"History of Java Programming Language"
Free Java Guide & Tutorials

Java is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages which are generally designed either to be compiled to native (machine) code, or to be interpreted from source code at runtime, Java is intended to be compiled to a bytecode, which is then run (generally using JIT compilation) by a Java Virtual Machine.
The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java is only distantly related to JavaScript, though they have similar names and share a C-like syntax.

History

Java was started as a project called "Oak" by James Gosling in June 1991. Gosling's goals were to implement a virtual machine and a language that had a familiar C-like notation but with greater uniformity and simplicity than C/C++. The first public implementation was Java 1.0 in 1995. It made the promise of "Write Once, Run Anywhere", with free runtimes on popular platforms. It was fairly secure and its security was configurable, allowing for network and file access to be limited. The major web browsers soon incorporated it into their standard configurations in a secure "applet" configuration. popular quickly. New versions for large and small platforms (J2EE and J2ME) soon were designed with the advent of "Java 2". Sun has not announced any plans for a "Java 3".

In 1997, Sun approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a proprietary de facto standard that is controlled through the Java Community Process. Sun makes most of its Java implementations available without charge, with revenue being generated by specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) which is a subset of the SDK, the primary distinction being that in the JRE the compiler is not present.

Philosophy

There were five primary goals in the creation of the Java language:
1. It should use the object-oriented programming methodology.
2. It should allow the same program to be executed on multiple operating systems.
3. It should contain built-in support for using computer networks.
4. It should be designed to execute code from remote sources securely.
5. It should be easy to use by selecting what was considered the good parts of other object-oriented languages.

To achieve the goals of networking support and remote code execution, Java programmers sometimes find it necessary to use extensions such as CORBA, Internet Communications Engine, or OSGi.

Object orientation

The first characteristic, object orientation ("OO"), refers to a method of programming and language design. Although there are many interpretations of OO, one primary distinguishing idea is to design software so that the various types of data it manipulates are combined together with their relevant operations. Thus, data and code are combined into entities called objects. An object can be thought of as a self-contained bundle of behavior (code) and state (data). The principle is to separate the things that change from the things that stay the same; often, a change to some data structure requires a corresponding change to the code that operates on that data, or vice versa. This separation into coherent objects provides a more stable foundation for a software system's design. The intent is to make large software projects easier to manage, thus improving quality and reducing the number of failed projects.

Another primary goal of OO programming is to develop more generic objects so that software can become more reusable between projects. A generic "customer" object, for example, should have roughly the same basic set of behaviors between different software projects, especially when these projects overlap on some fundamental level as they often do in large organizations. In this sense, software objects can hopefully be seen more as pluggable components, helping the software industry build projects largely from existing and well-tested pieces, thus leading to a massive reduction in development times. Software reusability has met with mixed practical results, with two main difficulties: the design of truly generic objects is poorly understood, and a methodology for broad communication of reuse opportunities is lacking. Some open source communities want to help ease the reuse problem, by providing authors with ways to disseminate information about generally reusable objects and object libraries.

Platform independence

The second characteristic, platform independence, means that programs written in the Java language must run similarly on diverse hardware. One should be able to write a program once and run it anywhere.

This is achieved by most Java compilers by compiling the Java language code "halfway" to bytecode (specifically Java bytecode)—simplified machine instructions specific to the Java platform. The code is then run on a virtual machine (VM), a program written in native code on the host hardware that interprets and executes generic Java bytecode. Further, standardized libraries are provided to allow access to features of the host machines (such as graphics, threading and networking) in unified ways. Note that, although there's an explicit compiling stage, at some point, the Java bytecode is interpreted or converted to native machine instructions by the JIT compiler.

There are also implementations of Java compilers that compile to native object code, such as GCJ, removing the intermediate bytecode stage, but the output of these compilers can only be run on a single architecture.

Sun's license for Java insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support the RMI and JNI interfaces and had added platform-specific features of their own. In response, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support Java applets without a third-party plug-in. However, Sun and others have made available Java run-time systems at no cost for those and other versions of Windows.

The first implementations of the language used an interpreted virtual machine to achieve portability. These implementations produced programs that ran more slowly than programs compiled to native executables, for instance written in C or C++, so the language suffered a reputation for poor performance. More recent JVM implementations produce programs that run significantly faster than before, using multiple techniques.

The first technique is to simply compile directly into native code like a more traditional compiler, skipping bytecodes entirely. This achieves good performance, but at the expense of portability. Another technique, known as just-in-time compilation (JIT), translates the Java bytecodes into native code at the time that the program is run which results in a program that executes faster than interpreted code but also incurs compilation overhead during execution. More sophisticated VMs use dynamic recompilation, in which the VM can analyze the behavior of the running program and selectively recompile and optimize critical parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes. JIT compilation and dynamic recompilation allow Java programs to take advantage of the speed of native code without losing portability.

Portability is a technically difficult goal to achieve, and Java's success at that goal has been mixed. Although it is indeed possible to write programs for the Java platform that behave consistently across many host platforms, the large number of available platforms with small errors or inconsistencies led some to parody Sun's "Write once, run anywhere" slogan as "Write once, debug everywhere".

Platform-independent Java is however very successful with server-side applications, such as Web services, servlets, and Enterprise JavaBeans, as well as with Embedded systems based on OSGi, using Embedded Java environments.

Automatic garbage collection

One idea behind Java's automatic memory management model is that programmers should be spared the burden of having to perform manual memory management. In

some languages the programmer allocates memory to create any object stored on the heap and is responsible for later manually deallocating that memory to delete any such objects. If a programmer forgets to deallocate memory or writes code that fails to do so in a timely fashion, a memory leak can occur: the program will consume a potentially arbitrarily large amount of memory. In addition, if a region of memory is deallocated twice, the program can become unstable and may crash. Finally, in non garbage collected environments, there is a certain degree of overhead and complexity of user-code to track and finalize allocations.

In Java, this potential problem is avoided by automatic garbage collection. The programmer determines when objects are created, and the Java runtime is responsible for managing the object's lifecycle. The program or other objects can reference an object by holding a reference to it (which, from a low-level point of view, is its address on the heap). When no references to an object remain, the Java garbage collector automatically deletes the unreachable object, freeing memory and preventing a memory leak. Memory leaks may still occur if a programmer's code holds a reference to an object that is no longer needed—in other words, they can still occur but at higher conceptual levels.

The use of garbage collection in a language can also affect programming paradigms. If, for example, the developer assumes that the cost of memory allocation/recollection is low, they may choose to more freely construct objects instead of pre-initializing, holding and reusing them. With the small cost of potential performance penalties (inner-loop construction of large/complex objects), this facilitates thread-isolation (no need to synchronize as different threads work on different object instances) and data-hiding. The use of transient immutable value-objects minimizes side-effect programming.

Comparing Java and C++, it is possible in C++ to implement similar functionality (for example, a memory management model for specific classes can be designed in C++ to improve speed and lower memory fragmentation considerably), with the possible cost of extra development time and some application complexity. In Java, garbage collection is built-in and virtually invisible to the developer. That is, developers may have no notion of when garbage collection will take place as it may not necessarily correlate with any actions being explicitly performed by the code they write. Depending on intended application, this can be beneficial or disadvantageous: the programmer is freed from performing low-level tasks, but at the same time loses the option of writing lower level code.

Syntax

The syntax of Java is largely derived from C++. However, unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built from the ground up to be virtually fully object-oriented: everything in Java is an object with the exceptions of atomic datatypes (ordinal and real numbers, boolean values, and characters) and everything in Java is written inside a class.

Applet

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java
import java.applet.Applet;
import java.awt.Graphics;

public class Hello extends Applet {
public void paint(Graphics gc) {
gc.drawString("Hello, world!", 65, 95);
}
}
```

This applet will simply draw the string "Hello, world!" in the rectangle within which the applet will run. This is a slightly better example of using Java's OO features in that the class explicitly extends the basic "Applet" class, that it overrides the "paint" method and that it uses import statements.

```
<!-- Hello.html -->
<html>
<head>
<title>Hello World Applet</title>
</head> <body>
<applet code="Hello" width="200" height="200">
</applet>
</body>
</html>
```

An applet is placed in an HTML document using the <applet> HTML element. The applet tag has three attributes set: code="Hello" specifies the name of the Applet class and width="200" height="200" sets the pixel width and height of the applet. (Applets may also be embedded in HTML using either the object or embed element, although support for these elements by Web browsers is inconsistent.

Servlet

Java servlets are server-side Java EE components that generate responses to requests from clients.

```
// Hello.java
import java.io.*;
import javax.servlet.*;
```

```
public class Hello extends GenericServlet {
public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException
{
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("Hello, world!");
pw.close();
}
}
```

The import statements direct the Java compiler to include all of the public classes and interfaces from the java.io and javax.servlet packages in the compilation. The Hello class extends the GenericServlet class; the GenericServlet class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.

The Hello class overrides the service(ServletRequest, ServletResponse) method defined by the Servlet interface to provide the code for the service request handler. The service() method is passed a ServletRequest object that contains the request from the client and a ServletResponse object used to create the response returned to the client. The service() method declares that it throws the exceptions ServletException and IOException if a problem prevents it from responding to the request.

The setContentType(String) method in the response object is called to set the MIME content type of the returned data to "text/html". The getWriter() method in the response returns a PrintWriter object that is used to write the data that is sent to the client. The println(String) method is called to write the "Hello, world!" string to the response and then the close() method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.

Swing application

Swing is the advanced graphical user interface library for the Java SE platform.

```
// Hello.java
import javax.swing.*;

public class Hello extends JFrame {
Hello() {
setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
add(new JLabel("Hello, world!"));
pack();
}

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

```
new Hello().setVisible(true);
}
}
```

The import statement directs the Java compiler to include all of the public classes and interfaces from the javax.swing package in the compilation. The Hello class extends the JFrame class; the JFrame class implements a window with a title bar with a close control.

The Hello() constructor initializes the frame by first calling the setDefaultCloseOperation(int) method inherited from JFrame to set the default operation when the close control on the title bar is selected to WindowConstants.DISPOSE_ON_CLOSE—this causes the JFrame to be disposed of when the frame is closed (as opposed to merely hidden), which allows the JVM to exit and the program to terminate. Next a new JLabel is created for the string "Hello, world!" and the add(Component) method inherited from the Container superclass is called to add the label to the frame. The pack() method inherited from the Window superclass is called to size the window and layout its contents.

The main() method is called by the JVM when the program starts. It instantiates a new Hello frame and causes it to be displayed by calling the setVisible(boolean) method inherited from the Component superclass with the boolean parameter true. Note that once the frame is displayed, exiting the main method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing top-level windows have been disposed.

Look and feel

The default look and feel of GUI applications written in Java using the Swing toolkit is very different from native applications. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK and Motif are supplied by Sun. Apple also provides an Aqua look and feel for Mac OS X. Though prior implementations of these look and feels have been considered lacking, Swing in Java SE 6 addresses this problem by using more native widget drawing routines of the underlying platforms. Alternatively, third party toolkits such as wx4j or SWT may be used for increased integration with the native windowing system.

Lack of OO purity and facilities

Java's primitive types are not objects. Primitive types hold their values in the stack rather than being references to values. This was a conscious decision by Java's designers for performance reasons. Because of this, Java is not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to write as if primitive types are their wrapper classes, and freely

interchange between them for improved flexibility. Java designers decided not to implement certain features present in other OO languages, including:

* multiple inheritance
* operator overloading
* class properties
* tuples

Java Runtime Environment

The Java Runtime Environment or JRE is the software required to run any application deployed on the Java Platform. End-users commonly use a JRE in software packages and Web browser plugins. Sun also distributes a superset of the JRE called the Java 2 SDK (more commonly known as the JDK), which includes development tools such as the Java compiler, Javadoc, and debugger.