# The Java® Language Specification

Java SE 8 Edition

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley

ORACLE

# The Java® Language Specification

*Java SE 8 Edition*

*This page intentionally left blank*

# The Java® Language Specification

*Java SE 8 Edition*

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2014936248
ISBN-13: 978-0-13-390069-9
ISBN-10: 0-13-390069-X



Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan. First printing, May 2014.

To Maurizio, with deepest thanks.

*This page intentionally left blank*

# Table of Contents

## 12 Execution 353



## 13 Binary Compatibility 375

*This page intentionally left blank*

# Preface to the Java SE 8 Edition

IN 1996, James Gosling, Bill Joy, and Guy Steele wrote for the First Edition of *The Java® Language Specification*:

*"We believe that the Java programming language is a mature language, ready for widespread use. Nevertheless, we expect some evolution of the language in the years to come. We intend to manage this evolution in a way that is completely compatible with existing applications."*

Java SE 8 represents the single largest evolution of the Java language in its history. A relatively small number of features - lambda expressions, method references, and functional interfaces - combine to offer a programming model that fuses the object-oriented and functional styles. Under the leadership of Brian Goetz, this fusion has been accomplished in a way that encourages best practices - immutability, statelessness, compositionality - while preserving "the feel of Java" - readability, simplicity, universality.

Crucially, the libraries of the Java SE platform have co-evolved with the Java language. This means that using lambda expressions and method references to represent behavior - for example, an operation to be applied to each element in a list - is productive and performant "out of the box". In a similar fashion, the Java Virtual Machine has co-evolved with the Java language to ensure that default methods support library evolution as consistently as possible across compile time and run time, given the constraints of separate compilation.

Initiatives to add first-class functions to the Java language have been around since the 1990s. The *BGGA* and *CICE* proposals circa 2007 brought new energy to the topic, while the creation of *Project Lambda* in OpenJDK circa 2009 attracted unprecedented levels of interest. The addition of method handles to the JVM in Java SE 7 opened the door to new implementation techniques while retaining "write once, run anywhere." In time, language changes were overseen by JSR 335, *Lambda Expressions for the Java Programming Language*, whose Expert Group consisted of Joshua Bloch, Kevin Bourrillion, Andrey Breslav, Rémi Forax, Dan Heidinga, Doug Lea, Bob Lee, David Lloyd, Sam Pullara, Srikanth Sankaran, and Vladimir Zakharov.

Programming language design typically involves grappling with degrees of complexity utterly hidden from the language's users. (For this reason, it is often compared to an iceberg: 90% of it is invisible.) In JSR 335, the greatest complexity

lurked in the interaction of implicitly typed lambda expressions with overload resolution. In this and many other areas, Dan Smith at Oracle did an outstanding job of thoroughly specifying the desired behavior. His words are to be found throughout this specification, including an entirely new chapter on type inference.

Another initiative in Java SE 8 has been to enhance the utility of annotations, one of the most popular features of the Java language. First, the Java grammar has been extended to allow annotations on types in many language constructs, forming the basis for novel static analysis tools such as the *Checker Framework*. This feature was specified by JSR 308, *Annotations on Java Types*, led by Michael Ernst with an Expert Group of myself, Doug Lea, and Srikanth Sankaran. The changes involved in this specification were wide-ranging, and the unstinting efforts of Michael Ernst and Werner Dietl over many years are warmly recognized. Second, annotations may be "repeated" on a language construct, to the great benefit of APIs that model domain-specific configuration with annotation types. Michael Keith and Bill Shannon in Java EE initiated and guided this feature.

Many colleagues in the Java Platform Group at Oracle have provided valuable support to this specification: Leonid Arbouzov, Mandy Chung, Joe Darcy, Robert Field, Joel Franck, Sonali Goel, Jon Gibbons, Jeannette Hung, Stuart Marks, Eric McCorkle, Matherey Nunez, Mark Reinhold, Vicente Romero, John Rose, Georges Saab, Steve Sides, Bernard Traversat, and Michel Trudeau.

Perhaps the greatest acknowledgement must go to the compiler engineers who turn the specification into real software. Maurizio Cimadamore at Oracle worked heroically from the earliest days on the design of lambda expressions and their implementation in `javac`. Support for Java SE 8 features in Eclipse was contributed by Jayaprakash Arthanareeswaran, Shankha Banerjee, Anirban Chakraborty, Andrew Clement, Stephan Herrmann, Markus Keller, Jesper Møller, Manoj Palat, Srikanth Sankaran, and Olivier Thomann; and in IntelliJ by Anna Kozlova, Alexey Kudravtsev, and Roman Shevchenko. They deserve the thanks of the entire Java community.

Java SE 8 is a renaissance for the Java language. While some search for the "next great language", we believe that programming in Java is more exciting and productive than ever. We hope that it continues to wear well for you.

Alex Buckley
*Santa Clara, California*

*March, 2014*

CHAPTER 1

# Introduction

THE Java® programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features.

The Java programming language is strongly and statically typed. This specification clearly distinguishes between the *compile-time errors* that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

The Java programming language is a relatively high-level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's `free` or C++'s `delete`). High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way.

The Java programming language is normally compiled to the bytecode instruction set and binary format defined in *The Java Virtual Machine Specification, Java SE 8 Edition*.

## 1.1 Organization of the Specification

Chapter 2 describes grammars and the notation used to present the lexical and syntactic grammars for the language.

Chapter 3 describes the lexical structure of the Java programming language, which is based on C and C++. The language is written in the Unicode character set. It supports the writing of Unicode characters on systems that support only ASCII.

Chapter 4 describes types, values, and variables. Types are subdivided into primitive types and reference types.

The primitive types are defined to be the same on all machines and in all implementations, and are various sizes of two's-complement integers, single- and double-precision IEEE 754 standard floating-point numbers, a `boolean` type, and a Unicode character `char` type. Values of the primitive types do not share state.

Reference types are the class types, the interface types, and the array types. The reference types are implemented by dynamically created objects that are either instances of classes or arrays. Many references to each object can exist. All objects (including arrays) support the methods of the class `Object`, which is the (single) root of the class hierarchy. A predefined `String` class supports Unicode character strings. Classes exist for wrapping primitive values inside of objects. In many cases, wrapping and unwrapping is performed automatically by the compiler (in which case, wrapping is called boxing, and unwrapping is called unboxing). Class and interface declarations may be generic, that is, they may be parameterized by other reference types. Such declarations may then be invoked with specific type arguments.

Variables are typed storage locations. A variable of a primitive type holds a value of that exact primitive type. A variable of a class type can hold a null reference or a reference to an object whose type is that class type or any subclass of that class type. A variable of an interface type can hold a null reference or a reference to an instance of any class that implements the interface. A variable of an array type can hold a null reference or a reference to an array. A variable of class type `Object` can hold a null reference or a reference to any object, whether class instance or array.

Chapter 5 describes conversions and numeric promotions. Conversions change the compile-time type and, sometimes, the value of an expression. These conversions include the boxing and unboxing conversions between primitive types and reference types. Numeric promotions are used to convert the operands of a numeric operator to a common type where an operation can be performed. There are no

loopholes in the language; casts on reference types are checked at run time to ensure type safety.

Chapter 6 describes declarations and names, and how to determine what names mean (denote). The language does not require types or their members to be declared before they are used. Declaration order is significant only for local variables, local classes, and the order of initializers of fields in a class or interface.

The Java programming language provides control over the scope of names and supports limitations on external access to members of packages, classes, and interfaces. This helps in writing large programs by distinguishing the implementation of a type from its users and those who extend it. Recommended naming conventions that make for more readable programs are described here.

Chapter 7 describes the structure of a program, which is organized into packages similar to the modules of Modula. The members of a package are classes, interfaces, and subpackages. Packages are divided into compilation units. Compilation units contain type declarations and can import types from other packages to give them short names. Packages have names in a hierarchical name space, and the Internet domain name system can usually be used to form unique package names.

Chapter 8 describes classes. The members of classes are classes, interfaces, fields (variables) and methods. Class variables exist once per class. Class methods operate without reference to a specific object. Instance variables are dynamically created in objects that are instances of classes. Instance methods are invoked on instances of classes; such instances become the current object `this` during their execution, supporting the object-oriented programming style.

Classes support single implementation inheritance, in which the implementation of each class is derived from that of a single superclass, and ultimately from the class `Object`. Variables of a class type can reference an instance of that class or of any subclass of that class, allowing new types to be used with existing methods, polymorphically.

Classes support concurrent programming with `synchronized` methods. Methods declare the checked exceptions that can arise from their execution, which allows compile-time checking to ensure that exceptional conditions are handled. Objects can declare a `finalize` method that will be invoked before the objects are discarded by the garbage collector, allowing the objects to clean up their state.

For simplicity, the language has neither declaration "headers" separate from the implementation of a class nor separate type and class hierarchies.

A special form of classes, enums, support the definition of small sets of values and their manipulation in a type safe manner. Unlike enumerations in other languages, enums are objects and may have their own methods.

Chapter 9 describes interface types, which declare a set of abstract methods, member types, and constants. Classes that are otherwise unrelated can implement the same interface type. A variable of an interface type can contain a reference to any object that implements the interface. Multiple interface inheritance is supported.

Annotation types are specialized interfaces used to annotate declarations. Such annotations are not permitted to affect the semantics of programs in the Java programming language in any way. However, they provide useful input to various tools.

Chapter 10 describes arrays. Array accesses include bounds checking. Arrays are dynamically created objects and may be assigned to variables of type `Object`. The language supports arrays of arrays, rather than multidimensional arrays.

Chapter 11 describes exceptions, which are nonresuming and fully integrated with the language semantics and concurrency mechanisms. There are three kinds of exceptions: checked exceptions, run-time exceptions, and errors. The compiler ensures that checked exceptions are properly handled by requiring that a method or constructor can result in a checked exception only if the method or constructor declares it. This provides compile-time checking that exception handlers exist, and aids programming in the large. Most user-defined exceptions should be checked exceptions. Invalid operations in the program detected by the Java Virtual Machine result in run-time exceptions, such as `NullPointerException`. Errors result from failures detected by the Java Virtual Machine, such as `OutOfMemoryError`. Most simple programs do not try to handle errors.

Chapter 12 describes activities that occur during execution of a program. A program is normally stored as binary files representing compiled classes and interfaces. These binary files can be loaded into a Java Virtual Machine, linked to other classes and interfaces, and initialized.

After initialization, class methods and class variables may be used. Some classes may be instantiated to create new objects of the class type. Objects that are class instances also contain an instance of each superclass of the class, and object creation involves recursive creation of these superclass instances.

When an object is no longer referenced, it may be reclaimed by the garbage collector. If an object declares a finalizer, the finalizer is executed before the object

is reclaimed to give the object a last chance to clean up resources that would not otherwise be released. When a class is no longer needed, it may be unloaded.

Chapter 13 describes binary compatibility, specifying the impact of changes to types on other types that use the changed types but have not been recompiled. These considerations are of interest to developers of types that are to be widely distributed, in a continuing series of versions, often through the Internet. Good program development environments automatically recompile dependent code whenever a type is changed, so most programmers need not be concerned about these details.

Chapter 14 describes blocks and statements, which are based on C and C++. The language has no `goto` statement, but includes labeled `break` and `continue` statements. Unlike C, the Java programming language requires `boolean` (or `Boolean`) expressions in control-flow statements, and does not convert types to `boolean` implicitly (except through unboxing), in the hope of catching more errors at compile time. A `synchronized` statement provides basic object-level monitor locking. A `try` statement can include `catch` and `finally` clauses to protect against non-local control transfers.

Chapter 15 describes expressions. This document fully specifies the (apparent) order of evaluation of expressions, for increased determinism and portability. Overloaded methods and constructors are resolved at compile time by picking the most specific method or constructor from those which are applicable.

Chapter 16 describes the precise way in which the language ensures that local variables are definitely set before use. While all other variables are automatically initialized to a default value, the Java programming language does not automatically initialize local variables in order to avoid masking programming errors.

Chapter 17 describes the semantics of threads and locks, which are based on the monitor-based concurrency originally introduced with the Mesa programming language. The Java programming language specifies a memory model for shared-memory multiprocessors that supports high-performance implementations.

Chapter 18 describes a variety of type inference algorithms used to test applicability of generic methods and to infer types in a generic method invocation.

Chapter 19 presents a syntactic grammar for the language.

## 1.2 Example Programs

Most of the example programs given in the text are ready to be executed and are similar in form to:

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
        System.out.println();
    }
}
```

On a machine with the Oracle JDK installed, this class, stored in the file `Test.java`, can be compiled and executed by giving the commands:

```
javac Test.java
java Test Hello, world.
```

producing the output:

```
Hello, world.
```

## 1.3 Notation

Throughout this specification we refer to classes and interfaces drawn from the Java SE platform API. Whenever we refer to a class or interface (other than those declared in an example) using a single identifier *N*, the intended reference is to the class or interface named *N* in the package `java.lang`. We use the canonical name (§6.7) for classes or interfaces from packages other than `java.lang`.

Non-normative information, designed to clarify the specification, is given in smaller, indented text.

> This is non-normative information. It provides intuition, rationale, advice, examples, etc.

The type system of the Java programming language occasionally relies on the notion of a *substitution*. The notation $[F_1:=T_1, \ldots, F_n:=T_n]$ denotes substitution of $F_i$ by $T_i$ for $1 \leq i \leq n$.

## 1.4 Relationship to Predefined Classes and Interfaces

As noted above, this specification often refers to classes of the Java SE platform API. In particular, some classes have a special relationship with the Java programming language. Examples include classes such as `Object`, `Class`, `ClassLoader`, `String`, `Thread`, and the classes and interfaces in package `java.lang.reflect`, among others. This specification constrains the behavior of such classes and interfaces, but does not provide a complete specification for them. The reader is referred to the Java SE platform API documentation.

Consequently, this specification does not describe reflection in any detail. Many linguistic constructs have analogs in the Core Reflection API (`java.lang.reflect`) and the Language Model API (`javax.lang.model`), but these are generally not discussed here. For example, when we list the ways in which an object can be created, we generally do not include the ways in which the Core Reflection API can accomplish this. Readers should be aware of these additional mechanisms even though they are not mentioned in the text.

## 1.5 Feedback

Readers may send feedback about errors, omissions, and ambiguities in *The Java® Language Specification* to `jls-comments_ww@oracle.com`.

Questions concerning the behavior of `javac` (the reference compiler for the Java programming language), and in particular its conformance to this specification, may be sent to `compiler-dev@openjdk.java.net`.

## 1.6 References

Apple Computer. *Dylan Reference Manual*. Apple Computer Inc., Cupertino, California. September 29, 1995.

Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990, ISBN 1-55558-041-6, 770-864.

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990, reprinted with corrections October 1992, ISBN 0-201-51459-1.

Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989, ISBN 0-201-13688-0.

Harbison, Samuel. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-596396.

Hoare, C. A. R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, December 1973. Reprinted in SIGACT/SIGPLAN Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, October 1973.

*IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8.

Madsen, Ole Lehrmann, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993, ISBN 0-201-62430-3.

Mitchell, James G., William Maybury, and Richard Sweet. *The Mesa Programming Language, Version 5.0*. Xerox PARC, Palo Alto, California, CSL 79-3, April 1979.

Stroustrup, Bjarne. *The C++ Progamming Language*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

Unicode Consortium, The. *The Unicode Standard, Version 6.0.0*. Mountain View, CA, 2011, ISBN 978-1-936213-01-6.

# Index

## Symbols



## A

# B

# C

# D

## E

# F

# G

# H



# I

## J



## K



## L

# M

# N

# O

# P

# Q



# R

# S

# T

# U

# V

# W