.requestReceived();
```

Functional interfaces and lambda expressions

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. 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 do not 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.[11]

```
// 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

Java allows method references using the operator `::` (it is not related to the C++ namespace qualifying operator `::`). 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

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

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 does not 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 String.format("%s Extended", input);
    }
}
```

Static methods

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

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

Annotations in Java are a way to embed metadata into code. This language feature was introduced in J2SE 5.0.

Annotation types

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

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(
    fileSystemOperations, // mandatory
    networkOperations = true // optional
)
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

Generics, or parameterized types, or parametric polymorphism, is one of the major features introduced in 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.[12]

Generic classes

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

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

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

- Java Platform, Standard Edition
- C# syntax
- C++ syntax
- C syntax
- JavaScript syntax

References

1. Mark Reinhold (March 8, 2016). "The State of the Module System" (http://openjdk.java.net/projects/jigsaw/spec/sotms/). Oracle Corporation. Retrieved February 18, 2017.
2. "JEP 396: Strongly Encapsulate JDK Internals by Default" (https://openjdk.java.net/jeps/396). Retrieved February 6, 2021.
3. "JDK Module Summary" (https://web.archive.org/web/20151208074800/http://cr.openjdk.java.net/~mr/jigsaw/ea/module-summary.html). Oracle Corporation. June 24, 2016. Archived from the original (http://cr.openjdk.java.net/~mr/jigsaw/ea/module-summary.html) on December 8, 2015. Retrieved February 18, 2017.
4. "JEP 494: Module Import Declarations (Second Preview)" (https://openjdk.org/jeps/494). *openjdk.org*.
5. "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. Archived (https://web.archive.org/web/20150624161036/http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html) from the original on June 24, 2015. Retrieved June 16, 2015.
6. "The switch Statement (The Java™ Tutorials > Learning the Java Language > Language Basics)" (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html). *docs.oracle.com*. Archived (https://web.archive.org/web/20100315060844/http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html) from the original on March 15, 2010. Retrieved August 15, 2021.
7. Owens, Sean. "Java and unsigned int, unsigned short, unsigned byte, unsigned long, etc. (Or rather, the lack thereof)" (http://darksleep.com/player/JavaAndUnsignedTypes.html). Archived (https://web.archive.org/web/20090220171410/http://darksleep.com/player/JavaAndUnsignedTypes.html) from the original on February 20, 2009. Retrieved April 21, 2010.
8. "Chapter 8. Classes" (https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html). *docs.oracle.com*. Retrieved April 25, 2024.
9. "Writing Final Classes and Methods" (https://docs.oracle.com/javase/tutorial/java/IandI/final.html). *docs.oracle.com*. Retrieved April 25, 2024.
10. "Java theory and practice: Is that your final answer?" (https://web.archive.org/web/20090208100217/http://www.ibm.com/developerworks/java/library/j-jtp1029.html). *developer.ibm.com*. Archived from the original (http://www.ibm.com/developerworks/java/library/j-jtp1029.html) on February 8, 2009. Retrieved April 25, 2024.
11. "Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects)" (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html). *docs.oracle.com*. Archived (https://web.archive.org/web/20200616055353/https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) from the original on June 16, 2020. Retrieved August 8, 2021.
12. "Generics in the Run Time (C# Programming Guide)" (https://msdn.microsoft.com/en-us/library/f4a6ta2h.aspx). Archived (https://web.archive.org/web/20160310012449/https://msdn.microsoft.com/en-us/library/f4a6ta2h.aspx) from the original on March 10, 2016. Retrieved March 9, 2016.

- Naughton, Patrick; Schildt, Herbert (1999). *Java 2: The Complete Reference* (3rd ed.). The McGraw-Hill Companies. ISBN 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 0-521-77768-2.
- Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad (2005). *Java Language Specification* (http://java.sun.com/docs/books/jls/) (3rd ed.). Addison-Wesley Professional. Archived (https://web.archive.org/web/20090226152425/http://java.sun.com/docs/books/jls/) from the original on February 26, 2009. Retrieved December 3, 2008.

External links

- The Java Language Specification, Third edition (https://docs.oracle.com/javase/specs/) Authoritative description of the Java language
- Java SE 24 API Javadocs (https://docs.oracle.com/en/java/javase/24/docs/api/)