Examining the JDK Software

The JDK software contains the components you need to perform the following tasks:

- Develop Java technology applications
- Deploy Java technology applications
- Execute Java technology applications:

Figure 1-1 provides an overview of the JDK software. It shows the components of the JDK and the activities with which they are associated.

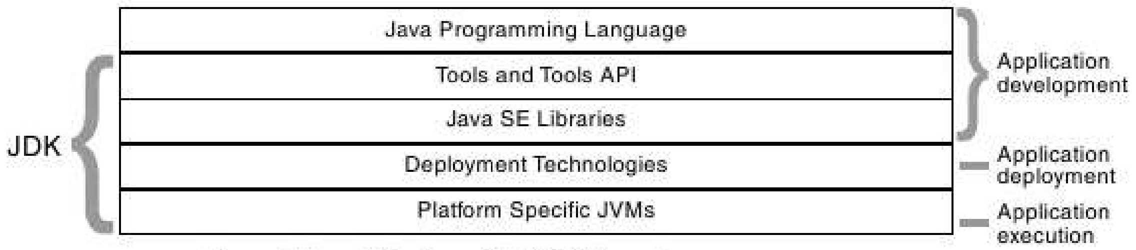


Figure 1-1 The Java SE JDK Overview

The JDK consists of the following components

- The Java programming language
- Tools and tools API
- Deployment technologies
- Java Platform, Standard Edition (Java SE) libraries
- Virtual Machine for the Java platform (Java Virtual Machine (JVM™))

Note – Strictly speaking, the Java programming language is not a component of the JDK software. However, it is included in the discussion of the JDK software on the basis that the JDK is primarily provided to support Java programming language-based application development.

To use the JDK, you need to download and install both the JDK software and the associated JDK documentation on your development platform. The following URL contains the instructions to download and install both the JDK software and the associated JDK documentation.

```
http://java.sun.com/javase/downloads/index.jsp
```

Examining the JDK Software Support for Developing Java Applications

The following JDK components are provided to support the development of Java technology applications:

- The Java programming language
- The JDK tools
- The JDK libraries

The Java Programming Language

The Java Programming Language is a general-purpose, concurrent, strongly typed, class-based object-oriented language. The Java programming language is defined by the Java language specification. The primary building block of a Java technology application is a *class*. The language specification specifies the language syntax for all Java language constructs including that of a class. Code 1-1 shows an example of a Java technology class.

Code 1-1 Example of a Java Technology Class

```
package trader;
import java.io.Serializable;
public class Stock implements Serializable {
    private String symbol;
    private float price;

    public Stock(String symbol, float price){
        this.symbol = symbol;
        this.price = price;
    }

    // Methods to return the private values of this object
    public String getSymbol() {
        return symbol;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float newPrice) {
        price = newPrice;
        }

        public String toString() {
            return "Stock:  " + symbol + "  " + price;
        }
    }
```

The JDK Tools and Tools API

Figure 1-2 highlights the tools and tool categories available with the Java SE JDK.

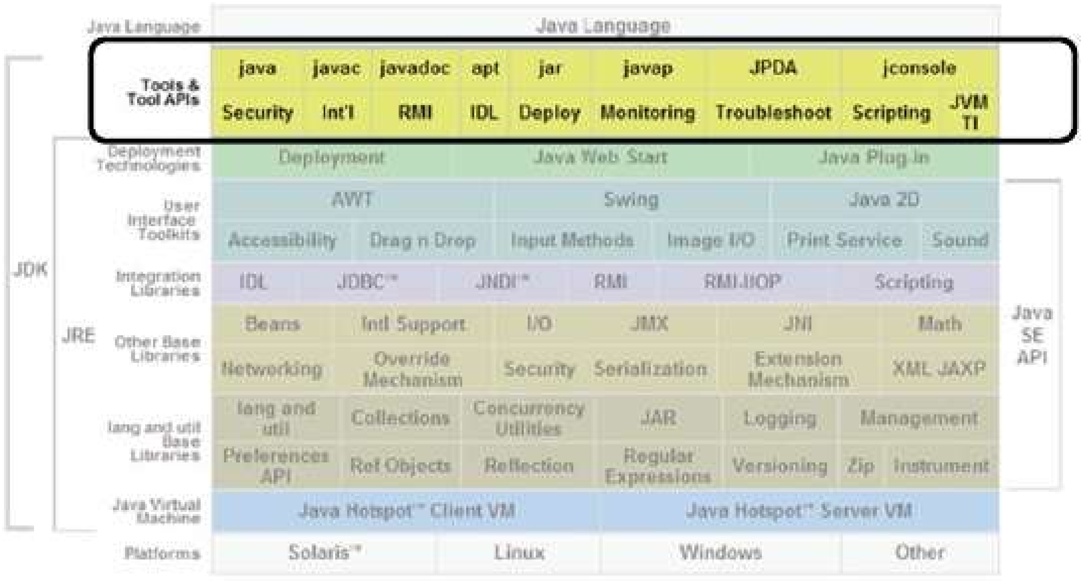


Figure 1-2 The Java SE JDK With the Java SE Tools Highlighted

The JDK tools provide support for the development and deployment of Java technology applications. The JDK tools can be classified into two categories.

- Basic tools

  These tools are the tools you use to create, build, and execute Java technology applications. Table 1-1 lists the tools from this group that you are could encounter in the early stages of learning to program using Java technology.

Table 1-1 Java SE Basic Tools: A Sample Selection

Tool Name	**Function**
`javac`	The compiler for the Java programming language
`java`	The launcher for Java technology applications
`jdb`	The Java debugger
`javadoc`	The API document generator
`jar`	Java Archive (JAR) file creator and management tool

- Advanced user tools

  The advanced user tools consist of several categories of tools. Each category helps you leverage a particular technology when creating a Java technology application. Table 1-2 lists the advanced tool categories.

Table 1-2 Advanced Tool Categories

Tool Category	**Comment**
Security tools	Help you create applications that work with security policies.
Internationalization tools	Help create applications that can be localized.
Remote method invocation (RMI) tools	Help create applications that work across a network.
Common object request broker architecture (CORBA) tools	Help create network applications that are based on CORBA technology.
Java deployment tools	Provide compression and decompression functionality that assist with the deployment of Java technology applications.

Table 1-2 Advanced Tool Categories

Tool Category	**Comment**
Java Plug-in tools	Utilities for use with the Java Plug-in.
Java web start tools	Utilities for use with Java web start technology.
Java Monitoring and Management (JMX) console	Used in conjunction with JMX technology.
Java web services tools	Support the development of Java web service applications.
Experimental tools	These tools are experimental and might not be available with future releases of the Java SE JDK.

Note – The JDK tools API is a set of libraries and accompanying documentation that enables the creation of custom tools. A discussion of this API is beyond the scope of this module. For more information, refer to the Java Platform Debugger Architecture (JPDA) documentation section of the JDK documentation.

JDK Libraries

The JDK libraries consist of predefined data types known as classes. These classes are grouped on the basis of functionality into libraries. In Java technology, a library is called a package. Figure 1-3 highlights the libraries available with the Java SE JDK.

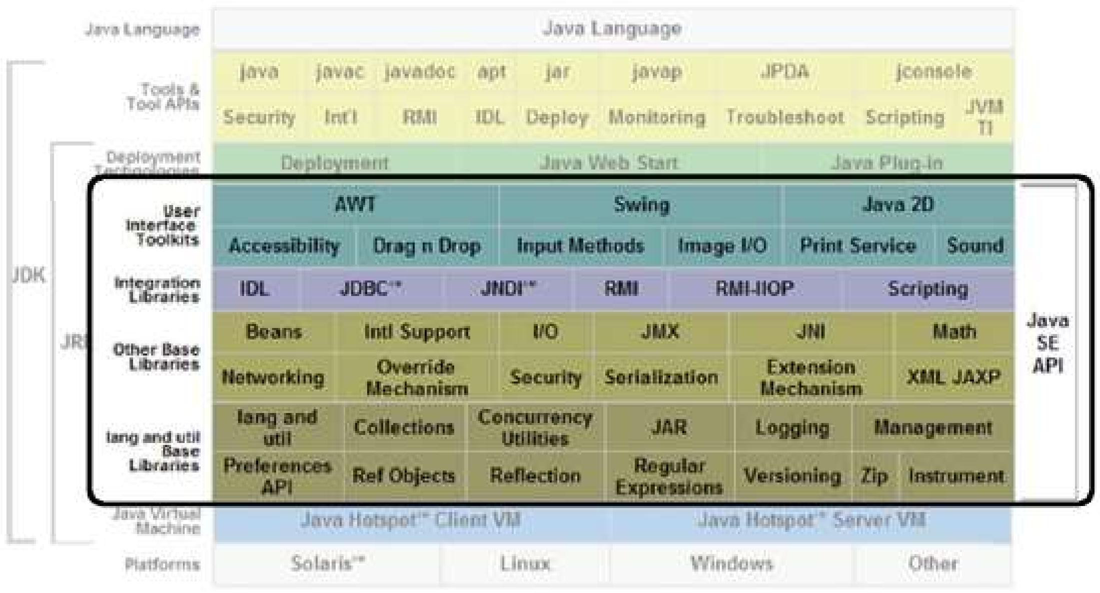


Figure 1-3 The Java SE JDK With the Java SE Libraries Highlighted

The major library groupings can be categorized as follows:

- User interface toolkits

  The user interface toolkits consist of a set of libraries that contain the classes necessary to create graphical user interfaces (GUI). These libraries range from libraries that provide GUI widgets (buttons, labels, tab panes, and so on) to libraries that support drag and drop, sound, accessibility, input, and printing.

- Integration libraries

  The integration libraries provide support for the use of various network technologies that enable the integration of distributed applications. These network technologies include RMI, CORBA, Java™ DataBase Connectivity (JDBC™), and Java Naming and Directory Interface™ (JNDI) API.

- Other base libraries

  This is a general grouping of base-level libraries. This grouping includes the security, internationalization, Java™ Management Extensions (JMX™) and other similar libraries.

- Language and utilities base libraries

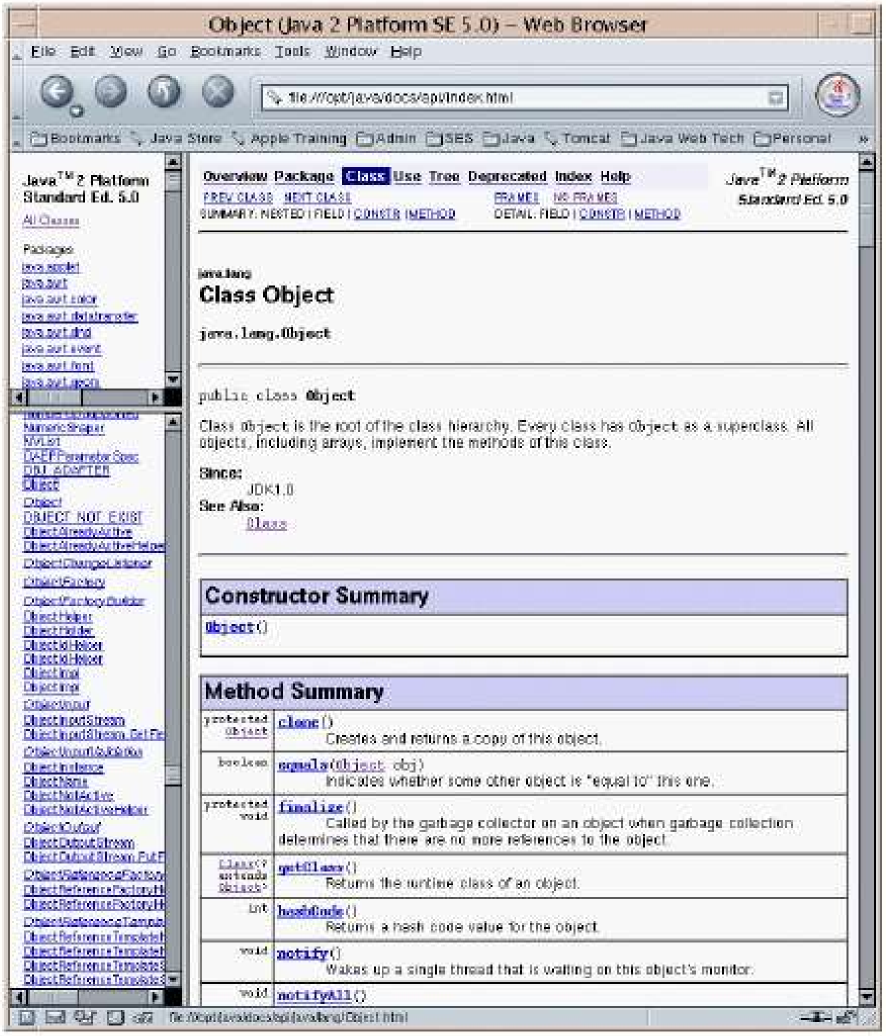
  This grouping consists of the core libraries required to create an application. At the core are the `lang` and `util` libraries. Other libraries included in this group are for concurrency (enabling the creation of multithreaded applications), logging, and management.

Table 1-3 lists a small selection of the standard libraries and a sample of classes from the selected libraries.

Table 1-3 Sample Selection of Standard Libraries and Classes

Library Name	Sample Classes in Library	Purpose
`java.lang`	`Enum`, `Float`, `String`, `Object`	Fundamental classes of the Java programming language
`java.util`	`ArrayList`, `Calendar`, `Date`	Utility classes
`java.io`	`File`, `Reader`, `Writer`	Input and output support classes
`java.math`	`BigDecimal`, `BigInteger`	Arbitrary precision maths support classes
`java.text`	`DateFormat`, `Collator`	Text handling and formatting support classes
`javax.crypto`	`Cipher`, `KeyGenerator`	Cryptography support classes
`java.net`	`Socket`, `URL`, `InetAddress`	Networking support classes
`java.sql`	`ResultSet`, `Date`, `Timestamp`	Structured Query Language (SQL) support classes.
`javax.swing`	`JFrame`, `JPanel`	Graphical user interface (GUI) classes
`javax.xml.parsers`	`DocumentBuilder`, `SAXParser`	Extensible Markup Language (XML) support classes

Documentation for the JDK library classes (also known as Java technology API documentation) is supplied as a set of HTML files. The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. When you select a particular package hotlink, the classes that are members of that package are listed. Selecting a class hotlink from a package page presents a page of information about that class. Figure 1-4 shows one such class.


Object (Java 2 Platform SE 5.0) – Web Browser
File Edit View Go Bookmarks Tools Window Help
file:///opt/java/docs/api/index.html
Bookmarks Java Store Apple Training Admin SES Java Tomcat Java Web Tech Personal
Java™ 2 Platform Standard Ed. 5.0
All Classes
Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
NumericShaper
NVList
OAEPParameterSpec
OBJ_ADAPTER
Object
Object
OBJECT_NOT_EXIST
ObjectAlreadyActive
ObjectAlreadyActiveHelper
ObjectChangeListener
ObjectFactory
ObjectFactoryBuilder
ObjectHelper
ObjectHolder
ObjectIdHelper
ObjectIdHelper
ObjectImpl
ObjectImpl
ObjectInput
ObjectInputStream
ObjectInputStream.GetField
ObjectInputValidation
ObjectInstance
ObjectName
ObjectNotActive
ObjectNotActiveHelper
ObjectOutput
ObjectOutputStream
ObjectOutputStream.PutField
ObjectReferenceFactory
ObjectReferenceFactoryHelper
ObjectReferenceFactoryHolder
ObjectReferenceTemplate
ObjectReferenceTemplateHelper
ObjectReferenceTemplateHolder
ObjectReferenceTemplateSeq
ObjectReferenceTemplateSeqHelper
Overview Package Class Use Tree Deprecated Index Help
Java™ 2 Platform Standard Ed. 5.0
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
file:///opt/java/docs/api/java/lang/Object.html

java.lang

Class Object

```
java.lang.Object
```

```
public class Object
```

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:
JDK1.0

See Also:
Class

Constructor Summary

Object()

Method Summary

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<? extends Object>	getClass() Returns the runtime class of an object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()

Figure 1-4 Java Technology API Documentation

The main sections of a class document include the following:

- The class hierarchy
- A description of the class and its general purpose
- A list of attributes
- A list of constructors
- A list of methods
- A detailed list of attributes with descriptions
- A detailed list of constructors with descriptions and formal parameter lists
- A detailed list of methods with descriptions and formal parameter lists

The Java Virtual Machine (JVM)

This section provides an introduction to the JVM implementation by using an Frequently Asked Questions (FAQ) format to highlight the information you need to know.

- What is a JVM implementation?

  A JVM implementation executes Java technology applications, which consist of compiled Java classes. Compiled Java classes consist of byte code, so to be precise, a JVM implementation loads the classes that compose a Java technology application and executes the byte code contained in the classes. Figure 1-6 illustrates the relationship between a Java technology application, the JVM implementation, the operating system (OS) and the hardware platform.

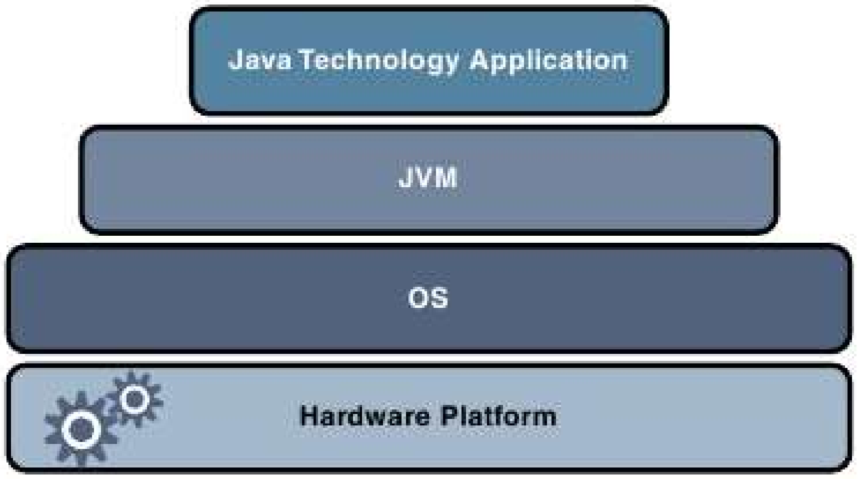


Figure 1-6 The JVM Implementation, the Application, and the OS

- Are JVM implementations platform dependent?

  JVM implementations are platform specific. For example, to execute a Java application on a Solaris OS, you need the JVM implementation that is specific to the Solaris OS. Similarly, to execute a Java technology application on a Windows OS, you need the JVM implementation that is specific to the Windows OS.

- Are Java technology applications platform dependent?

  An application written using the Java programming language is platform *independent*. Such an application can run on any platform that contains a supporting JVM implementation.

- What is a Java HotSpot™ virtual machine?

  The first generation of JVM implementations interpreted byte code. The latest generation of JVM implementations compile the byte code ahead of its execution. This delayed dynamic compiling enables a high level of optimization and performance improvements over statically compiled languages. The JVM compiler dynamically identifies repeatedly executed blocks of code known as hotspots and pays particular attention to the optimization of the byte codes in these hotspots. Hence the terminology hotspot JVM.

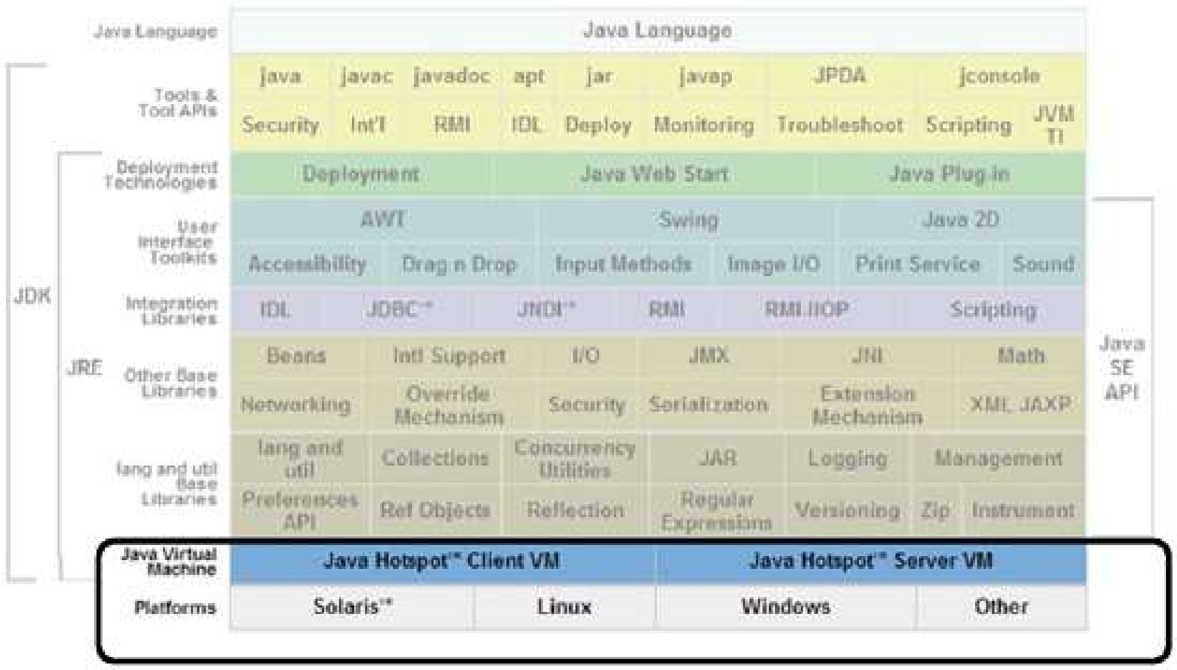
- What is a Java Hotspot™ Client VM?

  On platforms typically used for client applications, the JDK software comes with a virtual machine (VM) implementation called the Java Hotspot Client VM (client VM). The client VM is tuned for reducing start-up time and memory footprint.

- What is a Java Hotspot Server VM?

  On all platforms, the JDK software comes with an implementation of the Java virtual machine called the Java Hotspot Server VM (server VM). The server VM is designed for maximum program execution speed.

Figure 1-3 highlights the JVM implementations and supported platforms available for the JDK software.



Java Language
Java Language
Tools & Tool APIs
java javac javadoc apt jar javap JPDA jconsole
Security Int'l RMI IDL Deploy Monitoring Troubleshoot Scripting JVM TI
Deployment Technologies
Deployment Java Web Start Java Plug-in
User Interface Toolkits
AWT Swing Java 2D
Accessibility Drag n Drop Input Methods Image I/O Print Service Sound
Integration Libraries
IDL JDBC™ JNDI™ RMI RMI-IIOP Scripting
Other Base Libraries
Beans Intl Support I/O JMX JNI Math
Networking Override Mechanism Security Serialization Extension Mechanism XML JAXP
lang and util Base Libraries
lang and util Collections Concurrency Utilities JAR Logging Management
Preferences API Ref Objects Reflection Regular Expressions Versioning Zip Instrument
Java Virtual Machine
Java Hotspot™ Client VM
Java Hotspot™ Server VM
Platforms
Solaris™ Linux Windows Other
JDK
JRE
Java SE API

Figure 1-7 The JDK Software Highlighting the JVM Implementations and Supported Platforms

The Java™ Runtime Environment (JRE™)

The JRE™ software is a subset of the JDK software and contains all the components of the JDK that are required to execute a Java technology application. It does not contain the components of the JDK required to develop a Java technology application.

Figure 1-8 shows the components that are common to both the JDK and the JRE.

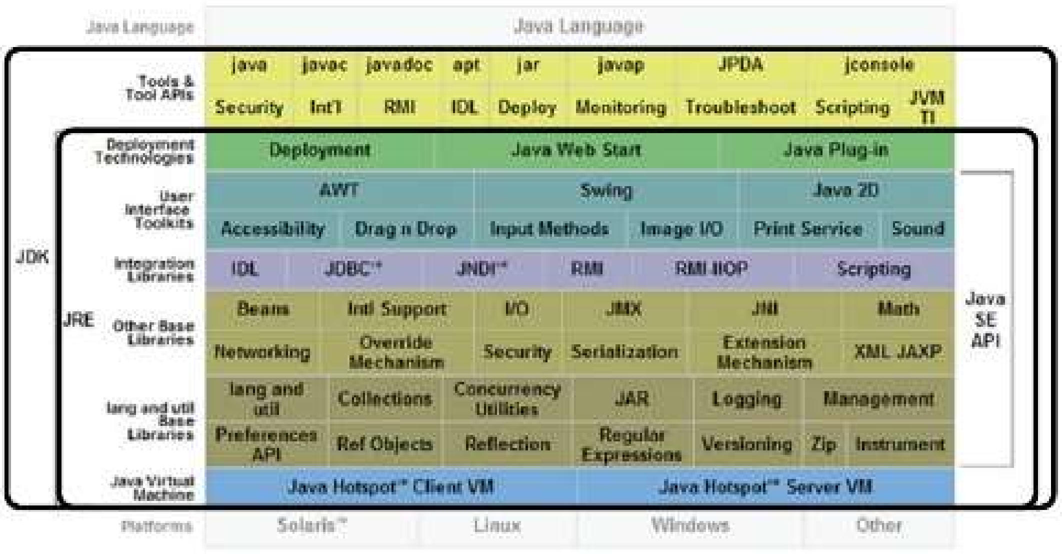


Figure 1-8 Comparing the JDK With the JRE

Examining Java Application Loading and Execution

Figure 1-9 illustrates the loading and execution of a Java application by the JVM implementation.

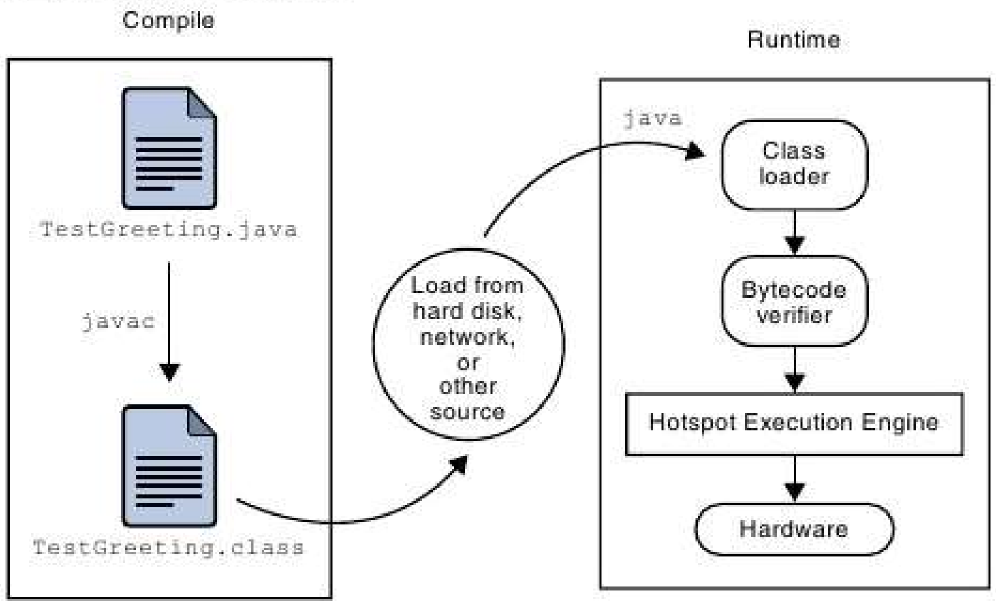


Figure 1-9 Operation of the JRE

Java software source files are *compiled* in the sense that they are converted into a set of byte codes from the text format in which you write them. The byte codes are stored in `.class` files.

At runtime, the byte codes that make up a Java program are loaded, checked, and run by the Java HotSpot execution engine. In the case of applets, you can download the byte codes, and then they are executed by the JVM implementation built into the browser.

The following section provides a more comprehensive discussion of the three main tasks performed by the JVM implementation:

- Loads code – Performed by the class loader
- Verifies code – Performed by the byte-code verifier
- Executes code – Performed by the execution engine

The Class Loader

The class loader loads all classes needed for the execution of a program. The class loader adds security by separating the namespaces for the classes of the local file system from those imported from network sources. This limits any Trojan Horse applications, because local classes are always loaded first.

After all of the classes have been loaded, the memory layout of the executable file is determined. At this point, specific memory addresses are assigned to symbolic references and the lookup table is created. Because memory layout occurs at runtime, the Java technology interpreter adds protection against unauthorized access into the restricted areas of code.

The Byte-Code Verifier

Java software code passes several tests before running on your machine. The JVM implementation puts the code through a byte-code verifier that tests the format of the code fragments and checks code fragments for illegal code, which is code that forges pointers, violates access rights on objects, or attempts to change object type.

Note – All class files imported across the network pass through the byte-code verifier.

Verification Process

The byte-code verifier makes four passes on the code in a program. It ensures that the code adheres to the JVM specifications and does not violate system integrity. If the verifier completes all four passes without returning an error message, then the following is ensured:

- The classes adhere to the class file format of the JVM specification.
- There are no access restriction violations.
- The code causes no operand stack overflows or underflows.
- The types of parameters for all operational codes are correct.
- No illegal data conversions, such as converting integers to object references, have occurred.

Creating a Simple Java Application

Like any other programming language, you use the Java programming language to create applications. Code 1-2 and Code 1-3 show how to create a simple Java application that prints a greeting to the world.

Code 1-2 The `TestGreeting.java` Application

```
//
// Sample "Hello World" application
//
public class TestGreeting {
   public static void main (String[] args) {
      Greeting hello = new Greeting();
      hello.greet();
   }
}
```

Code 1-3 The `Greeting.java` Class

```
public class Greeting {
    public void greet() {
       System.out.println("hi");
    }
}
```

The `TestGreeting` Application

This section describes the `TestGreeting` application.

Code 1-4 shows lines 1 to 3 of the sample application.

Code 1-4 Lines 1 to 3

```
1   //
2   // Sample "Hello World" application
3   //
```

Lines 1 to 3 in the program are comment lines, indicated by the `//`.

Code 1-5 Line 4

```
4   public class TestGreeting  {
```

Line 4 declares the class name as `TestGreeting`. A class name specified in a source file creates a *`classname`*`.class file` when the source file is compiled. If you do not specify a target directory for the compiler to use, this class file is in the same directory as the source code. In this case, the compiler creates a file called `TestGreeting.class`. It contains the compiled code for the public class `TestGreeting`.

Code 1-6 Line 5

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

Line 5 is where the program starts to execute. The Java technology interpreter must find this defined exactly as given or it refuses to run the program.

Other programming languages, notably C and C++, also use the `main()` declaration as the starting point for execution. The various parts of this declaration are briefly described here. The details are covered later in this course.

If the program is given any arguments on its command line, these are passed into the `main()` method in an array of `String` called `args`. In this example, no arguments are used.

The following describes each element of Line 5:

- `public` – The method `main()` can be accessed by anything, including the Java technology interpreter.
- `static` – This keyword tells the compiler that the `main()` method is usable in the context of the class `TestGreeting`. No instance of the class is needed to execute static methods.
- `void` – This keyword indicates that the method `main()` does not return any value. This is important because the Java programming language performs careful type-checking to confirm that the methods called return the types with which they were declared.
- `String` *`args[]`* – This method declares the single parameter to the `main` method, `args`, and has the type of a `String` array. When this method is called, the `args` parameter contains the arguments typed on the command line following the class name; for example:

  ```
  java TestGreeting  args[0] args[1] . . .
  ```

Code 1-7 Line 6

```
6       Greeting hello = new Greeting();
```

Line 6 illustrates how to create an object, referred to by the `hello` variable. The `new Greeting` syntax tells the Java technology interpreter to construct a new object of the class `Greeting`.

Code 1-8 Line 7

```
7       hello.greet();
```

Lines 7 demonstrates an object method call. This call tells the `hello` object to `greet` the world. The implementation of this method is shown on Lines 3–5 of the `Greeting.java` file.

Code 1-9 Lines 8 and 9

```
8     }
9   }
```

Lines 8 and 9 of the program, the two braces, close the method `main()` and the class `TestGreeting`, respectively.

The `Greeting` Class

This section describes the `Greeting` class.

Code 1-10 Line 1

```
1    public class Greeting {
```

Line 1 declares the `Greeting` class.

Code 1-11 Lines 2 to 4

```
2    public void greet() {
3        System.out.println("hi");
4    }
```

Lines 2 to 4 demonstrate the declaration of a method. This method is declared `public`, making it accessible to the `TestGreeting` program. It does not return a value, so `void` is used as the return type.

The `greet` method sends a string message to the standard output stream. The `println()` method is used to write this message to the standard output stream.

Code 1-12 Line 5

```
5    }
```

Line 5 closes the class declaration for the `Greeting` class.

Compiling and Running the `TestGreeting` Program

After you have created the `TestGreeting.java` source file, compile it by typing the following line:

```
javac TestGreeting.java
```

If the compiler does not return any messages, the new file `TestGreeting.class` is stored in the same directory as the source file, unless specified otherwise. The `Greeting.java` file has been compiled into `Greeting.class`. This is done automatically by the compiler because the `TestGreeting` class uses the `Greeting` class.

To run your `TestGreeting` application, use the Java technology interpreter. The executables for the Java technology tools (`javac`, `java`, `javadoc`, and so on) are located in the `bin` directory.

```
java TestGreeting
```

Note – You must set the `PATH` environment variable to find `java` and `javac`; ensure that it includes *`java_root`*`/bin` (where *`java_root`* represents the directory root where the Java technology software is installed).

Troubleshooting the Compilation

The following sections describe errors that you might encounter when compiling code.

Compile-Time Errors

The following are common errors seen at compile time, with examples of compiler-generated messages. Your messages can vary depending on which version of the JDK you are using.

- `javac: Command not found`

  The `PATH` variable is not set properly to include the `javac` compiler. The `javac` compiler is located in the `bin` directory below the installed JDK software directory.

-
  ```
  Greeting.java:4:cannot resolve symbol
  symbol   : method printl   (java.lang.String)
  location: class java.io.PrintStream
  System.out.printl("hi");
            ^
  ```

  The method name `println` is typed incorrectly.

- Class and file naming

  If the `.java` file contains a public class, then it must have the same file name as that class. For example, the definition of the class in the previous example is:

  ```
  public class TestGreeting
  ```

  The name of the source file must be `TestGreeting.java`. If you named the file `TestGreet.java`, then you would get the error message:

  ```
  TestGreet.java:4: Public class TestGreeting must be
  defined in a file called "TestGreeting.java".
  ```

- Class count

  You should declare only one top-level, non-static class to be public in each source file, and it must have the same name as the source file. If you have more than one public class, then you will get the same message as in the previous bullet for every public class in the file that does not have the same name as the file.

Runtime Errors

Some of the errors generated when typing `java TestGreeting` are:

- `Can't find class TestGreeting`

  Generally, this means that the class name specified on the command line was spelled differently than the *`filename`*`.class` file. The Java programming language is *case-sensitive*.

  For example,

  ```
  public class TestGreet {
  ```

  creates a `TestGreet.class`, which is not the class name (`TestGreeting.class`) that the runtime expected.

- `Exception in thread "main" java.lang.NoSuchMethodError: main`

  This means that the class you told the interpreter to execute does not have a static `main` method. There might be a main method, but it might not be declared with the static keyword or it might have the wrong parameters declared, such as:

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

  In this example, `args` is a single string, not an array of strings.

  ```
  public static void main() {
  ```

  In this example, the coder forgot to include any parameter list.

Figure 1-10 illustrates how Java technology programs can be compiled and then run on the JVM implementation. There are many implementations of JVM on different hardware and operating system platforms.

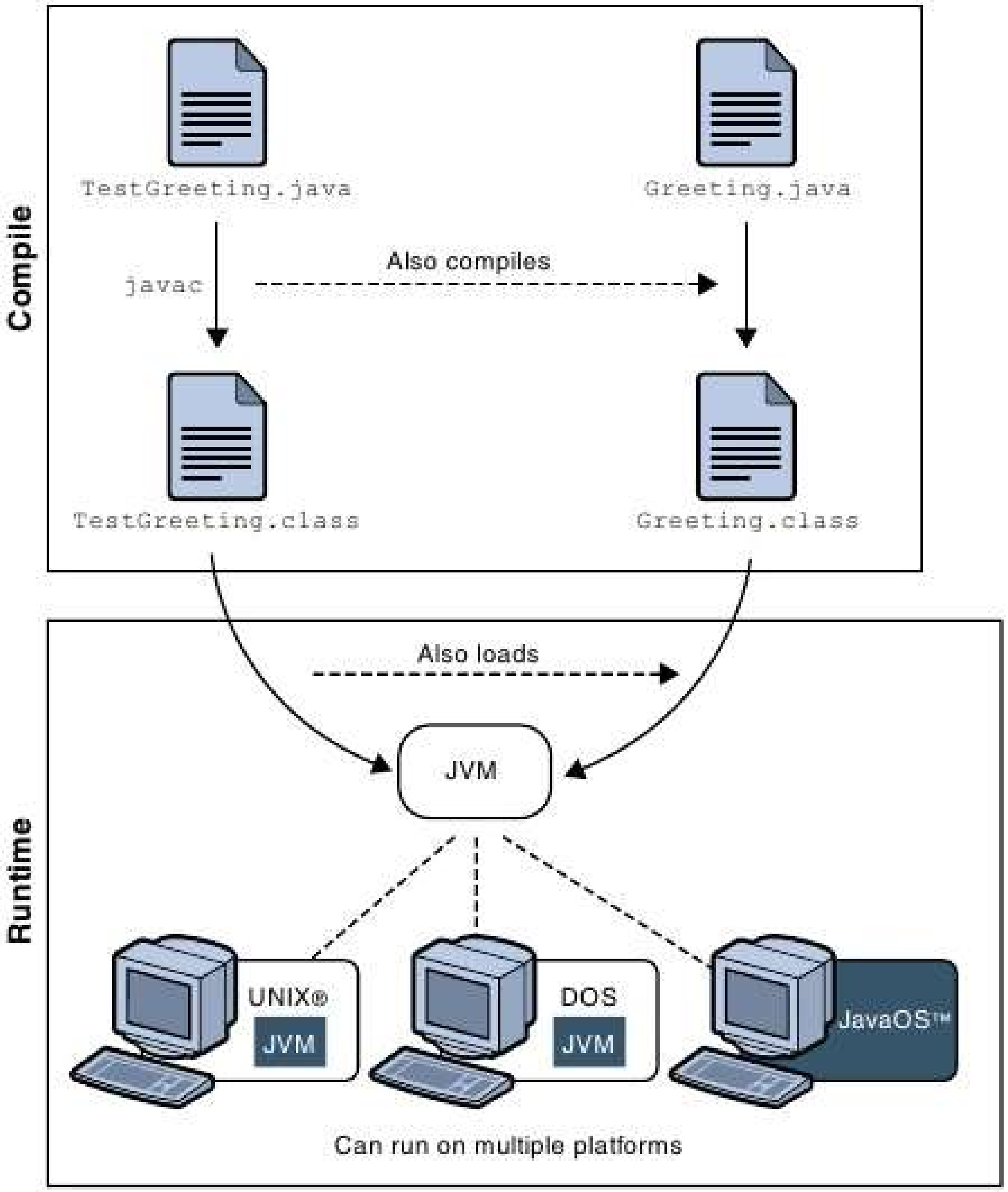


Figure 1-10 Java Technology Runtime Environment