Essential Java (1st Edition)

Rahul Batra

18 December 2013

Table of Contents

To Pria

PREFACE

Welcome to the first edition of *Essential Java*, a short (e)book meant as a gentle introduction to the basics of Java. Although no prior programming experience is necessary, any knowledge of how programs work will benefit the reader greatly while reading the text. As you can note from the length of the book, it does not cover a ton of topics that you would eventually need to know before calling yourself a Java programmer. The *Further Reading* section at the end of the book should give you some decent pointers on what to pick up next.

For the curious, the book was typeset using *Troff* and its *ms* macro set. The font family used was *Bookman*.

Your questions, comments, criticism, encouragement and corrections are most welcome and you can e-mail me at *rhlbatra[aht]hotmail[dot]com*. Ill try answering all on-topic mails and will try to include suggestions, errors and omissions in future editions.

Rahul Batra (18th December 2013)

ABOUT THE AUTHOR

Rahul Batra was first introduced to programming in 1996 in GWBASIC, but he did not seriously foray into it till 2001 when he started learning C++. Along the way, there were dabblings in many other languages like C, Ruby, Perl and Java. His first book *A Primer on SQL* was released in October 2012 and became very popular totalling over 15000 downloads in the first year.

Rahul has been programming professionally since 2006 and currently lives and works in Gurgaon, India.

ACKNOWLEDGEMENTS

This text would not have been possible without the unwavering support of my family and friends and I owe many thanks to them. My parents, who raised me and always kept my education as the foremost priority, and without whose encouragement to pursue my dreams I would not even have started this. My wife, Pria, who kept telling me to get into Java and kept on believing that I could write.

I also owe my gratitude to my sister for always looking out for me. And finally many thanks to my niece, newphew and friends for bring me such joy into my life.

LICENSING

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements: The modified version must be labeled as such.
The person making the modifications must be identified and the modifications dated.
Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
The location of the original unmodified document must be identified.
The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that: If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.

1. AN INTRODUCTION TO JAVA

1.1. What is Java?

To perform useful tasks on a computer, we either use a prebuilt software application (like a text editor) or build one ourselves. This process of making software using instructions that a computer can understand is called *programming*. The set of instructions given is called a *program*.

These instructions or programs must be given in a precise, formal language which is referred to as a *programming language*. **Java** is one such programming language.

Before a program can be run, we require another program to translate the program from the language we wrote in (say Java or Pascal) to a language the underlying hardware understands, i.e. a *machine language*. While this is not a one step process, for simplification we consider it as such. This program is called a *compiler*. Another class of programs with similar goals is an *interpreter*, which translates program statements directly into actions without the need for compilation to a machine language. This is also referred to as *program execution*.

Java uses both a compiler and an interpreter to execute its programs. In the first stage, the Java compiler translates the program listing to a intermediate *bytecode*. This bytecode is in the form of coded instructions a virtual hardware machine called the **Java Virtual Machine (JVM)** understands. The Java interpreter then runs this bytecode according to the specification of this virtual computer of sorts (the JVM) and we have our program execution.

1.2. Advantages of Java

Since Java is compiled to the JVM bytecode specification, it means that a Java program will run on any architecture or operating system where a Java interpreter exists. This is different from the typical execution flow of compiled languages like Pascal or C, because their source code has to be recompiled to target whichever underlying machine exists. The reason for this particular design choice was the fact that one of Java's original goals was to be the programming language for varied consumer devices, not just personal computers. Since these devices would not be the ideal computational machines for doing programming and compilation, the concept of a virtual machine to run intermediate bytecode became necessary.

Java also uses a familiar C/C++ style syntax which is widely used. Java itself has become one of the most popular languages since its inception in the mid-90s. The good side effect is that there are a huge number of jobs, books and reusable code available for the Java programmer.

1.3. Getting Java

When you go to download Java, you are faced with two variants of it: the **Java Development Kit (JDK)** and the **Java Runtime Environment (JRE)**. The former is used by people wanting to write programs in Java whereas the latter by those who just

wish to run Java programs. Thus for the purpose of this book, we will be needing a JDK.

The JDK itself comes in 3 *editions*.

Java SE Java Standard Edition

Java EE Java Enterprise Edition

Java ME Java Mobile Edition

We will be focussing on Java SE since the intended target machines for this edition is personal computers. As of the date of this text, the current version of Java is Java 7 but any version upwards of Java 6 should serve you fine. While the step-by-step installation of Java SE for all the major operating systems is beyond the scope of this book, it is a fairly simple procedure much like the install process of any other package on your choice of OS. You can download your copy of Java SE from the URL below.

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Another thing you would need is a good text editor for editing Java source code. Choose any editor that you are comfortable with, preferably one with syntax highlighting (colors different words in source code with special meaning).

1.4. Writing your first Java program

The first program we are going to try out will simply print out the words "Hello World!" on the screen. This is a common first program to teach and is a good way to learn about basic program structure. Fire up your editor and enter the text given below saving the file as ***HelloWorld.java***.

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

Listing: the source code of our Hello World Program

Note the case (capitalization) of certain words in the program. Also note how we have *indented* certain lines to give the appearance of a nested structure to our listing. While such indentation is not necessary in Java (to a compiler), it is done for human readability and is not considered optional by other programmers.

To run this program, we go through a two step process. Firstly we need to compile the program using the Java compiler *javac* and then run it through the interpreter which is called simply *java*. Look at the commands entered below alongwith the output of our program shown on the last line. Run the same on a console (command prompt) using your choice of directories and remember to save your program in the same directory.

```
d:\code\experiments\java> javac HelloWorld.java

d:\code\experiments\java> java HelloWorld

Hello World!
```

Figure: output of our Hello World program

If you enounter an error like *´javac is not recognized as an internal or external command´*, you have probably not added your JDK *bin* directory to your **PATH** variable. Check your OS manual or check online documentation on how to achieve this. If the first step ran successfully, you would have a file name **HelloWorld.class** in your directory. Be careful not to include the *.class* extension when running the program through the Java interpreter in step 2. If all goes well, you should see the text printed on the command line. Congratulations!

1.5. Analyzing your first program

A full explaination of all the components of this program listing goes beyond the scope of this chapter. For now it is important to study it as a taste of what Java programs look like. The first line declares the title of the program as ***HelloWorld*** and this must be done in the same casing (uppercase and lowercase) as the name of your *.java* file.

The second line declares the ***main***, which is where the program starts running. Notice how this is within the program title's curly braces. It also has its own set of curly braces, and within it we encounter the line which actually produces our desired output. The ***println*** is a way to tell Java to output something on the screen. We terminated this line with a semicolon and finally wrapped up our code listing with the closing curly braces which we had opened before.

Syntactically Java resembles the C/C++ family of programming languages. It is from them that it gets the concept of terminating a statement with a semicolon. Nesting of curly braces define heirarchy and a set of curly braces are used to combine multiple statements into a block. A program terminates when the last statement of the *main* completes its execution.

2. DATA TYPES AND VARIABLES

Computers are data processing machines. Their operations revolve around transforming data into meaningful information. For a programming language like Java to be able to act as a medium for such a transformation, it must understand how to work with different kinds of data.

2.1. Literals

Any data that you enter directly into your code is called a **literal**. That means that in our first program, the bunch of characters "Hello World!" was a character literal. Literals can also be numeric in nature.

The point of a literal is to give a formal name to *data* that you enter into the source code of the program, without it becoming a part of the vocabulary of Java as a programming language.

2.2. Variables

A **variable** is a container for data. We assign a purposeful name to some data that we wish to refer to in other statements of our program, and this binding of data to a name is called *variable* assignment. Consider the variable assignments given below.

```
a = 10
firstName = "Alan"
lastName  = "Turing"
amount = 103.55
```

Listing: Some variable assignments

It should be noted that the above are not valid statements in Java. However, these pseudo code statements illustrate the concept of variables nicely. While the first variable name '***a***' is not descriptive, we can see that it refers to a numeric value. The other variable names are more purposeful and store characters and even numbers with a decimal point in them. What would make these valid Java statements is a few syntactic rules and the important declaration of what is the *type* of data the variable will store.

2.3. Data Types

A ***type*** is a way to tie together the data of a program to the actual storage details of such data. The Java compiler must know what kind of data it will be dealing with for every variable. This way it will be able to allocate the correct amount of memory to store such variables. And so we have to explicitly declare our variables with data types in Java.

Java has numeric data types like *(int, float, long etc.)*, character *(char)* data type and a special type called *boolean* which only stores true or false values. To illustrate all these concepts better, we will now look at a program incorporating variables to calculate simple interest given the rate of interest, time period and the principal amount.

```
class SimpleInterest {
    public static void main(String[] args) {
        /*  Simple Interest Program
            SI = P * R * T
            P: Principal Amount
            R: Rate of Interest
            T: Time Period
        */

        float amount   = 100f;
        float rate     = 0.06f; //Interest is 6%
        int timePeriod = 2;     //Assuming 2 years

        float simpleInterest = amount * rate * timePeriod;

        System.out.println("Interest: " + simpleInterest);
    }
}
```

Listing: Calculating simple interest using numeric variables

The program presents many interesting and hitherto unknown facets of Java. The first being the use of comments in the program to convey some meaningful information to the reader of the source code. The Java compiler and interpreter suitably ignore comments. There are two ways to write comments in your program. Multi-line comments start with **/*** and end with ***/**. Single line comments start with **//** and continue till the end of the present line.

The basic structure of our program remains the same. We define the title of our program on the first line followed by the *main* method. In here we define our three variables - *amount, rate* and *timePeriod*. Notice that the data type for the first two is ***float*** signifying that these are floating point numbers, i.e. numbers with a decimal point. Such numbers are written with a suffix of **f** to their literal values. The *timePeriod* variable is declared as an ***int***, which stores whole numbers without any decimal point. Here we are assuming that this variable's value can never be a fractional number. Such decisions on which data type to pick for a computation variable are a regular part of a programmer's job.

Our comments at the top specify the formula we are going to use to calculate the simple interest. It is a good idea to note down such details along with the purpose of a block of code for readability purposes. Since we are dealing with the multiplication of floating point numbers, it makes sense to store the result also in a float. We multiply our three input variables using the multiply ***operator*** *******. We then print out the result using *println*. Run this program using the same two step procedure we used in the last chapter and you should see the output as given below.

```
Interest: 12.0
```

You would notice that we used another operator, namely **+**. This *concatenated* the word 'Interest:' with our result. However the same operator when used for numeric data types performs an addition. Thus we deduce that operators are context sensitive to the data types they are working upon.

3. INTRODUCING CLASSES & OBJECTS

3.1. Abstraction

A computer is a complex electronic machine. At its heart is a microprocessor that performs the computation. The data for a computation is composed of strings of 1s and 0s. Electronically these states of 1s and 0s are represented by current pulses. But to perform everyday tasks on a computer like word processing or sending emails, you do not have to know the details of the entire machine or network. This hiding of unnecessary details is called ***abstraction***.

Abstraction is a powerful programming concept too. The computer does not understand directly what you wish to do when you say something along the lines of `System.out.println()`. This has to be translated through a series of steps to instructions that would get you the output. Thus programming languages like Java are mediums of expressions providing us with the necessary level of abstraction to solve a problem.

3.2. Defining classes and objects

In the previous chapter we saw data types and variables combining to give us the ability to hold literal values. When you think about it, this is nothing but an abstraction of storing a value at a memory location with the variable name being a way to access this memory location. The data types we saw were simple and are called *primitive data types*.

But what if the kind of data you wish to hold cannot be represented elegantly by the primitive types? For example a date can be thought of as a combination of 3 integer types, one each for day, month and year. If programming is all about defining a series of abstractions and operations on data, it is important that our data be represented closer to its true meaning. Like a date being manipulated as a date rather than independantly as a set of 3 numbers. Such composite variables are called ***objects*** and their *type* is called a ***class***.

We said previously that the type of a variable determines what operations are valid on it. Since with classes we are creating new data types, we have to define a set of valid operations on them. A class thus becomes a combination of data types and the operations that are applicable on this new data type. An object becomes a variable of a class and is said to be an instance of its class.

The type variables which form the data being represented in the class are called the *data members* or *member variables* or simply ***members***. In the date example, the three integers to represent day, month and year are the members of that class. The operations defined for a class are called its ***methods.***

3.3. Creating a class in Java

We will pick up the example discussed in the previous section as an example for creating our own version of a *date* class in Java.

```
class MyDate {

    // Member variables
    int day;
    int month;
    int year;

    // Methods

    // Initializes members to some value
    void initDate() {
        day   = 10;
        month = 12;
        year  = 1815;
    }

    // Display the value stored
    void displayDate() {
        System.out.println(day + "/" + month + "/" + year);
    }
}
```

Listing: Our class for a simple date object

Save this file as ***MyDate.java***. Notice that we have some member variables in there (*day, month, year*) and two methods (*initDate, displayDate*) but no *main* method. What this file (more appropriately class) stores is *attributes* for a simple date data type and some simple functionality to set and display the date.

Note: The astute reader will no doubt be interested with the date chosen - 10th of December, 1815. This happens to be the birthdate of Lady Ada of Lovelace. She is considered to be the world's first programmer. For more information read her biography 'Ada, the Enchantress of Numbers' by Betty A Toole [1998, Strawberry Press].

Since execution starts in the *main* method, we will create a separate file which will create an object of the *MyDate* class and call its methods. Let us call this file ***DateRunner.java*** and the code for it is given below.

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

        //Create a MyDate object
        MyDate ada;
        ada = new MyDate();

        ada.initDate();
        ada.displayDate();
    }
}
```

Listing: DateRunner class containing main

We will now compile both these source files using *javac* as shown below.

```
javac MyDate.java DateRunner.java
```

The next step is to run the application using the *java* interpreter. Remember that only the *DateRunner* class contains the *main* method. It is the responsibility of this class to invoke other classes by creating their objects. Thus we run only the *DateRunner* through the interpreter to get our output.

```
java DateRunner
=> 10/12/1815
```

4. CONDITIONAL STATEMENTS

Making decisions are a fundamental part of programming. There are numerous conditional constructs available in Java to alter the flow of a program based upon a decision outcome you decide upon. To arrive at such an outcome, one has to first perform a *comparison* operation.

4.1. Comparison Operators

The comparison operators of Java are not that different from other programming languages you may have come across, or even elementary mathematics. For example, a greater-than operator is given a symbol of > and it checks if the value on the left hand side expression is greater than that of the right hand side. These operators return a *true* or *false* value. Thus it makes sense to store the result of such a comparison as a ***boolean*** variable.

Refer to the table given below for a detailed listing of the comparison operators available.

Operator	Description
>	Returns true if the left operand is greater than the right operand
<	Returns true if the right operand is greater than the left operand
==	Returns true if the right operand is equal to the left operand
!=	Returns true if the right operand is not equal to the left operand
>=	Returns true if the left operand is greater than or equal to the right operand
<=	Returns true if the right operand is greater than or equal to the left operand

4.2. If-Else Conditional

A simple conditional construct in Java is an **if-else** statement. If a condition is met, the block following the **if** is executed, otherwise the one following **else** is executed. A general syntax for this construct is given below.

```
if (<condition>) {
<Execute statements here if condition is met>
}

else {
<Else execute these statements>
}
```

Figure: the general syntax of an if-else statement

It should be noted that it is optional to have an *else* clause following an *if*. You can even omit the curly braces signifying block boundaries if you have to only execute one statement if a condition is met. However in the interest of readability, it is wise not to omit them. Let us see an example using the this conditional statement.

```
int temperature = 102;
String state;

if (temperature < 0) {
    state = "Solid Ice";
}
else if (temperature > 100) {
    state = "Vapour";
}
else {
    state = "Liquid Water";
}

System.out.println (state);
```

Listing: Using an if-else clause to change program output

In the example given above, we change our output to the state of water given its temperature in degrees celsius. Notice that we have also introduced an ***else if*** clause to determine whether the temperature is sufficiently high enough to turn water to vapour. Feel free to modify the value of the *temperature* variable to see the change in output.

4.3. Switch Conditionals

A ***switch*** conditional is a multi-way selection statement. If your *if-else* clause is becoming too unwieldy, it is a good idea to consider using a *switch* in its place. As an example, suppose we wish to simulate the operations on an ATM machine by allowing the inputs to be characters which are bound to particular action. Our specification is as given below.

D	Deposit amount
W	Withdraw amount
B	Check balance
M	Print a Mini-Statement

Turning this into an *if-else* construct becomes needlessly verbose.

```
char choice;
 ...

if (choice == 'D') {
    System.out.println("Deposit an amount");
}
else if (choice == 'W') {
    System.out.println("Withdraw an amount");
}
else if (choice == 'B') {
    System.out.println("Check Balance");
}
else if (choice == 'M') {
    System.out.println("Print a Mini Statement");
}
else {
    System.out.println("Wrong choice");
}
```

Listing: our ATM action specification modelled using if-else

We can use a ***switch*** here to make our code shorter without losing any details.

```
char choice;
 ...

switch (choice) {
    case 'D': System.out.println("Deposit an amount");
              break;
    case 'W': System.out.println("Withdraw an amount");
              break;
    case 'B': System.out.println("Check Balance");
              break;
    case 'M': System.out.println("Deposit an amount");
              break;
    default : System.out.println("Wrong choice");
}
```

Listing: our ATM action specification modelled using switch

Notice the use of the ***break*** statement after each *case*. This causes the execution to jump to the end of the *switch*. A *break* is optional, but if it is missing, execution continues to the following *case* statements till a *break* is enountered. The ***default*** statement executes if no match is encountered.

From the data types we have seen until now, a *switch* would typically operate upon *characters* and *integers*. With the release of Java 7, it now supports *String* objects for comparison.

5. ITERATION STATEMENTS

An ***iteration*** or ***loop*** repeats a statement or a block of statements till the point an entry condition is satisfied. It saves us from the tedium of writing similar statements again and again by giving us constructs that allow us to acheieve the same result in a compact manner. Java has three main iteration constructs: *while*, *do-while* and *for*.

5.1. The while loop

The ***while*** loop is a simple iteration construct which first tests a boolean condition and if true, executes a block of statements. The general syntax of a *while* loop construct is given below.

```
while (<boolean expression>) {
    // statements
}
```

Figure: general syntax of a while loop

First the boolean expression is evaluated. If it results in a *false* value, the execution jumps straight to the end of the block and executes the first statement after the block. However if the expression evaluates to a *true* value, the body of the block is executed and then the expression is reevaluated. This cycle (of evaluation and then block execution) continues till the boolean expression remains *true*. Let us use this construct to find the sum of the first twenty(20) natural numbers.

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

    int ctr = 1;
    int sum = 0;

    while(ctr <= 20) {
      sum += ctr;
      ctr++;
    }

    System.out.println("Sum: " + sum);
  }
}
```

Listing: a program to find the sum of the first 20 natural numbers using a while construct

Note the entry condition of the *while* loop and it's execution block contents. We start our counter from 1 (the first natural number) and the block will be executed till the counter remains less than 20 or equal to 20. In the execution block, we *increment* the counter with the ++ operator which adds 1 to the value of the variable.

5.2. The do-while loop

This construct is similar to the *while* loop with one difference. The boolean expression which forms the conditional part of the loop is written at the end, thus ensuring that **the loop will execute atleast once**.

```
do {
    // statements
} while (<boolean expression>);
```

Figure: general syntax of a do-while loop

5.3. The for loop

A ***for*** loop consists of three parts - *initialization, condition* and *step*. Here the variable doing the counting is initialized as a part of the construct itself. At the end of each iteration, *stepping (increment/decrement)* is done followed by checking of the condition.

```
for(<initialization>; <boolean condition>; <step>) {
    // statements
}
```

Figure: general syntax of the for loop construct

To understand this construct better, let us rewrite our previous example of finding the sum of the first 20 natural numbers using a *for* loop.

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

    int sum = 0;

    for(int ctr = 1; ctr <= 20; ctr++ ) {
      sum += ctr;
    }

    System.out.println("Sum: " + sum);
  }
}
```

Listing: usage of the for loop construct

In the example above, both the initialization of the *ctr* variable and its increment are a part of the *for* loop definition. The output is the same as we achieved using the *while* loop, but this version was shorter. It is a matter of personal preference and judgement on the kind of problem presented which would ultimately decide which loop construct one picks.

5.4. break and continue

We have already seen ***break*** when we were discussing the *switch* statement. In this section, we will talk about *break* and ***continue*** in more detail for their usefulness as loop flow controllers.

Both these statements are used for jumps in execution flow of a loop block. A *break* exits the loop block without executing the remaining statements. On the other hand, a *continue* stops the execution of the current iteration only. Execution again picks up from the next iteration. To clarify these flow jumps, let us look at an example.

```
int ctr = 0;
int sum = 0;

while(true) {
  ctr++;

  if (ctr == 20)
    break;

  if ((ctr % 2) == 0)
    continue;

  sum = sum + ctr;
}

System.out.println("Odd sum: " + sum);
```

Listing: using flow jump constructs

The objective of this program is a little different than the previous ones. It will print out the sum of only odd natural numbers below 20. Note that while this might not be the best way to achieve our objective, it does show a few interesting things. First, note how we used the *while* loop entry condition. The use of *true* ensures that the loop will run forever unless we explicitly *break* out of it. This is exactly what we do if we see that our counter is equal to 20. Another thing to note here is that an *if* conditional does not need curly braces for its block if there is only one statement to execute. In this case, it is a *break*.

Since we are only concerned with odd numbers, we do a *continue* before we increment the sum variable. This ensures that the next iteration starts but the even counter value is not included in the sum.

6. ARRAYS

An ***array*** is a set of variables of the same data type. Let's say you wish to store a the first 10 odd numbers. To associate the list of these with variables you can either associate each one with its own separate variable name - *oddNum1, oddNum2, oddNum3* and so on, or you can store them all in a single array. Each of these numbers become an array ***element***.

6.1. Array Initialization

Arrays can be declared using the *indexing operator* denoted as []. This is to be placed either after the data type identifier of the array or its variable name. The following two declarations of an integer array are equivalent.

```
int[] oddNums;
int oddNums[];
```

While the latter declaration might be familiar to C/C++ programmers, in this text we would adopt the former style. I believe it is clearer in indicating the type of the variable as an array of integers.

The declarations that we saw above just associate the variable identifier with the type of an array. It does not allocate any storage in memory for an array. To do that we write an array *definition* using the ***new*** keyword.

```
int[] oddNums = new int[10];
```

When using the *new* keyword to declare an array, you must specify the number of elements in the array by writing it within **[** and **]**. Each element of the array is given an initial value depending upon the type of the array. For numeric types like in the example above, the value assigned is 0 (zero).

We can also initialize values for arrays at the time of creation using curly braces { }. Each value is placed between these separated by a comma. To clarify this, let us look at an example below.

```
String[] designCommittee = { "Backus", "Bauer", "Bottenbruch", "Katz",
"Perlis", "Rutishauser", "Samelson", "Wegstein" };
```

We initialize an array called *designCommittee* of type *String*. Though we have not discussed this type yet, for the time being think of it as a data type holding words or a bunch of characters as a single entity. Thus, our variable is a *String* type array with 8 elements as initialized. Note that we did not have to use the *new* keyword.

For the curious, the names chosen as values for the designCommittee array are the members of the Algol 58 programming language design committee. Their decisions impacted programming languages for decades to come. Java itself is considered a descendant of the Algol family syntax style.

6.2. Using Arrays

To store values at an array location we simply write the element number, also known as an **index** or **subscript** in between the square brackets [] immediately following the array variable name. In Java, array indexes begin at 0 (zero). This means that the first element is stored at index 0, the second at index 1 and so on. Let us look at an example to clarify our concepts regarding the usage of arrays.

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

        // Initialize a 10 element array
        int[]oddNums = new int[10];

        // Will will go through the loop 10 times
        for(int ctr = 0; ctr < oddNums.length; ctr++) {
            oddNums[ctr] = (ctr * 2) + 1;
        }

        System.out.println("Last element: " + oddNums[oddNums.length -
1]);
    }
}
```

Listing: using arrays to store odd numbers

When you execute this program, you would see an output like *Last element: 19*. So what actually happened in the program? We declared an integer array of 10 elements called *oddNums*. The expression <*array*name>.length provides you with the length of the array, which in this case would be 10. Array indexes are integers, but you can specify expressions whose output is an integer too.

To find 10 (the length of our array) odd numbers, we initialize a counter to iterate with and initialize it to 0. The Nth odd number can be calculated simply by the formula,

Nth odd number = ((N - 1) x 2) + 1

Since array indexing starts at 0 which also is out initial counter (*ctr*) value, this formula can be reduced simply to

```
(ctr * 2) + 1
```

Finally we print out the last element of our array by calculating its index using *oddNums.length - 1*.

6.3. The enhanced for loop

The ***enhanced for*** or ***for-each*** loop was added in Java version 5, to iterate over arrays and other composite types which we will discuss later. The 3-part for loop, also called the C-style for loop, is somewhat verbose when you want to simply iterate over the array. Let us see an example of each of these constructs and compare the difference

in their syntactical clarity.

```
public class ForOld {
  public static void main (String[] args) {
    String[] designCommittee = { "Backus", "Bauer", "Bottenbruch",
"Katz", "Perlis", "Rutishauser", "Samelson", "Wegstein" };

    for (int ctr = 0; ctr < designCommittee.length; ctr++) {
      System.out.println(designCommittee[ctr]);
    }
  }
}
```

Listing: Using the C-style for loop

The objective of the program is fairly simple. It creates a String type array and then iterates over it, printing out the element values. We initialize our counter variable to achieve this iteration, check whether our counter has not yet reached the end of the array (using *length*) and increment the counter in each iteration. The same can be achieved using the enhanced for loop as below.

```
public class ForNew {
  public static void main (String[] args) {
    String[] designCommittee = { "Backus", "Bauer", "Bottenbruch",
"Katz", "Perlis", "Rutishauser", "Samelson", "Wegstein" };

    for (String committeeMember : designCommittee) {
      System.out.println(committeeMember);
    }
  }
}
```

Listing: using the enhanced for loop

Note the structure of the *for* loop in this case. Our iteration loop has been reduced to the general structure:

```
    for (<type> <variable> : <collection>)
```

We can then count on using the variable identifier (*committeeMember*) inside the loop block without using array subscripts.

7. METHODS

In chapter 3, we were introduced to classes and objects. But we haven't been truly doing *object oriented programming*. We will now begin to explore in earnest, objects and their fundamental building blocks - methods.

Recall that a ***method*** is an operation of a class. The concept is closely related to a *procedure* or *function* which is a block of statements performing a task. A method however is always declared inside a class. The general syntax of a method is given below.

```
AccessModifier ReturnType MethodName (optional parameter list) {
 . . .
}
```

Figure: general syntax of a method

The access modifier controls the *visibility* of a method, giving the programmer control over where all a method can be called from. We will be going over this concept a bit later. The *return type* is the data type of the return value. Think of it as a result or output of the computation done in the method. The inputs to the method are its The*parameters*. method signature.

7.1. Parameters & Return Values

Let us begin by rewriting our simple interest calculation as a separate method, rather than putting everything in *main*.

```
public class SimpleInterest {

    // Member variables
    float amount;
    float rate;
    int timePeriod;

    // Methods
    public void initValues (float principal, float rateOfInterest, int
numOfYears) {
        this.amount     = principal;
        this.rate       = rateOfInterest;
        this.timePeriod = numOfYears;
    }

    public float calcInterest() {
        float simpleInterest = amount * rate * timePeriod;
        return simpleInterest;
    }
}
```

Listing: the class for calculating simple interest on an amount

We create a class called *SimpleInterest* which does not contain the *main* method. But what it does contain are methods to initialize values needed to calculate the simple

interest and the calculation method. We intend to create a separate runner class which would create an object of *SimpleInterest* and then call its methods.

```
public class InterestCalculator {
    public static void main(String[] args) {
        SimpleInterest applicant = new SimpleInterest();
        applicant.initValues(100f, 0.06f, 2);

        float amountReturned = 100f + applicant.calcInterest();
        System.out.println("Amount after time period: " + amountRe-
turned);
    }
}
```

Listing: the runner class containing main

We create an object of the class *SimpleInterest* and initialize its member values by passing ***arguments*** in the method call - *initValues*. These arguments of the method call pass their values onto the ***parameters*** in the method signature.

We also see that our method *calcInterest* returns a *float* value. Thus in the method definition we explicitly state that it will do so, in contrast to *initValues* which does not return anything. Recall from the general syntax of a method definition that the second modifier tells the type of value a method returns. A method that has only side effects without a return value, is declared as ***void***.

Inside a method definition, we use ***this*** as to refer to the calling object. It is commonly used to access the member variables of a class. In our *initValues* method, we used *this* to differentiate between the parameters and the instance (member) variables. While the names of the two kinds of variables are different in this case, they can have the same name, making the use of *this* essential.

7.2. Constructors

A ***constructor*** is a special method that is called when an object is created using *new*. It has the same name as the class it is contained in and does not have a return type. The purpose of a constructor is to perform some initializations like setting values of some instance variables.

When we do not write a constructor, as we haven't until now, Java automatically creates a no argument constructor which is invoked when an object is created. In our previous example, consider the *SimpleInterest* class with the *initValues* method. We can replace this explicit call to a method by creating our own constructor. This makes sense because the purpose of *initValues* is close to the charter of responsibilities of a constructor.

```
public SimpleInterest (float principal, float rateOfInterest, int
numOfYears) {
        this.amount     = principal;
        this.rate       = rateOfInterest;
        this.timePeriod = numOfYears;
}
```

Listing: the SimpleInterest constructor

Our runner class *InterestCalculator* still makes a reference to a no-argument constructor and the *initValues* method. We modify both these lines by removing the line calling *initValues* altogether and changing the object creation line to the following.

```
SimpleInterest applicant = new SimpleInterest(100f, 0.06f, 2);
```

Our program now compiles and runs as before.

8. INHERITANCE

Inheritance is the process by which we model entities in our problem domain as a hierarchy to lead us to a more conceptually elegant solution structure. This definition is a bit heavy, so we attempt to simplify it. Think of it as creating parent and child *classes* where it makes sense to do so.

Consider a class called *Employee*. A company may wish to differentiate between hourly wage employees and full time employees. The solution to model this hierarchy is two make two child classes of *Employee*, namely *HourlyEmployee* and *FullTimeEmployee*. The common behavior (like all employees have a name) between them can be shared by defining it in the *Employee* class. This is called the ***base class***. The other two classes which inherit the properties and behaviors from the base class are called the ***derived classes***.

8.1. Extending a class

To create a subclass from an existing class, we use the ***extends*** keyword in Java. We can add or modify properties and behavior of existing classes, which keeps redundancy to a minimum.

```
public class <child-class> extends <parent-class> {
 . . .
}
```

Figure: general syntax for extending a class

Keep in mind that while a class can have multiple child classes, a class cannot have multiple parents in Java. It is however possible for a child class to act as a parent of another class. Let us now see an example of inheritance in Java.

```
public class Interest {

    // Member variables
    float amount;
    float rate;
    int timePeriod;

    public float calcInterest() {
        return 1.0f;
    }
}
```

Listing: the base class - Interest

```
public class SimpleInterest extends Interest {

    // Methods
    public SimpleInterest (float principal, float rateOfInterest, int
numOfYears) {
        this.amount     = principal;
        this.rate       = rateOfInterest;
        this.timePeriod = numOfYears;
    }

    public float calcInterest() {
        float simpleInterest = amount * rate * timePeriod;
        return simpleInterest;
    }
}
```

Listing: the derived class - SimpleInterest

Note that in *SimpleInterest*, we did not redefine the member variables but used them directly in our constructor for *SimpleInterest*. We did however, create a more specialized version of the *calcInterest()* method in the subclass. Our task now is to create a suitable runner class containing the *main* method to test our classes.

```
public class InterestCalculator {
    public static void main(String[] args) {
        SimpleInterest applicant = new SimpleInterest(100f, 0.06f, 2);

        float amountReturned = 100f + applicant.calcInterest();
        System.out.println("Amount after time period: " + amountRe-
turned);
    }
}
```

Listing: calculating the simple interest using the derived class

A useful feature given to us is that while creating an object we can specify its type as its parent class. This makes sense because the object, regardless of which child class it instantiates, would atleast have the behavior (think method) of its parent. Thus while creating the *applicant* object, we can use the statement below.

```
Interest applicant = new SimpleInterest(100f, 0.06f, 2);
```

8.2. Access Specifiers

An access specifier controls the access and visibility of its member. The member here means a class member field or a class method. There are three access specifiers available in Java, namely ***public, private*** and ***protected***. You have already used the *public* access specifier for the classes you have been writing so far.

A member designated as *private* can only be accessed by a method of its own class. A *public* member however can be accessed by any member of any class. A good general rule is to mark your member variables *private* and methods *public* unless the method is

only meant to be used internally by other methods of the class. Good programming principles encourage us to use methods to read and update member variables through *public* methods called BI getters and ***setters*** respectively. This allows us the flexibility to impose sensible restrictions on the values that a member can take.

```
private float principal;

public float getPrincipal () {
    return principal;
}

public void setPrincipal (float deposit) {
    if ( deposit < 0 ) {
        System.out.println("No negatives allowed");
        System.exit(0);
    }
    else {
        principal = deposit;
    }
}
```

Listing: getter and setter for principal

With respect to inheritance, access specifiers play a special part. *Private* members are not interited whereas *public* members are inherited.

8.3. The super Keyword

We have seen creating methods with the same name in the inherited subclasses, effectively overriding the super class version of it. The ***super*** keyword allows us to invoke the methods of the super class in the subclass. This is useful when the method in the derived class has just some additions to make to the existing computation in the parent class method.

```
class Cylinder {
    protected float radius;
    private float height;

    public Cylinder (float r, float h) {
        this.radius = r;
        this.height = h;
    }

    public float surfaceArea() {
        float sa = 2 * 3.14159f * radius * height;
        return sa;
    }
}

class Can extends Cylinder {
    public Can (float r, float h) {
        super(r, h);
    }

    public float surfaceArea() {
        float sa = super.surfaceArea() + (2 * 3.14159f * radius *
radius);
        return sa;
    }
}

public class SurfaceAreaRunner {
    public static void main(String[] args) {
        Can myCan = new Can(2.0f, 10.0f);
        System.out.println("Surface Area: " + myCan.surfaceArea());
    }
}
```

Listing: Using the super keyword to refer to the superclass members

The example above makes a *Can* class as a derived class of *Cylinder*. A can, we assume is a cylinder covered with both sides. It's surface area thus being the area of its cylinder and twice the area of any one of its (two) covers.

We use the *super* keyword twice here. Once when we calculate the surface area of the can, where we add the surface area of its covers to the already defined surface area of its cylinder. The other is when we call the super class constructor using simply *super()*. Though we did not add anything in the derived class constructor we could have kept some checks (like radius and height cannot be negative) and then called the super class constructor. Another point to note is the use of access specifiers for *radius* and *height*. The *radius* is given a specifier ***protected*** so that it behaves like a private member with one exception, that it is accessible to subclasses. We need it in this case because the subclass would calculate the surface area using it. The *height* on the other hand can remain ***private*** since the subclass does not need it for it's computations.

9. ABSTRACT CLASSES AND INTERFACES

9.1. Abstract Classes

While making inheritance hierarchies, we sometimes come across a case where it does not make sense to allow a superclass to be instantiated, i.e. have objects of their type. For example we should not allow instantiation of a superclass called *SpaceCraft* but derived classes like *Starship* or *Battlestar* should allow object creation. Without fleshing out the details of a space ship, which in this case is only possible in the child classes, it makes little sense to have a concrete implementation for it.

Such classes are termed as ***abstract classes*** and are useful only to serve as a template or contract by which all its derived classes must adhere to. Their utility is derived by their extension. Since they serve as blueprints, these classes must have at least one ***abstract method***, which are simply methods without any implementation details. They only contain the method name and their argument list.

We declare a class or method to be abstract by prepending the keyword *abstract* to the method or class name.

```
abstract class SpaceShip {
    void turnOnIgnition {
        // Some code
    }

    abstract void takeEvasiveManeuver(String direction);
}
```

Listing: an abstract class containing an abstract method

The abstract class *SpaceShip* contains a concrete method - *turnOnIgnition()*. This means that the implementation of this method is also fleshed out in this class unlike the abstract method - *takeEvasiveManeuver()*. Whether we override the concrete method is upto us, but we must give a implementation to the abstract method in our subclass.

9.2. Interfaces

An ***interface*** is like an abstract class containing only abstract methods. It's purpose is to serve as a pure blueprint for other classes and contains no implementation details. Classes *implement* an interface(s) rather than extending it. We declare an interface using the *interface* keyword and listing the method signatures with no implementations.

```
interface Interest {
    float getFinalAmount(float principal);
    void calcInterest ();
}
```

List: creating the Interest interface

We can now create a class *SimpleInterest* which will implement the *Interest* interface, meaning it will provide the implementation of both the methods listed in the interface.

```
class SimpleInterest implements Interest {

    // Member variables
    float amount;
    float rate;
    int timePeriod;
    float simpleInterest;

    // Methods
    public SimpleInterest (float principal, float rateOfInterest, int
numOfYears) {
        this.amount     = principal;
        this.rate       = rateOfInterest;
        this.timePeriod = numOfYears;
    }

    public void calcInterest() {
        this.simpleInterest = amount * rate * timePeriod;
    }

    public float getFinalAmount(float principal) {
        float finalAmount = principal + this.simpleInterest;
        return finalAmount;
    }
}
```

Listing: implementing the Interest interface

We can now create an object of the type *Interface* to instantiate *SimpleInterest* because the latter implements the former and it would satisfy the contract of its parent, just like in parent classes.

```
public class InterestRunner {
    public static void main(String[] args) {
        Interest savings = new SimpleInterest (100f, 0.08f, 2);
        savings.calcInterest();
        System.out.println(savings.getFinalAmount(100f));
    }
}
```

Listing: using the implemented interface as a type

9.3. Multiple Inheritance and Interfaces

Multiple inheritance occurs when a derived class inherits its members from two or more super classes. It can potentially lead to ambiguity when the derived class is supposed to inherit a behavior defined differently in both its super classes. Whose behavior would get preference? The designers of Java thought that multiple inheritance was more trouble than its worth and thus decided to remove it from the language.

They did this by allowing a class to extend one and only one super class. But if you wanted to adhere to multiple contracts, how would you achieve this? By *interfaces*. While you are allowed to derive from only one parent class, even if it is abstract, you can implement as many interfaces as you want.

```
class TravelGuide extends Guide implements HotelList, RestaurantList {
}
```

Listing: Implementing multiple interfaces

10. EXCEPTION HANDLING

Error handling is an important part of programming. Errors can be generated at compile time, like when you invoke *javac*, or at run time like when you try diving by zero. In Java, an abnormal condition at run time is called an *exception*. Syntactical errors like missing a semicolon are caught at compile time by the Java compiler and it does not let you proceed further till you fix those errors. Run time exceptions are usually logical in nature.

We make our programs reliable by writing *exception handling* code. The Java compiler enforces certain exception handling. Also we do not need to write the handling part exactly where the error occurred, but this can be separated out to ensure code flow readability. For this we need to be able to *throw* our exceptions to the handler.

10.1. The try-catch block

A ***try-catch*** block enables exception handling. Any code that may throw an exception is placed inside the *try* block and the *catch* takes remedial action if possible. Consider a trivial division program as given below.

```
class ExceptionalCalc {
    public static void main (String[] args) {
        int numerator   = 10;
        int denominator = 0;

        int result = numerator / denominator;
        System.out.println(result);
    }
}
```

Listing: a simple exception generating program

Even before we run this program, our mathematical senses tell us that unless the computer (or more specifically, the Java interpreter) can represent infinity on a monitor, we should be seeing an exception.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionalCalc.main(ExceptionalCalc.java:6)
```

Figure: output showing the divide by zero exception

We now introduce a try-catch block in this program to gracefully handle this exception. Note that it is the calculation of the *result* that throws the exception, so it makes sense to put it in a *try* block. We should also keep a note of the type of exception thrown, in this case being *ArithmeticException*, so that we can introduce it in the *catch* type.

```
class ExceptionalCalc {
    public static void main (String[] args) {
        int numerator   = 10;
        int denominator = 0;

        try {
            int result = numerator / denominator;
            System.out.println(result);
        }
        catch (ArithmeticException e) {
            System.out.println("I have caught an exception");
        }
    }
}
```

Listing: the same program with exception handling added

When we run this program, instead of a (not so) cryptic error message, we get the string we printed in our *catch* clause. Here we only gave a simple output indicating to the user that we saw an exception being raised. However in some cases it is possible to take remedial action provided we understand the exception raised and how to handle it.

10.2. Java Exception Classes

You might have noticed that we gave the type of exception being handled as an input to the *catch* clause. Are there other exception type? Yes, in fact *Exception* is a class type and there are many specialized derived class types from it. Let's see what happens when we change the type of exception caught to this superclass.

```
catch (Exception e) {
    . . .
}
```

When we run this source code, we get the same output as before. This is in line with our thinking that the *ArithmeticException* type is nothing but a specialisation of the super type *Exception*. A general rule of thumb however, is to use as specific an exception type as possible. You don't want to take remedial actions thinking something went wrong with a calculation when the arguments supplied are less than expected. We can even give multiple *catch* clauses for a single *try* block to be able to handle multiple kinds of exceptions.

10.3. The finally statement

You will frequently see or create code that will have multiple *catch* clauses and some statements following that. But what if there are a particular set of statements you always wish to execute. Any following statements after the *catch* clauses may not run if you haven't caught the particular type of exception thrown. But if there is some critical operation you must perform before you abort out of your program? This is where the ***finally*** statement comes in.

A *finally* statement defines a block of code that will execute after a *try*, regardless of how the exception handling went. A good example of what should go in a *finally* block is closing of any open files.

```
try {
    // Some vulnerable code
} catch (Exception1 e) {
    // How to handle Exception1
} catch (Exception2 e) {
    // How to handle Exception2
} finally {
    // This will always execute
}
```

Listing: the general syntax of a finally statement

FURTHER READING

This book is only meant as a short introduction to the Java programming language. It is by no means an exaustive reference on the same. I can only hope that reading this text will prepare you well for reading intermediate level books or atleast ease your passage into books that require some programming know-how beforehand. Below is a list of recommended books that should help you further your Java knowledge.

1) *Head First Java* by Kathy Sierra and Bert Bates (2nd Edition, 2005)

Graphical, readable and humorous introduction to Java programming which lays a lot of emphasis on exercises.

2) *Thinking in Java* by Bruce Eckel (4th Edition, 2006)

A thorough and well written introduction to Java and its object oriented nature. Great theoretical explainations.

3) *Learning Java* by Patrick Niemeyer and Daniel Leuck (4th Edition, 2013)

Advanced introduction to Java for programmers familiar with other languages. Well written and exaustive.

APPENDIX: Code Editors and Integrated Development Environments

There are many text editors, both free and paid, available for Java programming today. An Integrated Development Environment (IDE) facilitates faster development by supporting many language specific features, but I would not recommend these in your initial learning phase. This is because of their overly assistive nature which might result in a weakened learning foundation and their own added complexity. Below are some of the editors and IDE's I have used over the years and found useful.

1) EditPlus (ES-Computing)

Lightweight but well polished text editor with many features. Mature and commercial.
http://www.editplus.com

2) Vim (Bram Moolenaar)

Powerful, multi-platform editor with modal editing and lots of plugins. Open source.
http://www.vim.org/

3) Emacs (GNU)

Flexible, scriptable editing environment with many features. Open source.
http://www.gnu.org/software/emacs/

4) NetBeans (Oracle Corp)

Open source IDE with great Java support and nicely integrated features.
https://netbeans.org/

5) Eclipse (Eclipse Foundation)

Powerful, open source IDE with multiple language support and tons of plugins.
http://eclipse.org/

GLOSSARY

Array	A collection of elements with the same data type.
Bytecode	An intermediate code between human written source code and code that the machine understands.
Class	A user defined, composite data type binding both properties and methods into one entity.
Derived Class	A class that inherits its members from another (super) class.
Method	A group of statements that achieve a particular task by operating upon a members of the class.
Object	An instance of a class.
Super Class	A class which serves as a parent to other classes letting its members be inherited.