Java Source File Structure

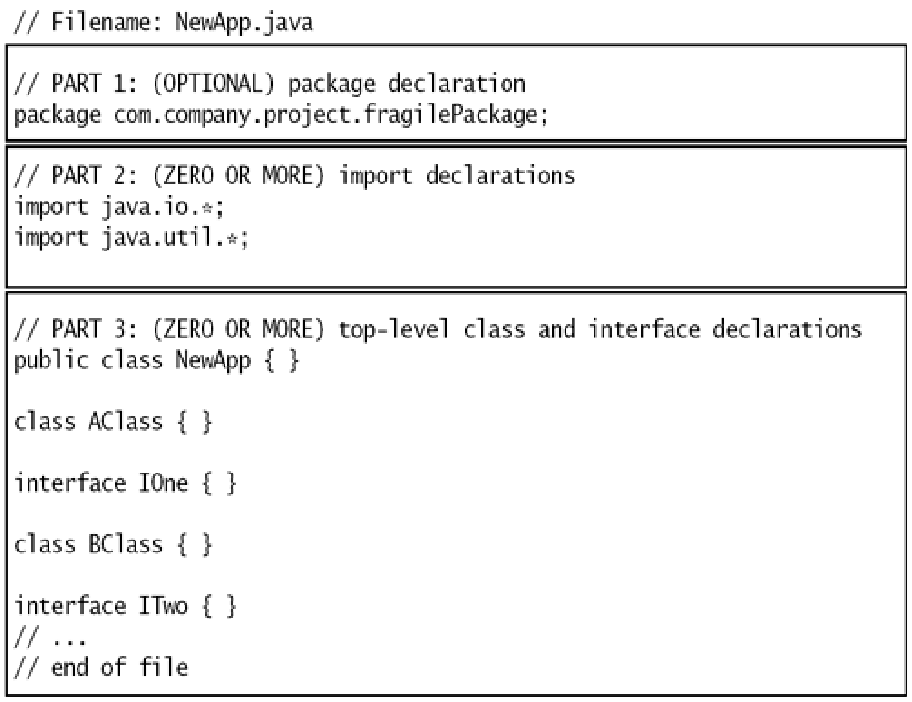
The structure of a skeletal Java source file is depicted in Figure 2.2. A Java source file can have the following elements that, if present, must be specified in the following order:

1. An optional `package` declaration to specify a package name.
2. Zero or more `import` declarations. Since `import` declarations introduce class and interface names in the source code, they must be placed before any type declarations.
3. Any number of top-level class and interface declarations. Since these declarations belong to the same package, they are said to be defined at the top level, which is the package level.

   The classes and interfaces can be defined in any order. Class and interface declarations are collectively known as type declarations. Technically, a source file need not have any such definitions, but that is hardly useful.

   The Java 2 SDK imposes the restriction that at the most one `public` class definition per source file can be defined. If a `public` class is defined, the file name must match this `public` class. If the `public` class name is `NewApp`, then the file name must be `NewApp.java`.

Figure 2.2. Java Source File Structure



```
// Filename: NewApp.java

// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilePackage;

// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;

// PART 3: (ZERO OR MORE) top-level class and interface declarations
public class NewApp { }

class AClass { }

interface IOne { }

class BClass { }

interface ITwo { }
// ...
// end of file
```

Note that except for the `package` and the `import` statements, all code is encapsulated in classes and interfaces. No such restriction applies to comments and white space.

Compilation

In Java, programs are not compiled into executable files; they are compiled into bytecode, which the JVM (Java Virtual Machine) then executes at runtime.

Java source code is compiled into bytecode when we use the `javac` compiler.

The bytecode gets saved on the disk with the file extension `.class`.

When the program is to be run, the bytecode is converted, using the just-in-time (JIT) compiler. The result is machine code which is then fed to the memory and is executed.

Java code needs to be compiled twice in order to be executed:

1. Java programs need to be compiled to bytecode.
2. When the bytecode is run, it needs to be converted to machine code.

The Java classes/bytecode are compiled to machine code and loaded into memory by the JVM when needed the first time. This is different from other languages like C/C++ where programs are to be compiled to machine code and linked to create an executable file before it can be executed.

Quick compilation procedure

To execute your first Java program, follow the instructions below:

1. Proceed only if you have successfully installed and configured your system for Java as discussed here.
2. Open your preferred text editor — this is the editor you set while installing the Java platform.

   For example, **Notepad** or **Notepad++** on Windows; **Gedit**, **Kate** or **SciTE** on Linux; or, **XCode** on Mac OS, etc.
3. Write the following lines of code in a new text document:

   Code listing 2.5: HelloWorld.java

   ```
   public class HelloWorld {
     public static void main(String[] args) {
       System.out.println("Hello World!");
     }
   }
   ```
4. Save the file as **HelloWorld.java** — the name of your file should be the same as the name of your class definition and followed by the `.java` extension. This name is *case-sensitive*, which means you need to capitalize the precise letters that were capitalized in the name for the class definition.
5. Next, open your preferred command-line application.

For example, **Command Prompt** on Windows; and, **Terminal** on Linux and Mac OS.

6. In your command-line application, navigate to the directory where you just created your file. If you do not know how to do this, consider reading through our crash courses for command-line applications for Windows or Linux.
7. Compile the Java source file using the following command which you can copy and paste in if you want:

   Compilation
   ```
   javac HelloWorld.java
   ```
8. Once the compiler returns to the prompt, run the application using the following command:

   Execution
   ```
   java HelloWorld
   ```
9. The above command should result in your command-line application displaying the following result:

   Output
   ```
   Hello World!
   ```
   Ask for help if the program did not execute properly in the Discussion page for this chapter.

Automatic Compilation of Dependent Classes

In Java, if you have used any reference to any other java object, then the class for that object will be automatically compiled, if that was not compiled already. These automatic compilations are nested, and this continues until all classes are compiled that are needed to run the program. So it is usually enough to compile only the high level class, since all the dependent classes will be automatically compiled.

Main class compilation
```
javac ... MainClass.java
```

However, you can't rely on this feature if your program is using reflection to create objects, or you are compiling for servlets or for a "jar", package. In these cases you should list these classes for explicit compilation.

Main class compilation
```
javac ... MainClass.java ServletOne.java ...
```

The JIT compiler

The Just-In-Time (JIT) compiler is the compiler that converts the byte-code to machine code. It compiles byte-code once and the compiled machine code is re-used again and again, to speed up execution. Early Java compilers compiled the byte-code to machine code each time it was used, but more modern compilers cache this machine code for reuse on the machine. Even then, java's JIT compiling was still faster than an "interpreter-language", where code is compiled from **high level language**, instead of from byte-code each time it was used.

The standard JIT compiler runs *on demand*. When a method is called repeatedly, the JIT compiler analyzes the bytecode and produces highly efficient machine code, which runs very fast. The JIT compiler is smart enough to recognize when the code has already been compiled, so as the application runs, compilation happens only as needed. As Java applications run, they tend to become faster and faster, because the JIT can perform runtime profiling and optimization to the code to meet the execution environment. Methods or code blocks which do not run often receive less optimization; those which run often (so called *hotspots*) receive more profiling and optimization.

Execution

There are various ways in which Java code can be executed. A complex Java application usually uses third party APIs or services. In this section we list the most popular ways a piece of Java code may be packed together and/or executed.

JSE code execution

Java language first edition came out in the client-server era. Thick clients were developed with rich GUI interfaces. Java first edition, JSE (Java Standard Edition) had/has the following in its belt:

- GUI capabilities (AWT, Swing)
- Network computing capabilities (RMI)
- Multi-tasking capabilities (Threads)

With JSE the following Java code executions are possible:

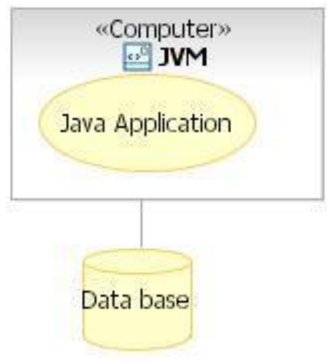


Figure 1: Stand alone execution
Stand alone Java application

(Figure 1) Stand alone application refers to a Java program where both the user interface and business modules are running on the same computer. The application may or may not use a database to persist data. The user interface could be either AWT or Swing.

The application would start with a `main()` method of a Class. The application stops when the `main()` method exits, or if an exception is thrown from the application to the JVM. Classes are loaded to memory and compiled as needed, either from the file system or from a *.jar file, by the JVM.
Invocation of Java programs distributed in this manner requires usage of the command line. Once the user has all the class files, he needs to launch the application by the following command line (where Main is the name of the class containing the main() method.)

Execution of class

```
java Main
```

Java 'jar' class libraries

Utility classes, framework classes, and/or third party classes are usually packaged and distributed in Java ' *.jar' files. These 'jar' files need to be put in the CLASSPATH of the java program from which these classes are going to be used.
If a jar file is executable, it can be run from the command line:

Execution of archive

```
java -jar Application.jar
```

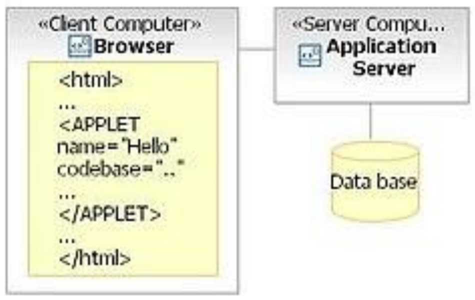


Figure 2: Applet Execution

Java Applet code

(Figure 2) Java Applets are Java code referenced from HTML pages, by the <APPLET> tag. The Java code is downloaded from a server and runs in the client browser JVM. Java has built-in support to render applets in the browser window.
Sophisticated GUI clients were found hard to develop, mostly because of download time, incompatibilities between browser JVM implementations, and communication requirements back to the server. Applets are rarely used today, and are most commonly used as small, separate graphic-like animation applets. The popularity of Java declined when Microsoft withdrew its Java support from Internet Explorer default configuration, however, the plugin is still available as a free download from java.com.
More information can be found about applets at the Applet Chapter, in this book. Also, Wikipedia has an article about Java Applets.

Client Server applications

The client server applications consist of a front-end, and a back-end part, each running on a separate computer. The idea is that the business logic would be on the back-end part of the program, which would be reused by all the clients. Here the challenge is to achieve a separation between front-end user interface code, and the back-end business logic code.
The communication between the front-end and the back-end can be achieved by two

ways.

- One way is to define a data communication protocol between the two tiers. The back-end part would listen for an incoming request. Based on the protocol it interprets the request and sends back the result in data form.
- The other way is to use Java Remote Invocation (RMI). With the use of RMI, a remote object can be created and used by the client. In this case Java objects are transmitted across the network.

More information can be found about client-server programming, with sample code, at the Client Server Chapter in this book.

Web Applications

For applications needed by lots of client installations, the client-server model did not work. Maintaining and upgrading the hundreds or thousands of clients caused a problem. It was not practical. The solution to this problem was to create a unified, standard client, for all applications, and that is the *Browser*.
Having a standard client, it makes sense to create a unified, standard back-end service as well, and that is the *Application Server*.
Web Application is an application that is running in the Application Server, and it can be accessed and used by the Browser client.
There are three main area of interest in Web Applications, those are:

- The Web Browser. This is the container of rendering HTML text, and running client scripts
- The HTTP protocol. Text data are sent back and forth between Browser and the Server
- The Web server to serve static content, Application server to serve dynamic content and host EJBs.

Lexical structure

Computer languages, like human languages, have a lexical structure. A source code of a Java program consists of tokens. Tokens are atomic code elements. In Java we have comments, identifiers, literals, operators, separators, and keywords.

Java programs are composed of characters from the Unicode character set.

Comments

Comments are used by humans to clarify source code. There are three types of comments in Java.

Comment type	Meaning
// comment	Single-line comments

/* comment */	Multi-line comments
/** documentation */	Documentation comments

If we want to add some small comment we can use single-line comments. For more complicated explanations, we can use multi-line comments. The documentation comments are used to prepare automatically generated documentation. This is generated with the `javadoc` tool.

Comments.java

```
package com.zetcode;

/*
  This is Comments.java
  Author: Jan Bodnar
  ZetCode 2017
*/

public class Comments {

    // Program starts here
    public static void main(String[] args) {

        System.out.println("This is Comments.java");
    }
}
```

The program uses two types of comments.

```
// Program starts here
```

This is an example of a single-line comment.

Comments are ignored by the Java compiler.

```
/*
  This is Comments.java
/*  Author: Jan Bodnar */
  ZetCode 2017
*/
```

Comments cannot be nested. The above code does not compile.

White space

White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code.

```
int i = 0;
```

White spaces are required in some places. For example between the `int` keyword and the variable name. In other places, white spaces are forbidden. They cannot be present in variable identifiers or language keywords.

```
int a=1;
int b = 2;
int c  =  3;
```

The amount of space put between tokens is irrelevant for the Java compiler. The white space should be used consistently in Java source code.

Identifiers

Identifiers are names for variables, methods, classes, or parameters. Identifiers can have alphanumerical characters, underscores and dollar signs ($). It is an error to begin a variable name with a number. White space in names is not permitted.

Identifiers are case sensitive. This means that `Name`, `name`, or `NAME` refer to three different variables. Identifiers also cannot match language keywords.

There are also conventions related to naming of identifiers. The names should be descriptive. We should not use cryptic names for our identifiers. If the name consists of multiple words, each subsequent word is capitalized.

```
String name23;
int _col;
short car_age;
```

These are valid Java identifiers.

```
String 23name;
int %col;
short car age;
```

These are invalid Java identifiers.

The following program demonstrates that the variable names are case sensitive. Event though the language permits this, it is not a recommended practice to do.

CaseSensitiveIdentifiers.java

```
package com.zetcode;

public class CaseSensitiveIdentifiers {

    public static void main(String[] args) {

        String name = "Robert";
        String Name = "Julia";

        System.out.println(name);
        System.out.println(Name);
    }
}
```

`Name` and `name` are two different identifiers. In Visual Basic, this would not be possible. In this language, variable names are not case sensitive.

```
$ java com.zetcode.CaseSensitiveIdentifiers
Robert
Julia
```

Literals

A *literal* is a textual representation of a particular value of a type. Literal types include boolean, integer, floating point, string, null, or character. Technically, a literal will be assigned a value at compile time, while a variable will be assigned at runtime.

```
int age = 29;
String nationality = "Hungarian";
```

Here we assign two literals to variables. Number 29 and string "Hungarian" are literals.

Literals.java

```
package com.zetcode;

public class Literals {

    public static void main(String[] args) {

        int age = 23;
        String name = "James";
        boolean sng = true;
        String job = null;
        double weight = 68.5;
        char c = 'J';

        System.out.format("His name is %s%n", name);
        System.out.format("His is %d years old%n", age);

        if (sng) {

            System.out.println("He is single");
        } else {

            System.out.println("He is in a relationship");
        }

        System.out.format("His job is %s%n", job);
        System.out.format("He weighs %f kilograms%n", weight);
        System.out.format("His name begins with %c%n", c);
    }
}
```

In the above example, we have several literal values. 23 is an integer literal. "James" is a string literal. The `true` is a boolean literal. The `null` is a literal that represents a missing value. 68.5 is a floating point literal. 'J' is a character literal.

```
$ java com.zetcode.Literals
His name is James
His is 23 years old
He is single
His job is null
He weighs 68.500000 kilograms
His name begins with J
```

This is the output of the program.

Operators

An *operator* is a symbol used to perform an action on some value. Operators are used in expressions to describe operations involving one or more operands.

```
+    -    *    /    %    ^    &    |    !    ~
=    +=   -=   *=   /=   %=    ^=    ++    --
==   !=    <   >    &=  >>=   <<=   >=   <=
||   &&    >>    <<    ?:
```

This is a partial list of Java operators. We will talk about operators later in the tutorial.

Separators

A *separator* is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data stream.

```
[ ]   ( )   { }   ,   ;   .   "
String language = "Java";
```

The double quotes are used to mark the beginning and the end of a string. The semicolon `;` character is used to end each Java statement.

```
System.out.println("Java language");
```

Parentheses (round brackets) always follow a method name. Between the parentheses we declare the input parameters. The parentheses are present even if the method does not take any parameters. The `System.out.println()` method takes one parameter, a string value. The dot character separates the class name (`System`) from the member (`out`) and the member from the method name (`println()`).

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
```

The square brackets `[]` are used to denote an array type. They are also used to access or modify array elements. The curly brackets `{}` are used to initiate arrays. The curly brackets are also used enclose the body of a method or a class.

```
int a, b, c;
```

The comma character separates variables in a single declaration.

Keywords

A keyword is a reserved word in Java language. Keywords are used to perform a specific task in the computer program. For example, to define variables, do repetitive tasks or perform logical operations.

Java is rich in keywords. Many of them will be explained in this tutorial.

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

In the following small program, we use several Java keywords.

Keywords.java
```
package com.zetcode;

public class Keywords {

    public static void main(String[] args) {

        for (int i = 0; i <= 5; i++) {

            System.out.println(i);
        }
    }
}
```

The `package`, `public`, `class`, `static`, `void`, `int`, `for` tokens are Java keywords.

Conventions

Conventions are best practices followed by programmers when writing source code. Each language can have its own set of conventions. Conventions are not strict rules; they are merely recommendations for writing good quality code. We mention a few conventions that are recognized by Java programmers. (And often by other programmers too).

- Class names begin with an uppercase letter.
- Method names begin with a lowercase letter.
- The public keyword precedes the static keyword when both are used.
- The parameter name of the `main()` method is called args.
- Constants are written in uppercase.
- Each subsequent word in an identifier name begins with a capital letter.

In this part of the Java tutorial, we covered some basic lexis for the Java language.