

Packaging, Compiling and Interpreting Java Code

The Java Platform

Platform Independence

When the Java language is compiled, it is targeted for execution on the Java virtual machine, or JVM, instead of a specific hardware architecture. The compiled Java code is called *bytecode*. The only requirement for the code to work on any computer is the presence of a compatible JVM.

JVMs share a common Java core, but platform independence is limited to compatible versions.

Java's Object Oriented Philosophy

Java was conceived as an object-oriented language, in contrast to the C language, which is procedural. An object-oriented language organizes related data and code together—a process called *encapsulation*. A properly encapsulated object uses data protection and exposes only some of its data and methods.

Object-oriented design also encourages *abstraction*, the ability to generalize algorithms. Abstraction facilitates code reuse and flexibility. Inheritance and polymorphism are key concepts in creating reusable code.

Robust and Secure

Security and robustness were major design goals when Java was created.

With Java, the JVM periodically runs the garbage collector, which looks for any objects that have gone out of scope or that are no longer referenced, and it automatically deallocates their memory. This frees the developer from this manual, error-prone task and increases robustness by ensuring that memory is properly managed.

Understand Packages

Package Design

Packaging is encouraged by Java coding standards to decrease the likelihood of classes colliding in the same namespace. The package name plus the class names creates the *fully qualified class name*. Packaging your classes also promotes code reuse, maintainability, and the object-oriented principle of encapsulation and modularity.

package and import Statements

To place a source file into a package, you use the package statement at the beginning of that file. You may use zero or one package statements per source file. To import classes from other packages into your source file, you may use the import statement or you may precede each class name with its package name. The java.lang package that houses the core language classes is imported by default.

The package Statement

The package statement includes the package keyword, followed by the package path delimited with periods. Package statements have the following attributes:

- They are optional.
- They are limited to one per source file.
- Standard coding convention for package statements reverses the domain name of the organization or group creating the package. For example, the owners of the domain name ocajexam.com may use the following package name for a utilities package: com.ocajexam.utilities.
- Package names equate to directory structures. The package name com.ocajexam.utils would equate to the directory com/ocajexam/utils. If a class includes a package statement that does not map to the relative directory structure, the class will not be usable.
- The package names beginning with java.* and javax.* are reserved.
- Package names should be lowercase. Individual words within the package name should be separated by underscores.

The import Statement

An import statement enables you to include source code from other classes into a source file at compile time. The import statement includes the import keyword followed by the package path delimited with periods and ending with a class name or an asterisk. These import statements occur

after the optional package statement and before the class definition. Each import statement can relate to one package only.

For maintenance purposes, it is better that you import your classes explicitly. This will allow the programmer to determine quickly which external classes are used throughout the class. For example, rather than using `import java.util.`, use `import java.util.Vector`. In this real-world example, the coder would quickly see (with the latter approach) that the class imports only one class and it is a collection type. In this case, it is a legacy type and the determination to update the class with a newer collection type could be done quickly.*

The static import Statement

Simply put, static import statements allow you to import static members.

Understand Package-Derived Classes

The following sections address these APIs:

1. Java Utilities API:

- The Java Utilities API is contained in the package `java.util`.
- Categories of classes:
 - Java Collections Framework;
 - date and time facilities;
 - internationalization;
 - miscellaneous utility classes

TABLE 1-4 Various Classes of the Java Collections Framework

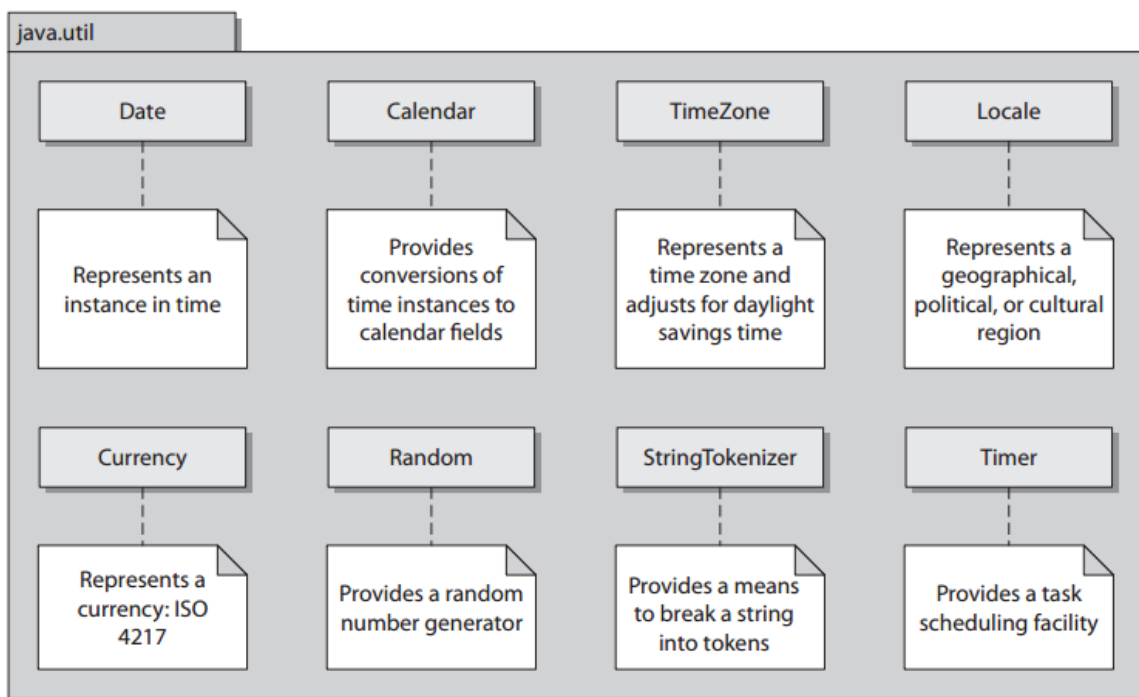
Interface	Implementations	Description
List	ArrayList, LinkedList, Vector	Data structures based on positional access.
Map	HashMap, Hashtable, LinkedHashMap, TreeMap	Data structures that map keys to values.
Set	HashSet, LinkedHashSet, TreeSet	Data structures based on element uniqueness.
Queue	PriorityQueue	Queues typically order elements in a first in, first out (FIFO) manner. Priority queues order elements according to a supplied comparator.

- Legacy date and time facilities are represented by the `Date`, `Calendar`, and `TimeZone` classes.

- Geographical regions are represented by the Locale class.
- The Currency class represents currencies per the ISO 4217 standard.
- A random-number generator is provided by the Random class.
- And StringTokenizer breaks strings into tokens.

Many packages have related classes and interfaces with unique functionality, so they are included in their own subpackages. For example, regular expressions are stored in a subpackage of the Java utilities (java.util) package. The subpackage is named java.util.regex and houses the Matcher and Pattern classes. Where needed, consider creating subpackages for your own projects.

FIGURE 1-2 Various utility classes

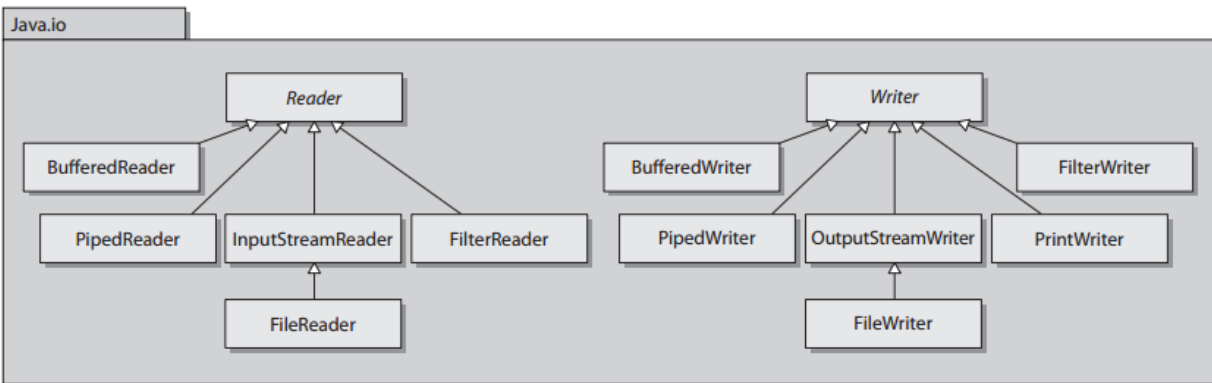


2. Java Basic Input/ Output API:

- The Java Basic Input/Output API is contained in the package `java.io`.
- This API provides functionality for general system input and output in relation to data streams, serialization, and the file system.
- Data-stream classes include byte-stream subclasses of the `InputStream` and `OutputStream` classes.
- Data-stream classes also include character-stream subclasses of the `Reader` and `Writer` classes.
- Other important `java.io` classes and interfaces include `File`, `FileDescriptor`, `FilenameFilter`, and `RandomAccessFile`.

- The File class provides a representation of file and directory pathnames.
- The FileDescriptor class provides a means to function as a handle for opening files and sockets.
- The FilenameFilter interface, as its name implies, defines the functionality to filter filenames.
- The RandomAccessFile class allows for the reading and writing of files to specified locations.
- In JDK 7, the NIO.2 API was introduced in the package java.nio. This included the useful Paths interface, the Path class, and the Files class. The Files class has lines, list, walk, and find methods that work hand-in-hand with the Stream API.

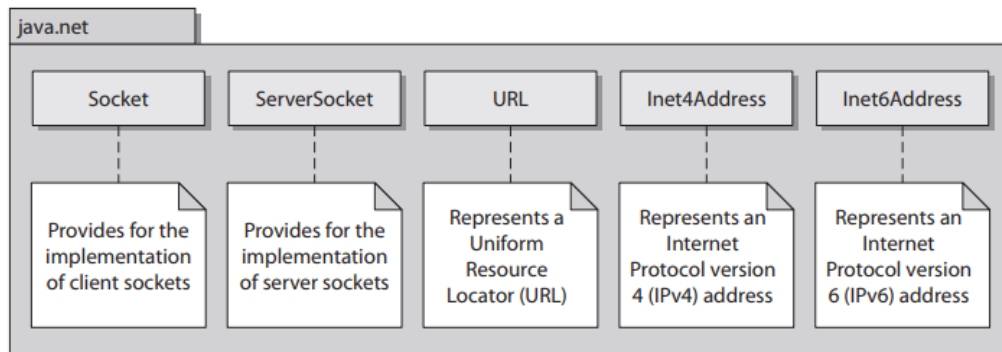
FIGURE 1-3 Reader and Writer class hierarchy



3. The Java Networking API

- The Java Networking API is contained in the package `java.net`.
- This API provides functionality in support of creating network applications.
- The improved performance I/O API (`java.nio`) package, which provides for nonblocking networking and the socket factory support package (`javax.net`), is not included on the exam.

FIGURE 1-4 Various classes of the Networking API

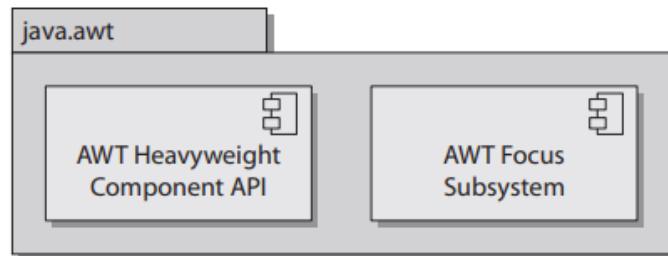


4. Java Abstract Window Toolkit API

- The Java Abstract Window Toolkit API is contained in the package `java.awt`.
- This API provides functionality for creating heavyweight components with regard to creating user interfaces and painting associated graphics and images.
- The AWT API was Java's original GUI API and has been superseded by the Swing API.
- The AWT Focus subsystem provides for navigation control between components.

FIGURE 1-5

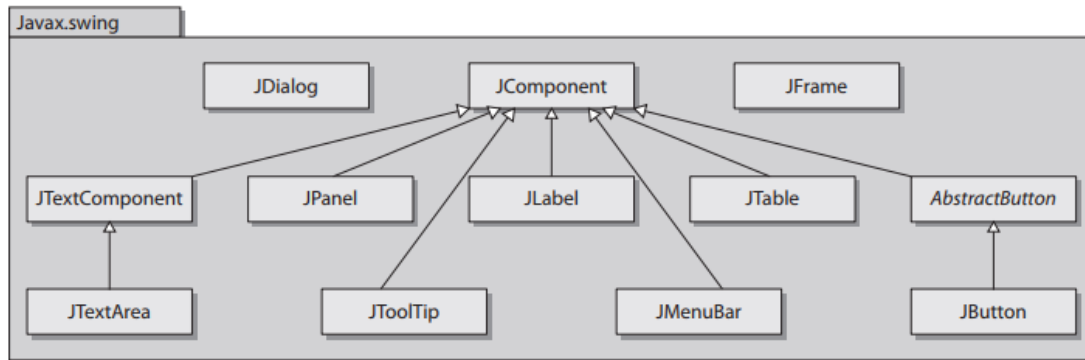
AWT major
elements



5. Java Swing API

- The Java Swing API is contained in the package `javax.swing`.
- This API provides functionality for creating lightweight (pure-Java) containers and components.
- The Swing API, providing a more sophisticated set of GUI components, supersedes the AWT API.
- Many of the Swing classes are simply prefaced with the addition of “J” in contrast to the legacy AWT component equivalent. For example, Swing uses the class `JButton` to represent a button container, whereas AWT uses the class `Button`.
- Swing also provides look-and-feel support, allowing for universal style changes of the GUI's components. Other features include tooltips, accessibility functionality, an event model, and enhanced components such as tables, trees, text components, sliders, and progress bars.
- Swing takes advantage of the model-view-controller (MVC) architecture. The *model* represents the current state of each component. The *view* is the representation of the components on the screen. The *controller* is the functionality that ties the UI components to events.

FIGURE 1-6 Various classes of the Swing API



It's good to be familiar with the package prefixes `java` and `javax`. The prefix `java` is commonly used for the core packages. The prefix `javax` is commonly used for packages that comprise Java standard extensions. Take special notice of the prefix usage in the AWT and Swing APIs: `java.awt` and `javax.swing`. Also note that JavaFX will be replacing Swing as the GUI toolkit for Java SE. Its prefix is `javafx`.

6. JavaFX API

- JavaFX is Java's latest technology for creating rich user interfaces.
- It is designed to provide lightweight, hardware-accelerated interfaces.
- JavaFX provides a similar set of features to the Swing library. JavaFX is intended to replace Swing in the same manner that Swing replaced AWT. The JavaFX libraries are part of the `javafx` package.
- JavaFX best practices suggest that the MVC architecture be used when designing applications.
- FXML, an XML-based markup language, has been created for defining user interfaces.
- Many of the more than 60 UI controls can be styled by using Cascading Style Sheets (CSS). These features together represent a powerful new way to create user interfaces.
- JavaFX makes going from whiteboard design to implemented software faster than ever before.

JavaFX is the latest technology for creating user interfaces. Oracle is actively promoting this technology as the go-to tool kit. However, the Swing libraries are not going away anytime soon. In Java 8, both JavaFX and Swing are fully supported and can be used interchangeably. The `SwingNode` class allows Swing elements to be embedded in JavaFX. The `JFXPanel` will allow the reverse so that JavaFX elements can be used in a Swing applications.

Understand Class Structure

Naming Conventions

Naming conventions are rules for the usage and application of characters in creation of identifiers, methods, class names, and so forth, throughout your code base. Naming conventions exist for the primary goal of making Java programs more readable, and therefore maintainable.

TABLE 1-5 Java Naming Conventions

Element	Lettering	Characteristic	Example
Class name	Begins uppercase, continues CamelCase	Noun	SpaceShip
Interface name	Begins uppercase, continues CamelCase	Adjective ending with “able” or “ible” when providing a capability; otherwise a noun	Dockable
Method name	Begins lowercase, continues CamelCase	Verb, may include adjective or noun	orbit
Instance and static variables names	Begins lowercase, continues CamelCase	Noun	moon
Parameters and local variables	Begins lowercase, continues CamelCase if multiple words are necessary	Single words, acronyms, or abbreviations	lop (line of position)
Generic type parameters	Single uppercase letter	The letter <i>T</i> is recommended	T
Constant	All uppercase letters	Multiple words separated by underscores	LEAGUE
Enumeration	Begins uppercase, continues CamelCase; the set of objects should be all uppercase	Noun	enum Occupation {MANNED, SEMI_MANNED, UNMANNED}
Package	All lowercase letters	Public packages should be the reversed domain name of the org	com.ocajexam.sim

Separators and Other Java Source Symbols

TABLE 1-6 Symbols and Separators

Symbol	Description	Purpose
()	Parentheses	Encloses set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions
{ }	Braces	Encloses blocks of codes, initializes arrays
[]	Box brackets	Declares array types, initializes arrays
< >	Angle brackets	Encloses generics
;	Semicolon	Terminates statement at the end of a line
,	Comma	Separates identifiers in variable declarations, separates values, separates expressions in a for loop
.	Period	Delineates package names, selects an object's field or method, supports method chaining
:	Colon	Follows loop labels
' '	Single quotes	Defines a single character
->	Arrow operator	Separates left-side parameters from the right-side expression
" "	Double quotes	Defines a string of characters
//	Forward slashes	Indicates a single-line comment
/* */	Forward slashes with asterisks	Indicates a blocked comment for multiple lines
/** */	Forward slashes with a double and single asterisk	Indicates Javadoc comments

Java Class Structure

Every Java program has at least one class. A Java class has a signature, optional constructors, optional data members (fields), and optional methods, as outlined here:

```
[modifiers] class classIdentifier [extends superClassIdentifier]
                                   [implements interfaceIdentifier1,
                                   interfaceIdentifier2, etc.] {
    [data members]
    [constructors]
    [methods]
}
```

Each class may extend one and only one superclass. Each class may implement one or more interfaces. Interfaces are separated by commas.

The override annotation (`@Override`) indicates that a method declaration intends on overriding a method declaration in the class's supertype.

Compile and Interpret Java Code

The Java Development Kit (JDK) includes several utilities for compiling, debugging, and running Java applications.

Java Compiler

The main method used as the entry point of the executed code. When the program is started, this is the first method to be called by the JVM.

1. Compiling Your Source Code:

The Java compiler simply converts Java source files into bytecode. The Java compiler's usage is as follows:

```
javac [options] [source files]
```

This will result in a bytecode file being produced with the same preface. This bytecode file will be placed into the same folder as the source code, unless the code is packaged and/or it's been told via a command-line option to be placed somewhere else.

You will find that many projects use Apache Ant and/or Maven build environments. Understanding the fundamentals of the command-line tools is necessary for writing and maintaining the scripts associated with these build products.

- **Compiling Your Source Code with -d Option:**

You may want to specify explicitly where you would like the compiled bytecode class files to go. You can accomplish this by using the -d option:

```
javac -d classes MyClass.java
```

This command-line structure will place the class file into the classes directory, and since the source code was packaged (that is, the source file included a package statement), the bytecode will be placed into the relative subdirectories.

- **Compiling Your Code with the -classpath Option:**

If you want to compile your application with user-defined classes and packages, you may need to tell the JVM where to look by specifying them in the classpath. This classpath inclusion is accomplished by telling the compiler where the

desired classes and packages are with the `-cp` or `-classpath` command-line option.

Note that you do not need to include the `classpath` option if the classpath is defined with the `CLASSPATH` environment variable, or if the desired files are in the present working directory.

On Windows systems, classpath directories are delimited with backward slashes and paths are delimited with semicolons and the period represents the present (or current) working directory:

```
-classpath .;\dir_a\classes_a\;\dir_b\classes_b\
```

2. Java Interpreter

```
java [-options] class [args...]
```

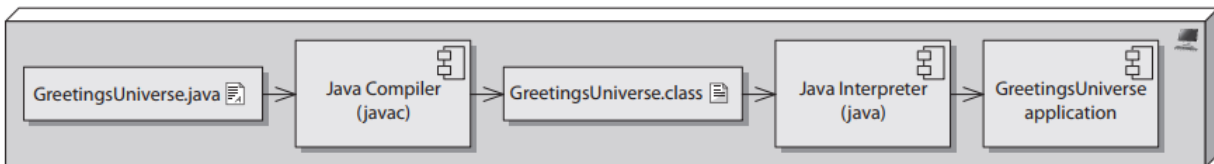
- **Interpreting your Bytecode:**

The Java interpreter is invoked with the `java[.exe]` command. Use it to interpret bytecode and execute your program.

You can easily invoke the interpreter on a class that's not packaged, as follows:

```
java MainClass
```

FIGURE 1-8 Bytecode conversion



You can optionally start the program with the `javaw` command on Microsoft Windows to exclude the command window. This is a nice feature with GUI-based applications, because the console window is often not necessary.

```
javaw.exe MainClass
```

- **Interpreting Your Code with the `-classpath` Option:**

When interpreting your code, you may need to define where certain classes and packages are located. You can find your classes at runtime when you include

the `-cp` or `-classpath` option with the interpreter. If the classes you want to include are packaged, then you can start your application by pointing the full path of the application to the base directory of classes, as in the following interpreter invocation:

```
java -cp classes com.ocajexam.tutorial.MainClass
```

- **Interpreting Your Bytecode with the `-D` Option:**

The `-D` command-line option allows for the setting of new property values. The usage is as follows:

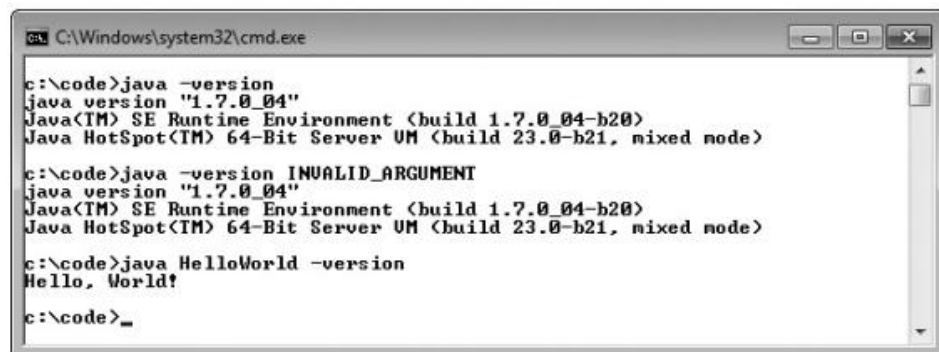
```
java -D<name>=<value> class
```

- **Retrieving the Version of the Interpreter with the `-version` Option**

The `-version` command-line option is used with the Java interpreter to return the version of the JVM and exit.

FIGURE 1-9

The `-version`
command-line
option



```
C:\Windows\system32\cmd.exe

c:\code>java -version
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java -version INVALID_ARGUMENT
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java HelloWorld -version
Hello, World!

c:\code>_
```

Exercise 1-2

```
package com.ocajexam.tutorial;

import com.ocajexam.tutorial.planets.Earth;
import com.ocajexam.tutorial.planets.Mars;
import com.ocajexam.tutorial.planets.Venus;

public class GreetingsUniverse {
    public static void main (String [] args) {
        System.out.println("Greetings, Universe!");
        Earth e = new Earth();
```

<pre> Mars m = new Mars(); Venus v = new Venus(); } } </pre>
<pre> package com.ocajexam.tutorial.planets; public class Earth { public Earth() { System.out.println("Hello from Earth!"); } } </pre>
<pre> package com.ocajexam.tutorial.planets; public class Mars { public Mars () { System.out.println("Hello from Mars!"); } } </pre>
<pre> package com.ocajexam.tutorial.planets; public class Venus { public Venus() { System.out.println("Hello from Venus!"); } } </pre>

1. Compile the program:

<i>javac -cp . com\ocajexam\tutorial\GreetingsUniverse.java</i>
<i>javac -cp . com\ocajexam\tutorial\planets\Earth.java</i>
<i>javac -cp . com\ocajexam\tutorial\planets\Mars.java</i>
<i>javac -cp . com\ocajexam\tutorial\planets\Venus.java</i>

2. Run the program to ensure it is error free:

```
java -cp . com.ocajexam.tutorial.GreetingsUniverse.java
```

CERTIFICATION SUMMARY

TWO-MINUTE DRILL:

1. Understand Packages:

- Packages are containers for classes.
- A package statement defines the directory path where files are stored.
- A package statement uses periods for delimitation.
- Package names should be lowercase and separated with underscores between words.
- Package names beginning with java.* and javax.* are reserved.
- There can be zero or one package statement per source file.
- An import statement is used to include source code from external classes.
- An import statement occurs after the optional package statement and before the class definition.
- An import statement can define a specific class name to import.
- An import statement can use an asterisk to include all classes within a given package.

2. Understand Package-Derived Classes:

- The Java Abstract Window Toolkit API is included in the java.awt package and subpackages.
- The java.awt package includes GUI creation and painting graphics and images functionality.
- The Java Swing API is included in the javax.swing package and subpackages.
- The javax.swing package includes classes and interfaces that support lightweight GUI component functionality.
- The Java Basic Input/Output-related classes are contained in the java.io package.
- The java.io package includes classes and interfaces that support input/ output functionality of the file system, data streams, and serialization.
- Java networking classes are included in the java.net package.

- The java.net package includes classes and interfaces that support basic networking functionality that is also extended by the javax.net package.
- Fundamental Java utilities are included in the java.util package.
- The java.util package and subpackages include classes and interfaces that support the Java Collections Framework, legacy collection classes, event model, date and time facilities, and internationalization functionality.

3. Understand Class Structure:

- Naming conventions are used to make Java programs more readable and maintainable.
- Naming conventions are applied to several Java elements, including class names, interface names, method names, instance and static variable names, parameter and local variable names, generic type parameter names, constant names, enumeration names, and package names.
- The preferred order of presenting elements in a class is data members, followed by constructors, followed by methods. Note that the inclusion of each type of element is optional.

4. Compile and Interpret Java Code:

- The Java compiler is invoked with the javac[.exe] command.
- The .exe extension is optional on Microsoft Windows machines and is not present on UNIX-like systems.
- The compiler's -d command-line option defines where compiled class files should be placed.
- The compiler's -d command-line option will include the package location if the class has been declared with a package statement.
- The compiler's -classpath command-line option defines directory paths in search of classes.
- The Java interpreter is invoked with the java[.exe] command.
- The interpreter's -classpath switch defines directory paths to use at runtime.
- The interpreter's -D command-line option allows for the setting of system property values.
- The interpreter's syntax for the -D command-line option is -Dproperty=value.
- The interpreter's -version command-line option is used to return the version of the JVM and exit.
- The -h command-line option can be applied either to the compiler or the interpreter to print out the tool's usage information.