



Java + Command Line Interface

Nadzeya Chybisava

- 15 months in EPAM
- 1 project as a developer
- Java Technologies Active Contributor



Command line overview

- **Advantages**

- We can provide any number of arguments using a command line argument.
- Information is passed as Strings. So we can convert it to numeric or another format easily.
- While launching our application it is useful for configuration information



Command line JDK installation check

- Open a command prompt and type:

```
%JAVA_HOME%\bin\javac -version
```

- If *JAVA_HOME* points to a JDK, the output should look like:

```
javac 11.0.11
```

- If *JAVA_HOME* doesn't point to a JDK, the OS will throw an error message and you should configure variables.

```
The system cannot find the path specified
```

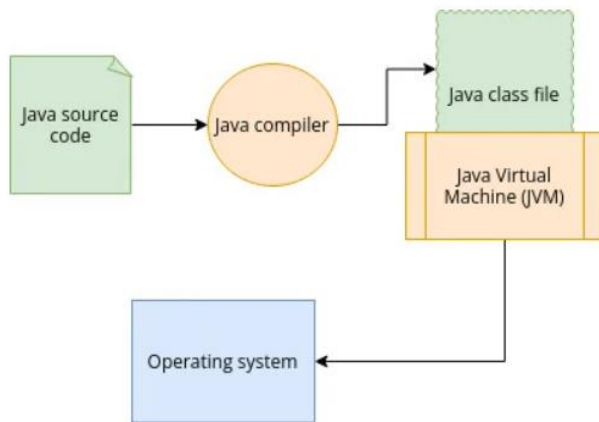


Simple application

To compile the program, you would type the following at the command line:

```
javac Hello.java
```

The Java compiler is named **javac**. The **javac** program is unique in that it does not produce actual machine code. Instead, it produces something called bytecode.



Simple application

To run this bytecode, the Java interpreter is invoked in the following way.

```
java Hello
```

Note the name of the Java interpreter is **java**. Also note that you do not include the **.class** at the end of the filename when invoking the interpreter



Application with parameters

The command-line arguments in Java allow the programmers to pass the arguments during the execution of a program. These arguments get stored as Strings in a String array `args[]` that is passed to the `main()` function.

```
class ExampleArgs {  
    public static void main(String[] args) {  
        // handle arguments  
    }  
}
```

However, Java 5 introduced varargs, which are arrays in sheep's clothing. Therefore, we can define our *main* with a *String* vararg.

```
class ExampleArgs {  
    public static void main(String... args) {  
        // handle arguments  
    }  
}
```



Application with parameters

We need to pass the arguments as space-separated values.

```
java CommandLine a b c
```

If an argument needs to contain spaces, the spaces must be enclosed between quotes.

```
java CommandLine a " b" c
```

We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments.

```
java CommandLine 10 20
```



Console input

`System.in` is an `InputStream` which is typically connected to keyboard input of console programs.

In other words, if you start a Java application from the command line, and you type something on the keyboard while the CLI console (or terminal) has focus, the keyboard input can typically be read via `System.in` from inside that Java application.

However, it is only keyboard input directed to that Java application (the console / terminal that started the application) which can be read via `System.in`. Keyboard input for other applications cannot be read via `System.in`.



Java options

All JVM implementations support standard options. Run the `'java'` command in a terminal to see a list of standard options.

Non-standard options start with `-X`. These are for general purpose use and are specific to a particular implementation of JVM.

Advanced options start with `-XX`.

You use the `-Xlog` option to configure or enable logging with the Java Virtual Machine (JVM) unified logging framework.



Java options

Option	Description
<code>-class-path</code> <code>-classpath</code> <code>-cp</code>	<p>A semicolon (;) separated list of directories, JAR archives, and ZIP archives to search for class files.</p> <p>Specifying <i>classpath</i> overrides any setting of the CLASSPATH environment variable. If the class path option isn't used and <i>classpath</i> isn't set, then the user class path consists of the current directory (.).</p>
<code>-Dproperty=value</code>	<p>Sets a system property value. The <i>property</i> variable is a string with no spaces that represents the name of the property. The <i>value</i> variable is a string that represents the value of the property. If <i>value</i> is a string with spaces, then enclose it in quotation marks (for example <code>-Dfoo="foo bar"</code>).</p>
<code>-version</code>	<p>Prints product version to the output stream and exits.</p>
<code>-X</code>	<p>Prints the help on extra options to the error stream</p>
<code>-Xms size</code>	<p>Sets the minimum and initial size (in bytes) of the heap. This value must be a multiple of 1024 and greater than 1 MB.</p> <p>-XX:NewSize to set the initial size -XX:MaxNewSize to set the maximum size</p>
<code>-Xmx size</code>	<p>Specifies the maximum size (in bytes) of the heap. This value must be a multiple of 1024 and greater than 2 MB.</p> <p>-XX:MinHeapSize to set the minimum size -XX:InitialHeapSize to set the initial size</p>



Multiple Java Source Files

Compile several source files at once, type:

```
javac Person.java PersonIntroduction.java
```

Compile all source files whose filenames start with Person:

```
javac Person*.java
```

Compile all source files:

```
javac *.java
```

Start the class with main() as usual with args:

```
java PersonIntroduction Igor
```



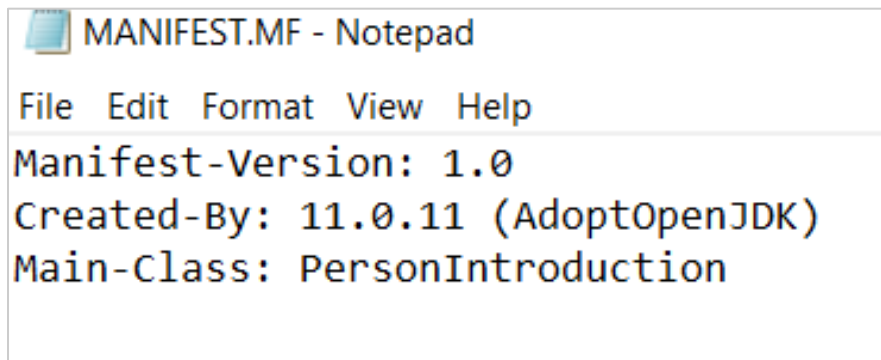
JAR

- A [JAR \(Java Archive\)](#) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.
- A [JAR file](#) can contain one or more main classes. **Each main class is the entry point of an application.** So, a JAR file can theoretically contain more than one application, but it has to contain at least one main class to be able to run.



Manifest

- The manifest file is named *MANIFEST.MF* and is located under the *META-INF* directory in the JAR. It's simply **a list of key and value pairs, called *headers* or *attributes*, grouped into sections.**
- These *headers* supply metadata that help us describe aspects of our JAR such as the versions of packages, what application class to execute, the classpath, signature material and much more.
- A manifest file is added automatically whenever we [create a JAR](#) or, we can specify our own manifest file. **A valid header must have a space between the colon and the value.** Another important point is there **must be a new line at the end of the file.** Otherwise, the last header is ignored.



```
MANIFEST.MF - Notepad
File Edit Format View Help
Manifest-Version: 1.0
Created-By: 11.0.11 (AdoptOpenJDK)
Main-Class: PersonIntroduction
```



Non Executable JAR

- A nonexecutable JAR is simply a JAR file that doesn't have a *Main-Class* defined in the manifest file.
 - `jar cf nonExecutable.jar *.class`
- To run an application in a nonexecutable JAR file, we have to use `-cp` option instead of `-jar` to specify the JAR file that contains the class file we want to execute
 - `java -cp nonExecutable.jar PersonIntroduction Anna`



Executable JAR

- An executable JAR is simply a JAR file that has a *Main-Class* defined in the manifest file.
 - `jar cfe executable.jar PersonIntroduction *.class`
- So, when invoking an executable JAR, we don't need to specify the main class name on the command line. We simply add our arguments after the JAR file name.

```
java -jar executable.jar Nansy
```



Option	Description
-c	Create the archive
-t, --list	List the table of contents for the archive
-u, --update	Update an existing jar archive
-f, --file=FILE	The archive file name.
-e, --main-class=CLASSNAME	The application entry point for stand-alone applications bundled into a modular, or executable, jar archive
-m, --manifest=FILE	Include the manifest information from the given manifest file
-M, --no-manifest	Do not create a manifest file for the entries



Do you have any questions?

