“Hello, World!” 1

1.1 The Programming Environment for Java

1.1.1 The Java Virtual Machine (JVM)

Java is one of the most popular programming languages. It is a descendant of the programming language **C** and is much related to **C++**. Java, like C++, embodies the concept of *object-oriented programming*, which allows a programmer to define a type of data with its permissible set of operations.

To execute a Java program on a machine, the machine needs an installation of the Java Running Environment (JRE). A major part of JRE is the Java Virtual Machine (JVM). JVM creates an environment where Java programs interact with the hardware.

A programmer creates a Java program by writing its source code. A source code is a text file that describes the program according to the syntax of Java and has the file name extension `.java`. An executable Java program generated from a source code is called `Java bytecode`, and has the file name extension `.class`. To generate a Java program from a source code, the machine needs an extension of JRE called the Java Development Kit (JDK) (see Fig. 1.1).

The Java language that comes with JDK consists of two parts. The first part, called `java.lang`, is an essential component of Java. The second part is a collection of source codes that can be selected and added to the Java program being written.

To write and edit a Java source code, a text editor is needed. Some editors understand the syntax of Java and offer a variety of assistance. Popular text editors, which can be used for editing general text, include: vim, emacs, and sublime.

The process of generating Java bytecode from a Java source code is called compilation. A primitive way to compile a Java source code is to execute a compilation command in a command line interface. Modern computers usually come with command line interfaces. For example, Mac OSX has **Terminal** and Windows has **cmd**. There are other downloadable command line interfaces. A command line interface is a program that interacts with the user on a text screen (see Fig. 1.2). The user types, according to the syntax of the command line interface, a character sequence representing an action he/she wishes to take. This sequence is called a command or a command line. On the interface screen, the characters that the user types appear (or “echo”) as they are being typed. After completing the command, the user hits the return key. The command line interface then attempts to parse the entered

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_1

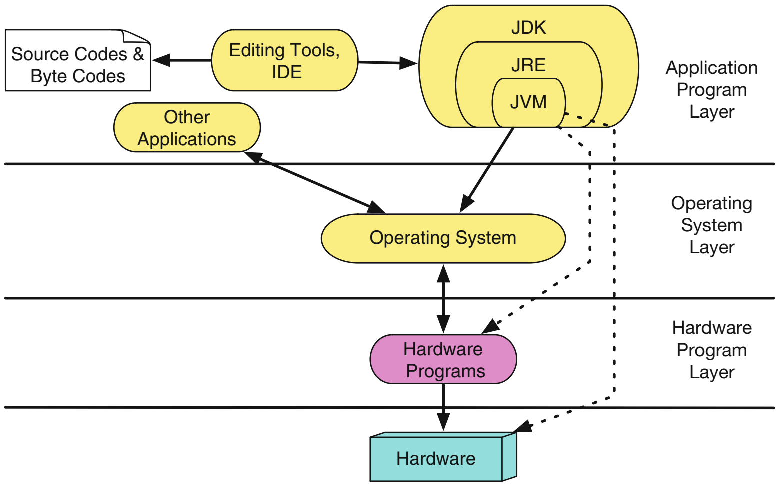
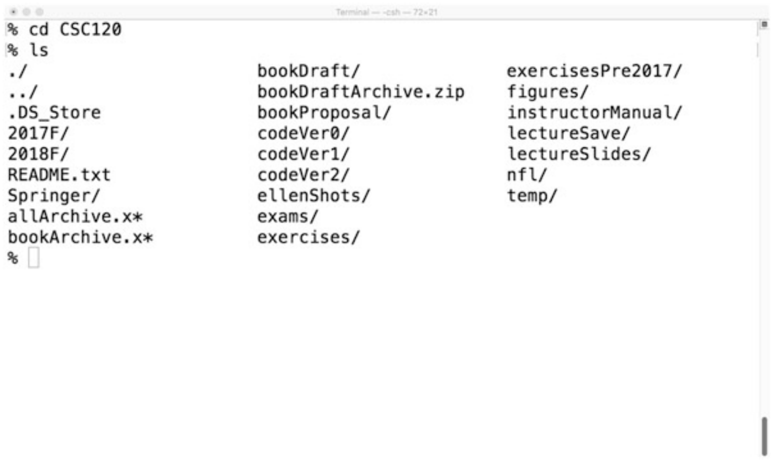


Fig. 1.1 The program layers, JVM, JRE, and JDK



```
% cd CSC120
% ls
./                     bookDraft/             exercisesPre2017/
../                    bookDraftArchive.zip   figures/
.DS_Store              bookProposal/          instructorManual/
2017F/                 codeVer0/              lectureSave/
2018F/                 codeVer1/              lectureSlides/
README.txt             codeVer2/              nfl/
Springer/              ellenShots/            temp/
allArchive.x*          exams/
bookArchive.x*         exercises/
%
```

Fig. 1.2 A screen shot of Terminal on a Mac OS X machine. The prompt is the percent sign followed by one white space

command according to its syntax. If the command is syntactically correct, the interface executes the action represented by the command. When the action occurs, some characters may appear on the screen after the command line. If there are many such characters, the text screen scrolls down to show however many characters will fit on the screen, starting with the most recent. If the command is not syntactically correct, the command line interface prints an error message. To inform the user that it is ready to accept a new command, the command line interface prints a special sequence of characters, called prompt, e.g., the "percent" symbol `%` followed by one white space, or the "greater than" symbol >. Table 1.1 is a select list of commands.

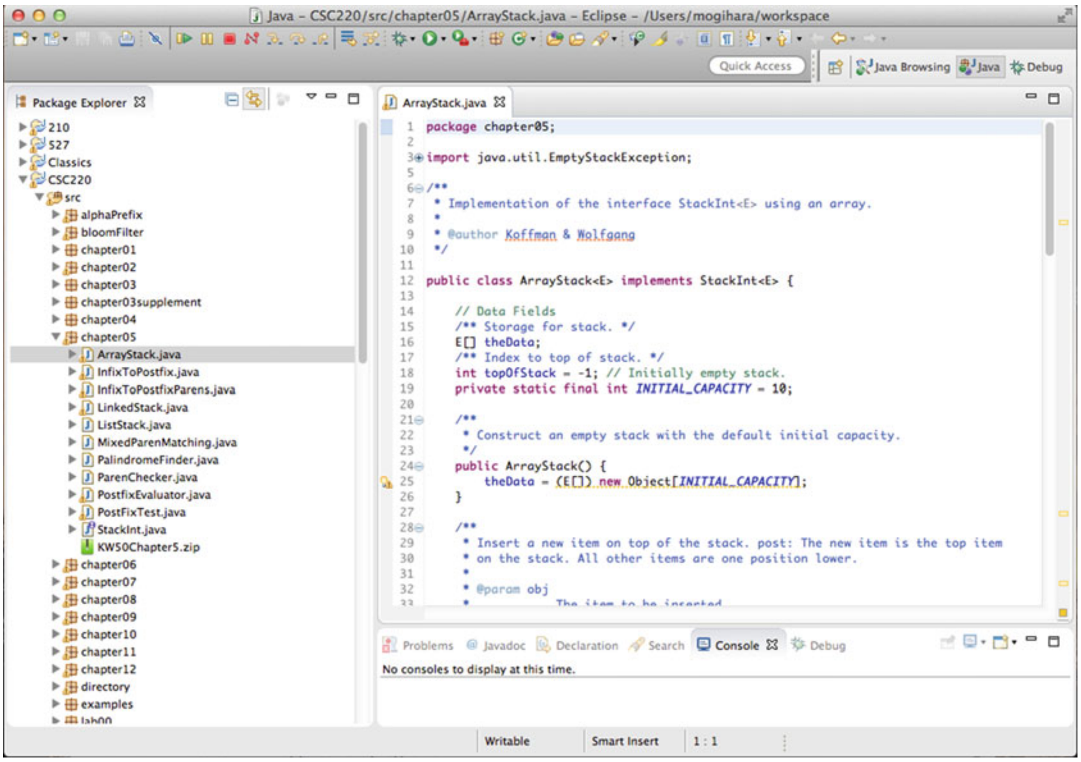
An interactive development environment (IDE) is a program that combines a program editor and a command line interface with many additional features that make it easy to write (in particular, large) computer programs (see Fig. 1.3). The most famous IDEs for Java are Eclipse and Netbeans.

1.1.2 Changing Folders in a Command Line Interface

A command line interface (such as the program Finder for Mac and the program Folder for Windows) works in one specific file folder. The specific folder is called the **working folder** (or the **working directory**). In Finder and Folder, switching to a different folder is done by clicking icons. In a

Table 1.1 A short list of commands available in the Terminal programs for Mac OSX and Linux, as well as their counterparts in the Windows cmd

Terminal	cmd	Function
(Mac OSX/Linux)	(Windows)	
`cd FOLDER`	`cd FOLDER`	Change folder to FOLDER
`cd`	`cd`	Change to the home folder
`cd ..`	`cd ..`	Change to the parent folder
`pwd`	`chdir`	Print the working folder
`ls`	`dir`	Listing of all files in the folder
`ls FILE`	`ls FILE`	Listing of FILE in the folder
`rm FILE`	`del FILE`	Remove the file FILE
`rmdir FOLDER`	`del FOLDER`	Remove the folder FOLDER (must be empty)
`mv FILE1 FILE2`	`move FILE1 FILE2`	Rename FILE1 to FILE2
`cp FILE1 FILE2`	`copy FILE1 FILE2`	Copy FILE1 to FILE2
`mkdir FOLDER`	`mkdir FOLDER`	Create the folder FOLDER
`cat FILE`	`type FILE`	Show the contents of FILE
`more FILE`	`more FILE`	Show the contents of FILE in chunks



```
package chapter05;

import java.util.EmptyStackException;

/**
 * Implementation of the interface StackInt<E> using an array.
 *
 * @author Koffman & Wolfgang
 */

public class ArrayStack<E> implements StackInt<E> {

    // Data Fields
    /** Storage for stack. */
    E[] theData;
    /** Index to top of stack. */
    int topOfStack = -1; // Initially empty stack.
    private static final int INITIAL_CAPACITY = 10;

    /**
     * Construct an empty stack with the default initial capacity.
     */
    public ArrayStack() {
        theData = (E[]) new Object[INITIAL_CAPACITY];
    }

    /**
     * Insert a new item on top of the stack. post: The new item is the top item
     * on the stack. All other items are one position lower.
     *
     * @param obj
     *            The item to be inserted
```

No consoles to display at this time.

Fig. 1.3 An IDE screen of Eclipse

command line interface, this switch is made by typing a special command. The name of the command is `cd` (which stands for "change directory"). The syntax of `cd` is:

```
cd FOLDER_PATH
```

Here, FOLDER_PATH specifies how to get to the target folder from the working folder. In Chap. 15, we study the general concept of file paths.

The folder moves are composed of two basic moves: moving to one of the folders that belong to the present folder or moving to the parent (that is, the folder that the working folder belongs to as a subfolder). To move to a child folder, the path that is specified is this child folder's name. To move to the parent, the path that is specified is `..` (i.e., two periods).

It is possible to combine successive folder moves into one path. For a non-Windows system like Mac OSX and Linux, this is done by inserting / between the individual moves. For Windows, \ is the folder separator. For example, in Mac OSX,

```
cd ../foo/bar
```

changes the working folder to the parent, to a child of the parent named `foo`, and then to a child of the child named `bar`. `bar` is thus a grandchild folder of the parent of the present working folder.

To check what the working folder directly is, the command `pwd` can be used for a non-Windows system, and `chdir` for Windows. These commands print the path to the working folder, starting from the "home" folder on the screen. It is also possible to take the inventory of the files in the working folder using the command `ls` for a non-Windows system, and `dir` for Windows. This command produces a list of all the files in the working folder.

1.1.3 Source Codes, Bytecodes, and Compilation

As mentioned earlier, all Java source files must have the file name extension `.java`. We call them `Java` files. To compile the source code `Foo.java` in a command line interface, one must type the command:

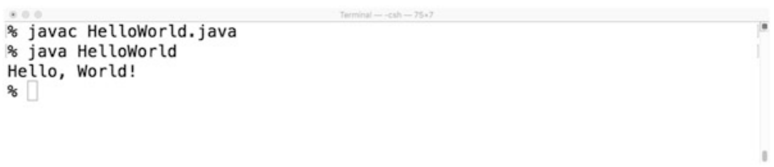
```
javac Foo.java
```

If the source code compiles without error, a file named `Foo.class` is created. This is the bytecode of the program. We call it a **class file**. If the class file already exists, then the file is overwritten upon successful compilation. In the case of IDE, `.class` files are generated only during the execution process. If `Foo.java` has compiled successfully, the program can be executed by using the command:

```
java Foo
```

after the prompt.

Here is an example of a user interaction with a command line interface, where the user tries to compile and then execute a program whose source code is `HelloWorld.java` (Fig. 1.4). The prompt of the command line interface is the percent symbol followed by one white space. The first line is the compilation command, the second line is the execution command, the third line is the result of executing the code, and the fourth line is the command prompt after the execution.



```
% javac HelloWorld.java
% java HelloWorld
Hello, World!
% 
```

Fig. 1.4 The compilation and execution of `HelloWorld.java`

1.2 The First Program, "Hello, World!"

Let us start writing Java programs. Our first program is the popular `Hello, World!` program.

```java
public class HelloWorld
{
  public static void main( String[] args )
  {
    System.out.println( "Hello, World!" );
  }
}
```

Listing 1.1 `HelloWorld.java`

Executing this program produces a single line of output:

```
Hello, World!
```

Using this code, we shall learn some important ingredients of Java programs.

The three words in the first line:

`public class HelloWorld`

state that:

(a) this is a program unit of type `class`,
(b) the unit is named `HelloWorld`, and
(c) the unit is accessible from all other Java program units.

There are four kinds of program units: `class`, `interface` `abstract class`, and `enum`. This book covers `class` and `interface` only.

The keyword `public` specifies that the unit is accessible from other program units. A keyword that specifies the accessibility of program units and their components is called a **visibility attribute**. There are three explicit visibility types: `public`, `private`, and `protected`. The default visibility type, when there is no explicit visibility attribute, is the `package visibility`.

The order of appearance of the three components,

`VISIBILITY_TYPE UNIT_TYPE NAME`

applies to all Java source code files.

Every source file in Java has the file extension `.java`. In a single `.java` file, multiple classes can be defined, simply by concatenating their source codes. In such a case, at most one class may be `public`. In addition, if a source file contains a unit with the public visibility, the name of the unit must match the file name. For example,

```java
class Foo
{
...
}
class Bar
{
...
}
```

is okay, but

```
public class Foo
{
...
}
public class Bar
{
...
}
```

is not.

```
public class Foo
{
...
}
class Bar
{
...
}
```

is acceptable as well, but the source file must be `Foo.java`.

Let us do an experiment. If the class name is changed from `HelloWorld` to `Hello` while preserving the file name `HelloWorld.java`:

```
public class Hello
{
  public static void main( String[] args )
  {
    System.out.println( "Hello, World!" );
  }
}
```

attempting to compile the source code (that is, the command `javac HelloWorld.java`) produces the following error message:

```
HelloWorld.java:1: error: class Hello is public, should be
declared in a file named Hello.java
public class Hello
{
       ^
1 error
```

The first two lines of the error message state the nature of the error, which is that, to define a class by the name of `Hello`, the file name must be `Hello.java`. The next three lines of the error message specify the location of the error using the "caret" symbol ^. According to the marking, the error is at the class declaration. The last line of the error message is the total number of errors found in the source code.

If the source file name `HelloWorld.java` is retained but the `public` attribute is removed, like this one:

```java
class Hello
{
  public static void main( String[] args )
  {
    System.out.println( "Hello, World!" );
  }
}
```

the code compiles, but the `.class` generated is `Hello.class`, not `HelloWorld.class`.

1.2.1 Methods and Their Declarations

In Java, curly brackets { } are used to enclose units, components, and code blocks. For example, the declaration of each program unit (such as class and interface) should be followed by a matching pair of curly bracket. Two matching pairs appear either one after the other or one inside the other; that is,

either `{ ... { ... } ... }` or `{ ... } ... { ... }`

For a source file to compile successfully, all of its curly brackets must have unique matching partners. Thus, the **depth** of a text in a source code can be defined as the number of matching pairs enclosing it. Multiple methods may appear in one source code.

In the source code of `HelloWorld.java`, the opening line `public class HelloWorld` is at depth 0, `public static void main( String[] args )` is at depth 1, and `System.out.println( ... )` is at depth 2. The component appearing between lines 3 and 6 is called a **method**.

A **method** has a name and defines a set of actions needs to be performed. Some methods process information given to the in the form of **parameters**. Some methods report the results through the use of **return values**. We cover this topic in Chap. 5.

A special kind of method is the **method `main`**. Each method `main` takes the form of:

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

as its declaration. The term `args` appearing in the parentheses represents the sequence of characters that the user types in the command line after the name of the program. We study `args` in Sect. 13.4. Only Java classes with a method `main` can be executed.

The general method declaration consists of the following, where the parts in boxes are optional.

$\boxed{\texttt{VISIBILITY}}$	$\boxed{\texttt{static}}$	`RETURN_TYPE`	`NAME`	`(` $\boxed{\texttt{PARAMETERS}}$ `)`	$\boxed{\texttt{THROWS}}$
Optional	Required for Static Methods	Required	Required	Required	Optional

In the case of the method `main`, the attribute `public` states that the method is accessible from outside, the attribute `static` states that the method is part of some executable program, the return type `void` states that the method has no return value, and `String[] args` specifies that the parameter is `args` and its data type is `String[]`. We study the meaning of square brackets in Chap. 12. The last component is about handling errors that occur during the execution of the program. We study this in Chap. 15.

1.2.2 `System.out.println` and `System.out.print`

The method `main` has only one action:

```
System.out.println( "Hello, World!" );
```

This produces the output of `Hello, World!`. A sequence of characters that defines one unit of action is called a **statement**.

Generally, a statement ends with a semicolon. The role of a statement is to perform a certain task. A method can contain any number of statements, including none. The statements are executed in the order they appear.

The period plays a special role in Java; it implies possession. `System.out.println` refers to a method (by the name of `println`) associated with `System.out`, which is part of a group of virtual hardware components named `System`. The siblings of `System.out` include: `System.err`, for printing error messages, and `System.in`, for keyboard input.

The method `println` converts the data appearing inside its parentheses to a series of characters, and then prints the series on the screen with the newline character (the equivalent of the return key) at the end. The sequence `"Hello, World!"` is the thirteen character sequence:

```
'H' 'e' 'l' 'l' 'o' ',' ' ' 'W' 'o' 'r' 'l' 'd' '!'
```

The double quotation mark `"` that surrounds the thirteen-character sequence is for marking the start and the end of the sequence. A character sequence encased in a pair of double quotation marks is called a **String literal**.

The method `System.out.println` automatically prints the newline character at the end. Sometimes the attachment of the newline is not desirable, e.g., when a single line of output is built by combining multiple outputs. The method `System.out.print`, a sibling of `System.out.println`, is helpful in such an occasion. The method `System.out.print` does the same as `System.out.println`, except that it does not append the newline character at the end. Furthermore, `System.out.println()` has the effect of typing the return key (that is, going to the next line without printing any other character), while `System.out.print()` is syntactically incorrect because `System.out.print()` means "print nothing".

If the method call is changed from `System.out.println` to `System.out.print`, how will the output change? Here is the new code. The program has a new name `HelloWorld01`.

```java
public class HelloWorld01
{
  public static void main( String[] args )
  {
     System.out.print( "Hello, World!" );
  }
}
```

Listing 1.2 A version of `HelloWorld` that uses `System.out.print` in place of `System.out.println`

The execution produces the following:

```
Hello, World!%
```

Listing 1.3 The result of executing `HelloWorld.java`

Note that the prompt `%` appears at the end line because of the use of `System.out.print`.

1.2.3 Spacing in the Source Code

In Java source code, the white space, the tab-stop, and the newline all serve as spacing characters. The following spacing rules must be followed in Java:

- There should be some spacing between two consecutive occurrences of any of the following: type, attribute, and name.
- Some mathematical and logical symbols run together to mean special operations. In this situation, there should not be spacing between symbols.
- There should not be any newlines appearing inside a `String` literal (a character sequence within a pair of double quotation marks).

Proper spacing makes Java source codes easy to read. For example, it is possible to write:

```
System   .   out   .   println    (    "Hello, World!"
    )
    ;
```

instead of the plain

```
System.out.println("Hello, World!");
```

Although the Java compiler finds no problem in understanding this line of code, a human may find it to be a bit difficult to parse.

It is advisable to limit the number of characters per line and work within that limit, breaking up long lines into multiple smaller ones. The spacing, indentation, and the line width are up to the programmer.

Furthermore, indenting from the left end proportionally to the depth of code is good practice (recall the discussion in Sect. 1.2.1). In other words, using some fixed quantity M, the code at depth D receives an indentation of $M \cdot D$ white spaces. The code in this book utilizes this scheme with $M = 2$.[1] Most code examples that appear in this book present each curly bracket as a stand-alone in one line.

1.2.4 Commenting

It is possible to insert texts that have no relevance to how the code runs. Such texts are called **comments**. Comments are free-form texts. Java compilers ignore comments when producing class files and so they exist only in the source file. A programmer can use comments to make notes to him or herself. For example, comments can be about the expected behavior of the program and about the underlying algorithms. To save space, the code examples presented in this book use comments sparingly.

Java has three kinds of comment formats.

The first kind starts with two forward slashes `//` appearing outside `String` literals. If two side-by-side forward slashes appear outside multiple-line comments (see below) and outside `String` literals, the two characters and all the characters appearing after them are treated as comments.

[1]Some people use $M = 4$ or $M = 8$. The latter is equivalent to the tab-stop; i.e., a tab-stop with a depth of 1.

Because a programmer tends to limit the number of characters per line in the source code, the comments that start with two forward slashes are naturally suitable for short comments. For example, in

```java
    System.out.println( "Hello!" ); // first line
    System.out.println( "How are you!" ); // second line
```

`// first line` and `// second line` are comments.

To place longer comments, multiple consecutive lines starting with two forward slashes after some indentation can be used, e.g.,

```java
    ////////////////////////////////////////////
    // This program receives two numbers from
    // the user and prints the result of performing
    // addition, subtraction, multiplication, and
    // subtraction on the two.
    ////////////////////////////////////////////
```

are long comment lines.

There is a special way of specifying multiple-line comments. If a line starts with `/*` after an indentation, then all the characters starting from the `/*` and ending with the next `*/` are comments. Using this option, a comment can be stated as:

```java
    /*
     * This program receives two numbers from
     * the user and prints the result of performing
     * addition, subtraction, multiplication, and
     * subtraction on the two.
     */
```

The `*` appearing in the middle four lines are extraneous, but programmers tend to put that character to make the vertical array of `*` look like the left border of the comments.

Be mindful of the following:

- `/*` appearing inside a matching pair of double quotation marks behaves as part of the `String` literal. The same holds for `*/` and `//`.
- All characters appearing between `/*` and the matching `*/` are comments. Therefore, `/*` appearing in a matching pair of `/*` and `*/` is part of the comment represented by the pair.

This means that the code

```java
1  public class Foo
2  {
3    public static void main( String[] args )
4    {
5      /* here is a comment
6      /* one more comment? */
7      */
8      System.out.println( "/*//" );
9    }
10 }
```

has no matching `/*` for the `*/` in Line 7.

The last kind of comment is the **Javadoc**. The Javadoc is a variant of the aforementioned multiple-line comment and uses a pair of `/**` and `*/` in place of `/*` and `*/`. Javadocs are for publicizing information about methods and classes and are written using a special syntax. IDEs such as Eclipse are capable of presenting information available through Javadocs.

The following code shows some examples of commenting.

```
 1  /*
 2   * Class for showing comment examples
 3   * Written by Mitsunori Ogihara
 4   */
 5  public class Comments
 6  {
 7    /**
 8     * main method
 9     * @param args the arguments
10     */
11    public static void main( String[] args )
12    {
13      // There are two lines in the program
14      System.out.println( "A code needs comments!" );
15    }
16  }
```

Listing 1.4 Examples of comments. Lines 1–4 form a multiple-line comment. Lines 7–10 form a Javadoc comment. Line 13 is a single-line comment

1.2.5 Errors

A **syntax error** is a part of source code that fails to conform to the Java syntax. If a source code contains syntax errors, the Java compiler, instead of producing the bytecode, produces an error message stating that there are syntax errors. If there is a bytecode generated from the prior successful compilation, that code remains the same without being updated.

The syntax error that a compiler produces is a bit cryptic and takes some experience to comprehend. Mainly because the compiler is not aware of the true intension of the programmer who wrote the erroneous code, the stated number of syntax errors does not necessarily agree with the actual number of syntax errors.

Consider the following code, which is intended to execute three `println` statements successively.

```
//---- This is the class name
public class BuggyHelloWorld
  /----
  //---- Main method of the code
  //----
  public static void main( String[] args )
  {
    System.out.pritnln( "Hello, World! );
    System.out.printin( Hello, Class!" );
    System.out.printin( "Hello, its' me!" ):
  }
}
}
```

Listing 1.5 A buggy version of the HelloWorld program. The intended class name is `BuggyHelloWorld`

There are four syntax errors in the code:

1. the forward slash in line 3 should be a double forward slash,
2. the `String` literal in line 7 does not have a closing double quotation mark,
3. the colon at the end of line 7 should be a semicolon, and
4. There should be one more "}" at the end.

At the compilation step, we encounter the following error messages:

```
BuggyHelloWorld.java:2: error: '{' expected
public class BuggyHelloWorld
                            ^
BuggyHelloWorld.java:8: error: unclosed string literal
    System.out.pritnln( "Hello, World! );
                        ^
BuggyHelloWorld.java:8: error: ';' expected
    System.out.pritnln( "Hello, World! );
                                         ^
BuggyHelloWorld.java:9: error: illegal start of expression
    System.out.printin( Hello, Class!" );
          ^
BuggyHelloWorld.java:9: error: ';' expected
    System.out.printin( Hello, Class!" );
              ^
BuggyHelloWorld.java:9: error: ')' expected
    System.out.printin( Hello, Class!" );
                                    ^
BuggyHelloWorld.java:9: error: unclosed string literal
    System.out.printin( Hello, Class!" );
                                     ^
BuggyHelloWorld.java:10: error: ';' expected
    System.out.printin( "Hello, its' me!" ):
          ^
BuggyHelloWorld.java:10: error: ';' expected
    System.out.printin( "Hello, its' me!" ):
                                           ^
BuggyHelloWorld.java:13: error: class, interface, or enum expected
}
^
10 errors
```

Each error message consists of the source file name, the line number of the error, the nature of the error, and the actual location of the error (indicated by the caret symbol). In the case of an IDE, instead of presenting the errors in its command line interface screen, these errors are highlighted in the source code editor screen.

As can be seen, the number of error messages is greater than the actual number of errors. Although the number of error messages may exceed the number of true errors, it is always the case that the very first error message corresponds to the very first syntax error. In the above example, "illegal start of type" pointing to `/---` is a true syntax error. Fixing the very first syntax error in the source code first is a good strategy.

There are two other types of errors: **run-time errors** and **logic errors**. Runtime errors are those that occur during the execution of code, interrupting the execution. They often result in a premature termination of the program. Logic errors do not necessarily result in run-time errors, but occur due to misconception or flaws in the logic of the programmer.

1.3 Using Multiple Statements

1.3.1 `System.out.println` and `System.out.print` (Reprise)

As mentioned earlier, a method can contain multiple statements. If multiple statements appear in a method, they act in the order of appearance. Multiple statements can be used to write a program that executes a complex task.

Consider the following program that prints the "ABC Song":

```java
public class MultiLines
{
  public static void main( String[] args )
  {
    System.out.println( "A B C D E F G" );
    System.out.println( "H I J K L M N O P" );
    System.out.println( "Q R S and T U V" );
    System.out.println( "W X Y and Z" );
    System.out.println( "Now I know my A B C" );
    System.out.println( "Won't you sing along with me" );
  }
}
```

Listing 1.6 A source code with multiple `println` statements

The program executes the six `System.out.println` statements from top to bottom, and produces the following six-line output.

```
A B C D E F G
H I J K L M N O P
Q R S and T U V
W X Y and Z
Now I know my A B C
Won't you sing along with me
```

Recall that `System.out.print` is the version of `System.out.println` without the newline at the end.

The next code is a variant of the previous code. We have changed the first, third, and fifth `System.out.println` statements to `System.out.print` statements.

```java
public class MultiLines01
{
  public static void main( String[] args )
  {
    System.out.print( "A B C D E F G" );
    System.out.println( "H I J K L M N O P" );
    System.out.print( "Q R S and T U V" );
    System.out.println( "W X Y and Z" );
    System.out.print( "Now I know my A B C" );
    System.out.println( "Won't you sing along with me" );
  }
}
```

Listing 1.7 A source code with multiple `print` and `println` statements

The result of executing the code is as follows:

```
A B C D E F GH I J K L M N O P
Q R S and T U VW X Y and Z
Now I know my A B CWon't you sing along with me
```

The use of `System.out.print` at three places has reduced the number of output lines from six to three. In each of the three lines, two outputs appear side-by-side with no gaps in between. Thus, to make the connections appear correct, some characters are needed in between. The next code example fixes this spacing issue by appending the command and the space (i.e., `", "`) to the first, third, and fifth sequences.

```java
public class MultiLines02
{
  public static void main( String[] args )
  {
    System.out.print( "A B C D E F G, " );
    System.out.println( "H I J K L M N O P" );
    System.out.print( "Q R S and T U V, " );
    System.out.println( "W X Y and Z" );
    System.out.print( "Now I know my A B C, " );
    System.out.println( "Won't you sing along with me" );
  }
}
```

Listing 1.8 A source code with multiple `print` and `println` statements with some punctuation

The result of executing the code is as follows:

```
A B C D E F G, H I J K L M N O P
Q R S and T U V, W X Y and Z
Now I know my A B C, Won't you sing along with me
```

1.3.2 Printing Multiple-Line Texts on the Screen

In a manner similar to the code for the ABC song, we can write a program that produces some selected texts on the screen, for example, the poem "Autumn" by an American poet Henry Wadsworth Longfellow (February 27, 1807 to March 24, 1882).

Thou comest, Autumn, heralded by the rain
With banners, by great gales incessant fanne
Brighter than brightest silks of Samarcand,
And stately oxen harnessed to thy wain!
Thou standest, like imperial Charlemagne,
Upon thy bridge of gold; thy royal hand
Outstretched with benedictions o'er the land,
Blessing the farms through all thy vast domain!
Thy shield is the red harvest moon, suspended
So 'long' beneath the heaven's o'er-hanging eaves;
Thy steps are by the farmer's prayers attended;
Like flames upon an altar shine the sheaves;
And, following thee, in thy ovation splendid,
Thine almoner, the wind, scatters the golden leaves!

The code `Autumn.java` that appears next produces this poem on the screen by combining `System.out.print` and `System.out.println` statements, where each line of the poem is split into two statements.

```java
public class Autumn
{
  public static void main( String[] args )
  {
    System.out.println( "Autumn, by Longfellow" );
    System.out.println();
    System.out.print( "Thou comest, Autumn, " );
    System.out.println( "heralded by the rain" );
    System.out.print( "With banners, " );
    System.out.println( "by great gales incessant fanne" );
    System.out.print( "Brighter than brightest " );
    System.out.println( "silks of Samarcand," );
    System.out.print( "And stately oxen " );
    System.out.println( "harnessed to thy wain!" );
    System.out.print( "Thou standest, " );
    System.out.println( "like imperial Charlemagne," );
    System.out.print( "Upon thy bridge of gold; " );
    System.out.println( "thy royal hand" );
    System.out.print( "Outstretched with benedictions " );
    System.out.println( "o'er the land," );
    System.out.print( "Blessing the farms through " );
    System.out.println( "all thy vast domain!" );
    System.out.print( "Thy shield is the red harvest moon, " );
    System.out.println( "suspended" );
    System.out.print( "So long beneath the heaven's " );
    System.out.println( "o'er-hanging eaves;" );
    System.out.print( "Thy steps are by the farmer's " );
    System.out.println( "prayers attended;" );
    System.out.print( "Like flames upon an altar " );
    System.out.println( "shine the sheaves;" );
    System.out.print( "And, following thee, " );
    System.out.println( "in thy ovation splendid," );
    System.out.print( "Thine almoner, the wind, " );
    System.out.println( "scatters the golden leaves!" );
  }
}
```

Listing 1.9 A source code for `Autumn.java`

The program produces the following output:

```

Thou comest, Autumn, heralded by the rain
With banners, by great gales incessant fanne
Brighter than brightest silks of Samarcand,
And stately oxen harnessed to thy wain!
Thou standest, like imperial Charlemagne,
Upon thy bridge of gold; thy royal hand
Outstretched with benedictions o'er the land,
Blessing the farms through all thy vast domain!
Thy shield is the red harvest moon, suspended
So long beneath the heaven's o'er-hanging eaves;
Thy steps are by the farmer's prayers attended;
Like flames upon an altar shine the sheaves;
And, following thee, in thy ovation splendid,
Thine almoner, the wind, scatters the golden leaves!
```

1.3.3 Escaping Characters

Suppose we wish to print the following character sequence:

```
abc"def
```

To print a character sequence directly with `System.out.print` and `System.out.println`, we attach the double quotation mark before and after the sequence. What if the sequence were `abc"def` and we wrote out the statement as follows?

```
System.out.println( "abc"def" );
```

This would produce a compilation error.

The next code is one that has triple double quotation marks.

```
public class TripleQuote
{
  public static void main( String[] args )
  {
    System.out.println( "abc"def" );
  }
}
```

Listing 1.10 A code that attempts to use a quotation mark inside a character sequence

The compiler produces the following error messages:

```
TripleQuote.java:3: error: ')' expected
    System.out.println( "abc"def" );
                             ^
TripleQuote.java:3: error: unclosed string literal
    System.out.println( "abc"def" );
                                ^
TripleQuote.java:3: error: ';' expected
    System.out.println( "abc"def" );
                                    ^
TripleQuote.java:5: error: reached end of file while parsing
}
 ^
4 errors
```

What happened during the compilation attempt? The Java compiler tried to pair the first double quotation mark with another. It chose, however, to pair the second quotation mark with the first. The compiler then tried to make sense of the remainder `def"`, but it could not.

To correct this problem, we need to tell the compiler that the middle double quotation mark is not the end marker. Attaching a backslash \ before the quotation mark accomplishes this.

```
"abc\"def"
```

With this modification, the code looks like:

```
public class TripleQuoteCorrect
{
  public static void main( String[] args )
  {
    System.out.println( "abc\"def" );
  }
}
```

Listing 1.11 A code that prints a quotation mark inside a character sequence

and the code generates the output as intended:

```
abc"def
```

We call the action of attaching the backslash to relieve a symbol of its regular duties **escaping**.

With escaping, we can insert a newline character using the combination `\n`. To include a tab-stop character, we can use `\t` instead of using of the actual tab-stop. The benefit of using the `\t` is that the character is visible; if we use the tab-stop character as it is, it is difficult to tell later whether a gap we see is indeed a tab-stop or just a series of the white space.

Finally, to escape the backslash character, we use the double backslash `\\`.

Assuming that the tab-stop positions of a terminal program are at every eighth position starting from the left end, the statement:

```
System.out.println( "abcdefgh\n\"\\i\tj\nk" );
```

produces the following output:

```
abcdefgh
"\i     j
k
```

We can use escaping to print texts with quotation marks and backward slashes. Listing 1.12 is a program that prints a quotation from Mark Twain's *Adventures of Huckleberry Finn*. In one line of the quote, the addition of `System.out.println` and the indentation makes the line too long to fit in the width of 72 characters. To solve this issue, we split the line into two: the first half with `System.out.print` and the second half with `System.out.println` (Lines 17 and 18).

```
 1 public class HuckleberryFinn
 2 {
 3   public static void main(String[] args)
 4   {
 5     System.out.println("\\Quoted from Huckleberry Finn\\");
 6     System.out.println("I broke in and says:");
 7     System.out.println("\"They're in an awful peck of trouble, and\"");
 8     System.out.println("\"Who is?\"");
 9     System.out.println("\"Why, pap and mam and sis and Miss Hooker;");
10     System.out.println("\tand if you'd take your ferryboat and go up
         there\"");
11     System.out.println("\"Up where?  Where are they?\"");
12     System.out.println("\"On the wreck.\"");
13     System.out.println("\"What wreck?\"");
14     System.out.println("\"Why, there ain't but one.\"");
15     System.out.println("\"What, you don't mean the Walter Scott?\"");
16     System.out.println("\"Yes.\"");
17     System.out.print("\"Good land! what are they doin' there, ");
18     System.out.println("for gracious sakes?\"");
19     System.out.println("\"Well, they didn't go there a-purpose.\"");
20   }
21 }
```

Listing 1.12 A program that prints a quotation from Mark Twain's *Adventures of Huckleberry Finn*

Executing the code produces the following output.

```
\Quoted from Huckleberry Finn\
I broke in and says:
"They're in an awful peck of trouble, and"
"Who is?"
"Why, pap and mam and sis and Miss Hooker;
        and if you'd take your ferryboat and go up there"
"Up where? Where are they?"
"On the wreck."
"What wreck?"
"Why, there ain't but one."
"What, you don't mean the Walter Scott?"
"Yes."
"Good land! what are they doin' there, for gracious sakes?"
"Well, they didn't go there a-purpose."
```

Using `\n` as the newline, we can print multiple short lines into single statements, as shown in List 1.13. Note that most of the lines contain `\n` in the character sequence that needs to be printed.

```java
public class HuckleberryFinn01
{
  public static void main(String[] args)
  {
    System.out.print( "\\Quoted from Huckleberry Finn\\\n" );
    System.out.print( "I broke in and says:\n\"They're in" );
    System.out.print( " an awful peck of trouble, and\"\n" );
    System.out.print( "\"Who is?\"\n\"Why, pap and mam and " );
    System.out.print( "sis and Miss Hooker;\n\tand if you'd " );
    System.out.print( "take your ferryboat and go up there\"" );
    System.out.print( "\n\"Up where?  Where are they?\"\n" );
    System.out.print( "\"On the wreck.\"\n\"What wreck?\"\n" );
    System.out.print( "\"Why, there ain't but one.\"\n" );
    System.out.print( "\"What, you don't mean the Walter " );
    System.out.print( "Scott?\"\n\"Yes.\"\n\"Good land! " );
    System.out.print( "what are they doin' there, for " );
    System.out.print( "gracious sakes?\"\n\"Well, they " );
    System.out.print( "didn't go there a-purpose.\"\n" );
  }
}
```

Listing 1.13 A program that uses squeezed print statements to produce the same quotation from Mark Twain's *Adventures of Huckleberry Finn* as before

The execution produces exactly the same result as before.

Java uses many symbol sequences, including escaping. Table 1.2 summarizes all such symbol sequences.

1.3.4 Printing Shapes

Previously, we used multiple `System.out.println` statements to produce multiple-line texts on the terminal screen. Expanding on the idea, now we write Java programs that print shapes on the terminal screen.

Table 1.2 The list of meaningful symbols in Java

[]	Arrays	()	Parameters
{}	Code block	<>	Type parameter
.	Class membership	=	Assignment
;	Statement separator	,	Parameter separator
?	If-then-else value selection	:	Case presentation
+	Addition/concatenation	-	Subtraction, negative sign
*	Multiplication	/	Quotient
%	Remainder, format type parameter		
++	Increment	-	Decrement
+=	Direct addition	-=	Direct subtraction
*=	Direct multiplication	/=	Direct quotient
%=	Direct remainder		
==	Equality	!=	Inequality
>	Greater than	<	Smaller than
>=	Greater than or equal to	>=	Smaller than or equal to
&&	Logical AND	||	Logical OR
!	Negation		
«	Signed left shift	»	Signed right shift
«<	Unsigned left shift	»>	Unsigned right shift
&	Bit-wise AND	|	Bit-wise OR
^	Bit-wise XOR		
&=	Direct bit-wise AND	|=	Direct bit-wise OR
^=	Direct bit-wise XOR		
@	Javadoc keyword	//	Line comment
/*	Multiple-line comment start	/**	Javadoc start
*/	Multiple-line comment/Java end		
\	Escape	\\	Backslash character
\n	The newline character	\t	The tab-stop character
\'	Single quote in a char literal	%%	the % Character in format strings
%n	The newline character in format strings		

Suppose we want to print the figure of a right-angled triangle like the one appears next:

```
      /|
     / |
    /  |
   /   |
  /    |
 /     |
/______|
```

In the drawing, we use the forward slash / for the left side of the triangle, the vertical | for the right side of the triangle, and the underscore _ for the bottom side.

The following code will do the job:

```java
//-- print a triangle
public class Triangle
{
  public static void main( String[] args )
  {
    System.out.println( "      /|" );
    System.out.println( "     / |" );
    System.out.println( "    /  |" );
    System.out.println( "   /   |" );
    System.out.println( "  /    |" );
    System.out.println( " /     |" );
    System.out.println( "/______|" );
  }
}
```

Listing 1.14 The code for producing a right-angled triangle

How about an isosceles triangle, like this one?

```
      /\
     /  \
    /    \
   /      \
  /        \
 /          \
/____________\
```

Using the \ for the right edge, we come up with the following code:

```java
//-- print an isosceles
public class Isosceles
{
  //-- main method
  public static void main( String[] args )
  {
    System.out.println( "      /\\" );  // line 1
    System.out.println( "     /  \\" ); // line 2
    System.out.println( "    /    \\" );  // line 3
    System.out.println( "   /      \\" ); // line 4
    System.out.println( "  /        \\" );  // line 5
    System.out.println( " /          \\" ); // line 6
    System.out.println( "/____________\\" );  // line 7
  }
}
```

Listing 1.15 A code for printing on the screen an isosceles triangle

Let’s try printing an isosceles upside down, as shown next:

```
\------------/
 \          /
  \        /
   \      /
    \    /
     \  /
      \/
```

The shape looks a bit unwieldy, since we are using the dash to draw the top line. The triangle will look better if we draw the top line using an over-line bar character, but unfortunately, such a character does not exist in our standard character set, so the dash is our only option.

```java
//-- print an isosceles upside down
public class UpsideDownIsoscelesCorrect
{
  //-- main method
  public static void main( String[] args )
  {
    System.out.println( "\\------------/" );  // line 7
    System.out.println( " \\          /" ); // line 6
    System.out.println( "  \\        /" );  // line 5
    System.out.println( "   \\      /" ); // line 4
    System.out.println( "    \\    /" );  // line 3
    System.out.println( "     \\  /" );   // line 2
    System.out.println( "      \\/" );    // line 1
  }
}
```

Listing 1.16 The code for producing an upside-down isosceles triangle on the screen

Try writing programs that draw other interesting shapes!

Summary

- A command line interface is an environment in which the user, through typing commands after a prompt, interacts with the system.
- In command line interfaces and programming environments, there exists a “working folder”.
- The standard header of a Java class is `public class CLASS_NAME`. Its file name should be `CLASS_NAME.java`.
- An executable Java class has `public static void main( String[] arg )`.
- To compile a Java program, use the command `javac CLASS_NAME.java`.
- The Java compiler may produce compilation errors due to syntax errors.
- The command to use when executing a Java bytecode by the name of `CLASS_NAME` is `java CLASS_NAME`.
- A code that compiles may produce terminating errors. Such terminating errors are called run-time errors.
- A code that compiles and is free of run-time errors may still not run correctly. Logical errors are the culprits.
- Java programs use pairs of curly braces to define code blocks.
- Unless they are appearing in the source code for an object class, methods need to have the `static` attribute.
- Methods and classes may have a visibility attribute.
- Method declarations must have both a return value specification and a parameter specification.
- In a method, multiple statements may appear. Each statement ends with a semicolon.
- `System.out.println` is a method that produces the value of its parameter on the screen and then prints a newline.
- `System.out.print` is a method that produces the value of its parameter on the screen.

- To print the double quotation mark, the escape sequence of `\"` is used.
- To print the backslash as a character, the escape sequence of `\\` is used.
- There are three types of comments: single line comments, multiple-line comments, and Javadocs.

Exercises

1. **Terminology** Answer the following questions:
   (a) What is the formal name of the process for creating an executable Java code from a .java file? What about the command used in the Unix environment for creating such an executable code?
   (b) What is the file extension of a Java byte code?
   (c) What is the command used for executing a Java byte code?
   (d) In a .java file two words usually appear before its class name. What are they?
   (e) What are the three words that precede the `main` method in a .java file?
   (f) State the differences between `System .out.print` and `System.out.println`.
   (g) What are the three categories of errors in programming?
   (h) In `String` literals, what sequences of characters must you use to print the double quote, the backslash, and the newline?
2. **Main Declaration** Of the following possible declarations for the method `main`, which ones will compile?
   (a) `public static void main( String[] args )`
   (b) `static void main( String[] args )`
   (c) `public static main( String[] args )`
   (d) `public static void( String[] args )`
   (e) `public static void main( )`
   (f) `public static void main( String[] )`
   (g) `public static void main( args )`
3. **Fixing errors** The following code contains some errors and will not compile. State what we must fix so that it will compile.

```
public class MyBuggyProgram {
  public static main( []String args )
  [
    System.out.prink( 'Hello!' ):
  ]
}
```

4. **Escaping** Which of the following require a backslash when being included in a `String` literal (i.e., a series of characters flanked by a pair of double quotation marks)?
   - `A`
   - `/`
     (i.e., the forward slash)
   - `\`
     (i.e, the backslash)
   - `"`
   - `%`
     (i.e., the percentage symbol)
   - `@`
     (i.e., the at sign)

Programming Projects

5. **Alma Mater** Write a program named `AlmaMater` that upon execution prints the Alma Mater of your school. For the University of Miami, the output of the program should look like:

```
UM ALMA MATER

Southern suns and sky blue water,
Smile upon you Alma mater;
Mistress of this fruitful land,
With all knowledge at your hand,
Always just to honor true,
All our love we pledge to you.
Alma Mater, stand forever
On Biscayne's wondrous shore.
```

6. **Printing a Diamond** Write a program named `Diamond.java` that prints the shape of a diamond of height 10 and width 10 as shown:

```
    /\
   /  \
  /    \
 /      \
/        \
\        /
 \      /
  \    /
   \  /
    \/
```

7. **Printing a Filled Diamond** Write a program named `DiamondFilled.java` that prints the shape of a diamond of height 10 and width 10 with the white space filled with forward slashes on the left-hand side and backward slashes on the right-hand side, as shown:

```
    /\
   //\\
  ///\\\
 ////\\\\
/////\\\\\
\\\\\/////
 \\\\////
  \\\///
   \\//
    \/
```

8. **Printing an X with 'X'**
   Write a program, `XwithXs`, that produces the following shape on the screen:

```
X       X
 X     X
  X   X
   X X
    X
   X X
  X   X
 X     X
X       X
```

9. **Cascaded printing** Write a program, `CascadedChildren`, that prints the first two verses of "Children" by Henry Wadsworth Longfellow with increasing indentations:

```
Come to me, O ye children!
 For I hear you at your play,
  And the questions that perplexed me
   Have vanished quite away.

    Ye open the eastern windows,
     That look towards the sun,
      Where thoughts are singing swallows
       And the brooks of morning run.
```

10. **Slashes** Write a program, `Slashes`, that produces the following shape on the screen:

```
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
```

11. **Backlashes** Write a program, `BackSlashes`, that produces the following shape on the screen:

```
\ \ \ \ \ \ \ \ \ \ \ \
 \ \ \ \ \ \ \ \ \ \ \
\ \ \ \ \ \ \ \ \ \ \ \
 \ \ \ \ \ \ \ \ \ \ \
\ \ \ \ \ \ \ \ \ \ \ \
 \ \ \ \ \ \ \ \ \ \ \
\ \ \ \ \ \ \ \ \ \ \ \
 \ \ \ \ \ \ \ \ \ \ \
\ \ \ \ \ \ \ \ \ \ \ \
 \ \ \ \ \ \ \ \ \ \ \
```

12. **Tabstop** You can use the tab-stop character `\t` to line things up (to improve the readability of the text output). Consider the following code:

```java
public class TestTabStop
{
  public static void main( String[] args )
  {
    System.out.println( "Abbie Zuckerman 23yrs Soccer" );
    System.out.println( "Brittany Ying 21yrs Swimming" );
    System.out.println( "Caty Xenakis 22yrs Softball" );
    System.out.println( "Dee Wick 20yrs Basketball" );
    System.out.println( "Eva Venera 19yrs Polo" );
  }
}
```

The code produces the following output:

```
Abbie Zuckerman 23yrs Soccer
Brittany Ying 21yrs Swimming
Caty Xenakis 22yrs Softball
Dee Wick 20yrs Basketball
Eva Venera 19yrs Polo
```

Revise the program so that it uses the same code, but replaces the whitespace (inside each pair of double quotation marks) with the tab-stop. Run the program to see how it changes the way the information appears on the screen.

13. **Self introduction** Write a program named `SelfIntro` that introduces yourself as follows:

```
My name is NAME.
 I was born in PLACE.
  My favorite television program is PROGRAM.
   I woke up at HOUR:MINUTE today.
    I own NUMBER books.
     My target grade point average is GPA.
```

The uppercase words `NAME`, `PLACE`, `PROGRAM`, `HOUR`, `MINUTE`, `NUMBER`, and `GPA`, are placeholders, so you must substitute them with appropriate text. `PROGRAM` must appear with double quotation marks. For example,

```
My name is Mitsu Ogihara.
 I was born in Japan.
  My favorite television program is "Parks and Recreation".
   I woke up at 6:30 today.
    I own 1000 books.
     My target grade point average is 4.0.
```