

## 1

### Getting started

7

Introduction	8
Installing the JDK	10
Writing a first Java program	12
Compiling & running programs	14
Creating a variable	16
Recognizing data types	18
Creating constants	20
Adding comments	21
Troubleshooting problems	22
Summary	24

## 2

### Performing operations

25

Doing arithmetic	26
Assigning values	28
Comparing values	30
Assessing logic	32
Examining conditions	34
Setting precedence	36
Escaping literals	38
Working with bits	40
Summary	42

## 3

### Making statements

43

Branching with if	44
Branching alternatives	46
Switching branches	48
Looping for	50
Looping while true	52
Doing do-while loops	54
Breaking out of loops	56
Returning control	58
Summary	60

# 4

## Directing values

61

Casting type values	62
Creating variable arrays	64
Passing an argument	66
Passing multiple arguments	68
Looping through elements	70
Changing element values	72
Adding array dimensions	74
Catching exceptions	76
Summary	78

# 5

## Manipulating data

79

Exploring Java classes	80
Doing mathematics	82
Rounding numbers	84
Generating random numbers	86
Managing strings	88
Comparing strings	90
Searching strings	92
Manipulating characters	94
Summary	96

# 6

## Creating classes

97

Forming multiple methods	98
Understanding program scope	100
Forming multiple classes	102
Extending an existing class	104
Creating an object class	106
Producing an object instance	108
Encapsulating properties	110
Constructing object values	112
Summary	114

# 7

## Importing functions

115

Handling files	116
Reading console input	118
Reading files	120
Writing files	122
Sorting array elements	124
Making array lists	126
Managing dates	128
Formatting numbers	130
Summary	132

# 8

## Building interfaces

133

Creating a window	134
Adding push buttons	136
Adding labels	138
Adding text fields	140
Adding item selectors	142
Adding radio buttons	144
Arranging components	146
Changing appearance	148
Summary	150

# 9

## Recognizing events

151

Listening for events	152
Generating events	153
Handling button events	154
Handling item events	156
Reacting to keyboard events	158
Responding to mouse events	160
Announcing messages	162
Requesting input	164
Summary	166

# 10

## Deploying programs

167

Producing an application	168
Distributing programs	170
Building Java archives	172
Deploying applications	173
Creating Android projects	174
Exploring project files	176
Adding resources & controls	178
Inserting Java code	180
Testing the application	182
Deploying Android apps	184
Summary	186

## Index

187

# Preface

The creation of this book has provided me, Mike McGrath, a welcome opportunity to update my previous books on Java programming with the latest techniques. All examples I have given in this book demonstrate Java features supported by current compilers on both Windows and Linux operating systems, and the book's screenshots illustrate the actual results produced by compiling and executing the listed code, or by implementing code snippets in the Java shell.

## Conventions in this book

In order to clarify the code listed in the steps given in each example, I have adopted certain colorization conventions. Components of the Java language itself are colored blue; programmer-specified names are red; numeric and string values are black; and comments are green, like this:

```
// Store then output a text string value.  
String message = "Welcome to Java programming!" ;  
System.out.println( message ) ;
```

Additionally, in order to identify each source code file described in the steps, a colored icon and file name appears in the margin alongside the steps, like these:



App.java



App.class



App.jar



App.xml

## Grabbing the source code

For convenience, I have placed source code files from the examples featured in this book into a single ZIP archive. You can obtain the complete archive by following these easy steps:

- 1 Browse to [www.ineasysteps.com](http://www.ineasysteps.com) then navigate to [Free Resources](#) and choose the [Downloads](#) section
- 2 Find [Java in easy steps, 6th Edition](#) in the list, then click on the hyperlink entitled [All Code Examples](#) to download the archive
- 3 Now, extract the archive contents to any convenient location on your computer

I sincerely hope you enjoy discovering the programming possibilities of Java and have as much fun with it as I did in writing this book.

Mike McGrath

# 1

# Getting started

*Welcome to the exciting world of Java programming.*

*This chapter shows how to create and execute simple Java programs, and demonstrates how to store data within programs.*

- 8** Introduction
- 10** Installing the JDK
- 12** Writing a first Java program
- 14** Compiling & running programs
- 16** Creating a variable
- 18** Recognizing data types
- 20** Creating constants
- 21** Adding comments
- 22** Troubleshooting problems
- 24** Summary



# Introduction

The Java™ programming language was first developed in 1990 by an engineer at Sun Microsystems named James Gosling. He was unhappy using the C++ programming language so he created a new language that he named “Oak”, after the oak tree that he could see from his office window.

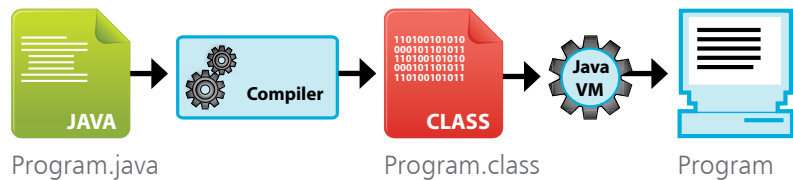
As the popularity of the World Wide Web grew, Sun recognized that Gosling’s language could be developed for the internet. Consequently, Sun renamed the language “Java” (simply because that name sounded cool) and made it freely available in 1995. Developers around the world quickly adopted this exciting new language and, because of its modular design, were able to create new features that could be added to the core language. The most endearing additional features were retained in subsequent releases of Java as it developed into the comprehensive version of today.

The essence of Java is a library of files called “classes”, which each contain small pieces of ready-made proven code. Any of these classes can be incorporated into a new program, like bricks in a wall, so that only a relatively small amount of new code ever needs to be written to complete the program. This saves the programmer a vast amount of time, and largely explains the huge popularity of Java programming. Additionally, this modular arrangement makes it easier to identify any errors than in a single large program.

Java technology is both a programming language and a platform. In Java programming, the source code is first written as human-readable plain text files ending with the `.java` extension. These are compiled into machine-readable `.class` files by the `javac` compiler. The `java` interpreter can then execute the program with an instance of the Java Virtual Machine (Java VM):



The New icon pictured above indicates a new or enhanced feature introduced with the latest version of Java.



As the Java VM is available on many different operating systems, the same `.class` files are capable of running on Windows, Linux and Mac operating systems – so Java programmers theoretically enjoy the cross-platform ability to “write once, run anywhere”.

...cont'd

In order to create Java programs, the Java class libraries and the **javac** compiler need to be installed on your computer. In order to run Java programs, the **Java™ Runtime Environment (JRE)** needs to be installed to supply the **java** interpreter. All of these components are contained in a freely available package called the **Java™ Platform, Standard Edition Development Kit (JDK)**.

The Java programs in this book use version JDK 9, which incorporates both the Development Kit itself and the Runtime Environment, and can be downloaded from the Oracle® website at [www.oracle.com/technetwork/java/javase/downloads](http://www.oracle.com/technetwork/java/javase/downloads)



The Oracle download page also features other packages, but only the JDK 9 package is required to get started with Java programming.

The JDK 9 package is available in versions for 32-bit and 64-bit variants of the Linux, Mac, Solaris and Windows platforms – accept the Oracle License Agreement, then select the appropriate version for your computer to download the Java Development Kit.

Java SE Development Kit 9		
You must accept the <a href="#">Oracle Binary Code License Agreement for Java SE</a> to download this software.		
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.		
Product / File Description	File Size	Download
Linux	298.02 MB	<a href="#">jdk-9_linux-x64_bin.rpm</a>
Linux	330.23 MB	<a href="#">jdk-9_linux-x64_bin.tar.gz</a>
macOS	371.64 MB	<a href="#">jdk-9_osx-x64_bin.dmg</a>
Windows	358.69 MB	<a href="#">jdk-9_windows-x64_bin.exe</a>
Solaris SPARC	207.05 MB	<a href="#">jdk-9_solaris-sparcv9_bin.tar.gz</a>



There is no truth in the rumor that JAVA stands for “Just Another Vague Acronym”.

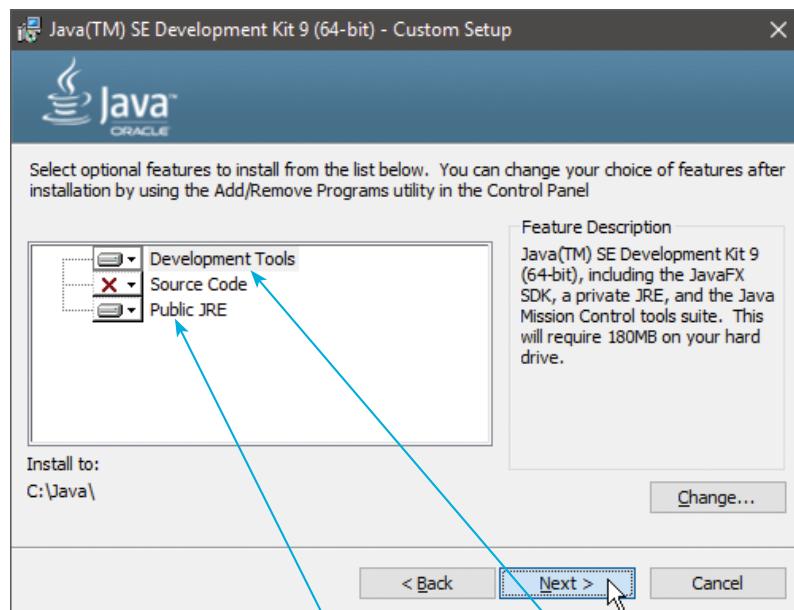


A previous version of the JRE may be installed so your web browser can run Java applets. It is best to uninstall this to avoid confusion with the newer version in JDK 9.

# Installing the JDK

Select the appropriate Java Development Kit (JDK) package for your system from the Oracle® downloads page, and then follow these steps to install Java on your computer:

- 1 Uninstall any previous versions of the JDK and/or Java Runtime Environment (JRE) from your system
- 2 Start the installation and accept the License Agreement
- 3 When the “Custom Setup” dialog appears, either accept the suggested installation location or click the **Change** button to choose your preferred location – such as **C:\Java** for Windows systems or **/usr/Java** for Linux systems



- 4 Ensure that the **Public JRE** and **Development Tools** features are selected from the list. Optionally, you may deselect the other features as they are not required to start programming with this book
- 5 Click the **Next** button to install all the necessary Java class libraries and tools at the chosen location



You can start out by installing just the minimum features to avoid confusion.



...cont'd

The tools to compile and run Java programs are normally operated from a command-line prompt and are located in the **bin** sub-directory of the Java directory. They can be made available system-wide by adding their location to the system path:

- On Windows, navigate through **Control Panel, System, Advanced System Settings, Advanced** tab, **Environment Variables**, then select the system variable named “Path”. Click the **Edit** button and add the address of Java’s **bin** sub-directory to the list (e.g. **C:\Java\bin**), then click **OK** to apply the change.
- On Linux, add the location of Java’s **bin** sub-directory to the system path by editing the **.bashrc** file in your home directory. For instance, add **PATH=\$PATH:/usr/Java/bin** then save the file.

You are now able to test the environment:

- 1 Open a command-line prompt window, such as **Windows PowerShell** or **Linux Terminal**
- 2 Type the command **java -version** then hit the **Enter** key to see the Java interpreter’s version number
- 3 Next, type the command **javac -version** then hit the **Enter** key to see the Java compiler’s version number
- 4 Now, type the command **jshell -version** then hit the **Enter** key to see the Java shell version number
- 5 Ensure that all version numbers match (9), and you’re ready to begin Java programming

```
Java in easy steps
C:\> java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+181)
Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode)
C:\> javac -version
javac 9
C:\> jshell -version
jshell 9
```



Paths that contain spaces must be enclosed within double quotes and terminated by a semicolon on older versions of Windows. For example, with the path “**C:\Program Files\Java\jdk-9\bin**”;



If the **.bashrc** file is not visible in your Linux home directory choose View, Show Hidden Files to reveal it.



The Java shell **jshell** is a new feature in Java 9. This interactive tool lets you quickly test snippets of code, without the need to first compile the code. It is used in the next chapter to demonstrate the various “operators” available in Java programming.



# Writing a first Java program

All Java programs start as text files that are later used to create “class” files, which are the actual runnable programs. This means that Java programs can be written in any plain text editor, such as the Windows Notepad application.

Follow these steps to create a simple Java program that will output the traditional first program greeting:



Hello.java



Java is a case-sensitive language where “Hello” and “hello” are distinctly different – traditionally, Java program names should always begin with an uppercase letter.



Java programs are always saved as their exact program name followed by the “.java” extension.

- 1 Open a plain text editor, like Notepad, and type this code exactly as it is listed – to create a class named “Hello”
 

```
class Hello
{
}
```
- 2 Between the curly brackets of the **Hello** class, insert this code – to create a “main” method for the **Hello** class
 

```
public static void main ( String[] args )
{
}
```
- 3 Between the curly brackets of the **main** method, insert this line of code – stating what the program will do
 

```
System.out.println( "Hello World!" );
```
- 4 Save the file at any convenient location, but be sure to name it precisely as **Hello.java** – the complete program should now look like this:

```

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

```

...cont'd

The separate parts of the program code on the opposite page can be examined individually to understand each part more clearly:

## The Program Container

```
class Hello {    }
```

The program name is declared following the **class** keyword, and followed by a pair of curly brackets. All of the program code that defines the **Hello** class will be contained within these curly brackets.

## The Main Method

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

This fearsome-looking line is the standard code that is used to define the starting point of nearly all Java programs. It will be used in most examples throughout this book exactly as it appears above – so it may be useful to memorize it.

The code declares a method named “main” that will contain the actual program instructions within its curly brackets.

Keywords **public static void** precede the method name to define how the method may be used, and are explained in detail later.

The code ( **String[] args** ) is useful when passing values to the method, and is also fully explained later in this book.

## The Statement

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

Statements are actual instructions to perform program tasks, and must always end with a semicolon. A method may contain many statements inside its curly brackets to form a “statement block” defining a series of tasks to perform, but here a single statement instructs the program to output a line of text.

Turn to page 14 to discover how to compile and run this program.



All stand-alone Java programs must have a main method. Java applets are different, and their format is explained later.



Create a “MyJava” directory in which to save all your Java program files.

# Compiling & running programs



On Windows use the Windows PowerShell app or the older Command Prompt app to provide a command-line prompt, and on Linux use a Terminal window.



At a prompt type **javac** and hit Return to reveal the Java compiler options.



You can also compile the source code from another location if you state the file's full path address to the javac compiler – in this case, **C:\MyJava\Hello.java**

Before a Java program can run, it must first be compiled into a **class** file by the Java compiler. This is located in Java's **bin** sub-directory, and is an application named **javac**. The instructions on page 11 described how to add the **bin** sub-directory to the system path so that **javac** can be invoked from any system location.

Follow these steps to compile the program on page 13:

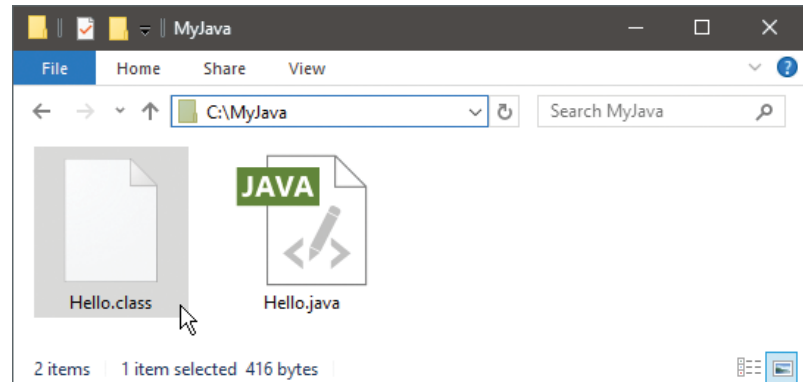
- 1 Open a command-line window, then navigate to the directory where you saved the **Hello.java** source code file
- 2 Type **javac** followed by a space then the full name of the source code file **Hello.java** and hit the **Enter** key

```

C:\> cd MyJava
C:\MyJava> javac Hello.java
C:\MyJava>
  
```

If the **javac** compiler discovers errors in the code it will halt and display a helpful report indicating the nature of the error – see page 22 for troubleshooting problems.

If the **javac** compiler does not find any errors it will create a new file with the program name and the **.class** file extension.



...cont'd

When the Java compiler completes compilation, the command-line prompt window focus returns to the prompt without any confirmation message – and the program is ready to run.

The Java program interpreter is an application named **java** that is located in Java's **bin** sub-directory – alongside the **javac** compiler. As this directory was previously added to the system path, on page 11, the **java** interpreter can be invoked from any location.

Follow these steps to run the program that was compiled using the procedure described on the page opposite:

- 1 Open a command-line prompt window, then navigate to the directory where the **Hello.class** program file is located
- 2 At the prompt, type **java** followed by a space then the program name **Hello** and hit the **Enter** key



```
C:\> cd MyJava
C:\MyJava> java Hello
Hello World!
C:\MyJava> █
```

The **Hello** program runs and executes the task defined in the statement within its main method – to output “Hello World!”. Upon completion, focus returns to the prompt once more.

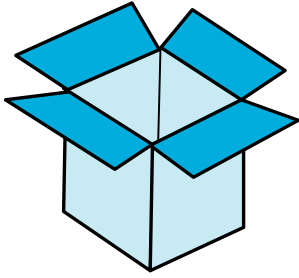
The process of compiling and running a Java program is typically combined in sequential steps, and is the same regardless of platform. The screenshot below illustrates the **Hello** program being compiled and run in combined steps on a Linux system:



```
Linux Terminal
user > cd MyJava
user > javac Hello.java
user > java Hello
Hello World!
user > █
```



Do not include the **.class** extension when running a program – only use the program name.



# Creating a variable

In Java programming, a “variable” is simply a useful container in which a value may be stored for subsequent use by the program. The stored value may be changed (vary) as the program executes its instructions – hence the term “variable”.

A variable is created by writing a variable “declaration” in the program, specifying the type of data that variable may contain and a given name for that variable. For example, the **String** data type can be specified to allow a variable named “message” to contain regular text with this declaration:

**String message ;**

Variable names are chosen by the programmer but must adhere to certain naming conventions. The variable name may only begin with a letter, dollar sign \$, or the underscore character `_`, and may subsequently have only letters, digits, dollar signs, or underscore characters. Names are case-sensitive, so “var” and “Var” are distinctly different names, and spaces are not allowed in names.

Variable names should also avoid the Java keywords listed in the table below, as these have special meaning in the Java language.



Each variable declaration must be terminated with a semicolon character – like all other statements.



Strictly speaking, some words in this table are not actually keywords – **true**, **false**, and **null** are all literals; **String** is a special class name; **const** and **goto** are reserved words (currently unused). These are included in the table because they must also be avoided when naming variables.

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

...cont'd

As good practice, variables should be named with words or easily recognizable abbreviations, describing that variable's purpose. For example, "button1" or "btn1" to describe button number one. Lowercase letters are preferred for single-word names, such as "gear", and names that consist of multiple words should capitalize the first letter of each subsequent word, such as "gearRatio" – the so-called "camelCase" naming convention.

Once a variable has been declared, it may be assigned an initial value of the appropriate data type using the equals sign = , either in the declaration or later on in the program, then its value can be referenced at any time using the variable's name.

Follow these steps to create a program that declares a variable, which gets initialized in its declaration then changed later:

1 Start a new program named "FirstVariable", containing the standard main method

```
class FirstVariable
{
    public static void main ( String[] args ) {
    }
```

2 Between the curly brackets of the main method, insert this code to create, initialize, and output a variable

```
String message = "Initial value" ;
System.out.println( message ) ;
```

3 Add these lines to modify and output the variable value

```
message = "Modified value" ;
System.out.println( message ) ;
```

4 Save the program as **FirstVariable.java**, then compile and run the program



FirstVariable.java



If you encounter problems compiling or running the program, you can get help from Troubleshooting problems on page 22.

```
Java in easy steps
C:\MyJava> javac FirstVariable.java
C:\MyJava> java FirstVariable
Initial value
Modified value
C:\MyJava>
```

# Recognizing data types

The most frequently-used data types in Java variable declarations are listed in this table, along with a brief description:



Due to the irregularities of floating-point arithmetic the **float** data type should never be used for precise values, such as currency – see page 130 for details.

Data type:	Description:	Example:
<b>char</b>	A single Unicode character	'a'
<b>String</b>	Any number of Unicode characters	"my String"
<b>int</b>	An integer number, from -2.14 billion to +2.14 billion	1000
<b>float</b>	A floating-point number, with a decimal point	3.14159265f
<b>boolean</b>	A logical value of either true or false	true

Notice that **char** data values must always be surrounded by single quotes, and **String** data values must always be surrounded by double quotes. Also, remember that **float** data values must always have an "f" suffix to ensure they are treated as a **float** value.

In addition to the more common data types above, Java provides these specialized data types for use in exacting circumstances:



All data type keywords begin with a lowercase letter except **String** – which is a special class.

Data type:	Description:
<b>byte</b>	Integer number from -128 to +127
<b>short</b>	Integer number from -32,768 to +32,767
<b>long</b>	Positive or negative integer exceeding 2.14 billion
<b>double</b>	Extremely long floating-point number

Specialized data types are useful in advanced Java programs – the examples in this book mostly use the common data types described in the top table.



...cont'd

Follow these steps to create a Java program that creates, initializes, and outputs variables of all five common data types:

- 1 Start a new program named "DataTypes" containing the standard main method  

```
class DataTypes
{
    public static void main ( String[] args ) {
    }
}
```



DataTypes.java

- 2 Between the curly brackets of the main method, insert these declarations to create and initialize five variables  

```
char letter = 'M' ;
String title = "Java in easy steps" ;
int number = 365 ;
float decimal = 98.6f ;
boolean result = true ;
```



Notice how the + character is used here to join (concatenate) text strings and stored variable values.

- 3 Add these lines to output an appropriate text **String** concatenated to the value of each variable  

```
System.out.println( "Initial is " + letter ) ;
System.out.println( "Book is " + title ) ;
System.out.println( "Days are " + number ) ;
System.out.println( "Temperature is " + decimal ) ;
System.out.println( "Answer is " + result ) ;
```

- 4 Save the program as **DataTypes.java**, then compile and run the program

```
Java in easy steps
C:\MyJava> javac DataTypes.java

C:\MyJava> java DataTypes
Initial is M
Book is Java in easy steps
Days are 365
Temperature is 98.6
Answer is true

C:\MyJava> _
```



The Java compiler will report an error if the program attempts to assign a value of the wrong data type to a variable – try changing the values in this example, then attempt to recompile the program to see the effect.

# Creating constants

The “final” keyword is a modifier that can be used when declaring variables to prevent any subsequent changes to the values that are initially assigned to them. This is useful when storing a fixed value in a program to avoid it becoming altered accidentally.

Variables created to store fixed values in this way are known as “constants”, and it is convention to name constants with all uppercase characters – to distinguish them from regular variables. Programs that attempt to change a constant value will not compile, and the **javac** compiler will generate an error message.

Follow these steps to create a Java program featuring constants:



Constants.java

1 Start a new program named “Constants” containing the standard main method

```
class Constants
{
    public static void main ( String[] args ) {
    }
}
```

2 Between the curly brackets of the main method, insert this code to create and initialize three integer constants

```
final int TOUCHDOWN = 6 ;
final int CONVERSION = 1 ;
final int FIELDGOAL = 3 ;
```

3 Now, declare four regular integer variables

```
int td , pat , fg , total ;
```

4 Initialize the regular variables – using multiples of the constant values

```
td = 4 * TOUCHDOWN ;
pat = 3 * CONVERSION ;
fg = 2 * FIELDGOAL ;
total = ( td + pat + fg ) ;
```

5 Add this line to display the total score

```
System.out.println( "Score: " + total ) ;
```

6 Save the program as **Constants.java**, then compile and run the program to see the output, Score: 33  
( 4 x 6 = 24, 3 x 1 = 3, 2 x 3 = 6, so 24 + 3 + 6 = 33 ).



The \* asterisk character is used here to multiply the constant values, and parentheses surround their addition for clarity.

# Adding comments

When programming in any language, it is good practice to add comments to program code to explain each particular section. This makes the code more easily understood by others, and by yourself, when revisiting a piece of code after a period of absence.

In Java programming, comments can be added across multiple lines between `/*` and `*/` comment identifiers, or on a single line after a `//` comment identifier. Anything appearing between `/*` and `*/`, or on a line after `//`, is completely ignored by the `javac` compiler.

When comments have been added to the `Constants.java` program, described opposite, the source code might look like this:

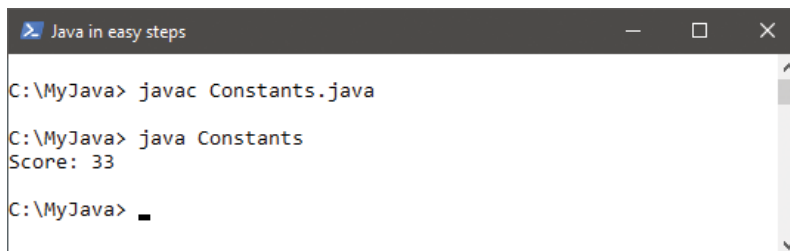
```
/*
    A program to demonstrate constant variables.
*/

class Constants
{
    public static void main( String args[] )
    {
        // Constant score values.
        final int TOUCHDOWN = 6 ;
        final int CONVERSION = 1 ;
        final int FIELDGOAL = 3 ;

        // Calculate points scored.
        int td , pat , fg , total ;
        td = 4 * TOUCHDOWN ;           // 4x6=24
        pat = 3 * CONVERSION ;         // 3x1= 3
        fg = 2 * FIELDGOAL ;          // 2x3= 6
        total = ( td + pat + fg ) ;    // 24+3+6=33

        // Output calculated total.
        System.out.println( "Score: " + total ) ;
    }
}
```

Saved with comments, the program compiles and runs as normal:



```
Java in easy steps
C:\MyJava> javac Constants.java
C:\MyJava> java Constants
Score: 33
C:\MyJava> █
```



Constants.java  
(commented)



You can add a statement that attempts to change the value of a constant, then try to recompile the program to see the resulting error message.

# Troubleshooting problems

Sometimes, the **javac** compiler or **java** interpreter will complain about errors, so it's useful to understand their cause and how to quickly resolve the problem. In order to demonstrate some common error reports, this code contains some deliberate errors:



Test.java

```
class test
{
    public static void main ( String[] args )
    {
        String text ;
        System.out.println( "Test " + text )
    }
}
```

A first attempt to compile **Test.java** throws up this error report:

```
Java in easy steps
C:\MyJava> javac Test.java
javac : The term 'javac' is not recognized as the name of a
cmdlet, function, script file, or operable program. Check the
spelling of the name, or if a path was included, verify that
the path is correct and try again.
C:\MyJava> █
```

- Cause – the **javac** compiler cannot be found.
- Solution – edit the system **PATH** variable, as described on page 11, or use its full path address to invoke the compiler.



The path address must be enclosed within quotation marks if it contains any spaces, such as the path address "C:\Program Files\Java".

```
Java in easy steps
C:\MyJava> javac Test.java
javac: file not found: Test.java
Usage: javac <options> <source files>
use --help for a list of possible options
C:\MyJava> █
```

- Cause – the file **Test.java** cannot be found.
- Solution – navigate to the directory where the file is located, or use the full path address to the file in the command.

...cont'd

```
Java in easy steps
C:\MyJava> javac Test.java
Test.java:6: error: ';' expected
    System.out.println( "Test " + text )
                               ^
1 error
```

- Cause – the statement is not terminated correctly.
- Solution – in the source code add a semicolon at the end of the statement, then save the file to apply the change.

```
Java in easy steps
C:\MyJava> javac Test.java
Test.java:1: error: class test is public, should be declared in a
file named test.java
public class test
      ^
1 error
```

- Cause – the program name and class name do not match.
- Solution – in the source code change the class name from **test** to **Test**, then save the file to apply the change.

```
Java in easy steps
C:\MyJava> javac Test.java
Test.java:6: error: variable text might not have been initialized
    System.out.println( "Test " + text ) ;
                               ^
1 error
```

- Cause – the variable **text** has no value.
- Solution – in the variable declaration assign the variable a valid **String** value, for instance = "**success**", then save the file.

```
Java in easy steps
C:\MyJava> javac Test.java
C:\MyJava> java Test
Test success
C:\MyJava> █
```



You must run the program from within its directory – you cannot use a path address as the Java launcher requires a program name, not a file name.

# Summary

- Java is both a programming language and a runtime platform.
- Java programs are written as plain text files with a **.java** file extension.
- The Java compiler **javac** creates compiled **.class** program files from original **.java** source code files.
- The Java interpreter **java** executes compiled programs using an instance of the Java Virtual Machine.
- The Java VM is available on many operating system platforms.
- Adding Java's **bin** sub-directory to the system **PATH** variable allows the **javac** compiler to be invoked from anywhere.
- Java is a case-sensitive language.
- The standard **main** method is the entry point for Java programs.
- The **System.out.println()** statement outputs text.
- A Java program file name must exactly match its class name.
- Java variables can only be named in accordance with specified naming conventions, and must avoid the Java keywords.
- In Java programming, each statement must be terminated by a semicolon character.
- The most common Java data types are **String**, **int**, **char**, **float** and **boolean**.
- **String** values must be enclosed in double quotes; **char** values in single quotes; and **float** values must have an "f" suffix.
- The **final** keyword can be used to create a constant variable.
- Comments can be added to Java source code between **/\*** and **\*/**, on one or more lines, or after **//** on a single line.
- Error reports identify compiler and runtime problems.