You remain well-grounded when you know your roots. Same is true about Java. So a look at how it came into existence and where it stands amongst other programming languages...

Before we can begin to write programs in Java, it would be interesting to find out what really is Java, how it came into existence and how does it compare with other computer languages. Also, it is important to know what tools we are going to use for executing programs in this book, from where to get them and how to install them. In this chapter, we would briefly outline these issues.

The Evolution

Approaches to programing keep evolving all the time. These approaches are more or less driven by the computing needs of those times. When these needs cannot be addressed by languages of that era, a need is felt for a new language. These needs have become more and more complex over the years.

In the early days of computing when the need was that a machine should somehow be able to execute instructions, programming was done by manually keying in the binary machine instructions. So long as the programming task was small, programmers were ready to take the pains of keying in instructions in binary.

As the tasks became more complex and the program lengths increased, need was felt for a new language that could make it easier to write programs. That's when Assembly language was invented. In Assembly, instead of binary, small abbreviations were used to write instructions. These abbreviations were nothing but representations of binary instructions. This made life much easier for the programmer. The assembly language programs were very efficient.

As the demands of computing increased, it was felt that learning and using Assembly language are not very easy. To address this need, many languages were invented. These included FORTRAN, BASIC and COBOL. But these turned out to be suitable for specific domains. For example, FORTRAN found widespread acceptance in scientific and engineering applications, whereas, COBOL was typically used for building business applications like payrolls, inventory management, etc.

These languages suffered from three important limitations. They are as follows:

(a) They could not be used apart from the domains that they were supposed to serve. So a change in domain necessitated a programmer to learn a new language.

(b) They could not be used to write system-level code that could interact with hardware easily.

(c) All these languages were not designed around structured programming principles. Hence, in programs of sizeable length it became difficult to follow the flow of control.

As a result, a feeling started growing—could there not be a universal programming language that could address all these three concerns? The answer came in the form of C language. It was invented by Dennis Ritchie at AT&T's Bell Laboratories. Since it was designed by a programmer, and not driven by a committee, it addressed the needs of programmers very well. These included speed, efficiency and brevity. Programmers loved it and it soon became a dominant programming language. This dominance continued for almost two decades.

As new hardware evolved, and computers gained widespread acceptance, demands from the program grew multi-fold. The complexity of programs hit the roof, and this is where C language started showing signs of strain. It simply didn't contain those elements that could handle the complexity of the problem being solved. There was a need for a fresh approach to handle the complexity. This gave birth to a new way of organizing the program, called Object Oriented Programming (OOP). C++ was based on these principles and was invented by Bjarne Stroustrup at AT&T's Bell Labs. 1990 was the decade of C++. Since C++ was built on foundation of C, it became easier for programmers to migrate to this new language quite quickly. It was largely accepted that C++ is a prefect language and there would be possibly no need for a new language. But this belief got dented as you would see in the next section.

The Birth of Java

C and C++ were being used for building most applications till late 1990s. The computing world was more or less divided into three camps—Intel, Macintosh and Solaris. Compliers were available that targeted these microprocessors and created machine language instructions that could get executed on these microprocessors. This was alright for the PC world. However, the microprocessor diversity was too much in consumer electronics world. The microprocessors used in washing machines, microwave ovens and other such devices were so many that creating a full-fledged compiler for each microprocessor was impractical. So a thought started taking shape to create new language that could be used to create software that could run on different microprocessors embedded in various consumer electronic devices. This was the initial motivation that led to the birth of Java.

Thus creation of an architecturally neutral and portable language for consumer electronics devices was the primary factor for Java to come into

existence. However, it gained impetus for a very different reason. World Wide Web and the Internet were growing like wildfire, and its programming needs were similar to those that Java was trying to address. There were all types of machines that were getting connected to the Internet. A language was needed that could be used to create programs that could run on machines connected to the Internet and had different microprocessors and operating systems. Java fitted this bill perfectly, because it was designed from ground up with this motive in mind, namely, platform-independence (portability).

So it may not be an exaggeration to state that had Internet and World Wide Web not caught the fancy of the world at the same time in which Java was growing, Java would have possibly remained a language to be used only in the consumer electronics world.

With that historic perspective under our belt, I think we are well poised to begin learning Java.

What is Java?

Java is a programming language developed at Sun Microsystems in 1995. It was designed by James Gosling. The language derives much of its syntax from C++ (and its predecessor, C), but is simpler to use than C++. Reputation of Java has spread wide and far and has captured the imagination of most software professionals. Literally thousands of applications are being built today in Java for different platforms including desktop, web and mobile.

Possibly why Java seems so popular is because it is reliable, portable and easy to use. It has modern constructs that are required to represent today's problems programmatically. Java, like C++ makes use of a principle called Object-Oriented Programming (OOP) to organize the program. This organizing principle has lots of advantages to offer. We would be discussing this in detail in Chapter 9.

Let us now try to understand how Java achieves portability and reliability.

Traditional Programming Model

When we execute a program on any computing device like PC, Laptop, Tablet or Smartphone, the instructions in it are executed by the microprocessor present in that device. However, the microprocessor cannot understand the instructions written in languages like C, C++ or Java. Hence, these instructions have to be first converted into instructions that can be understood by the microprocessor. These converted instructions are in machine language. This conversion process is known as compilation.

The machine language instructions understood by a microprocessor are often called its Instruction Set. Problem is that instruction sets of different microprocessors are different. Thus, instructions of an Intel microprocessor are different than those of an ARM microprocessor. Therefore, any program being executed on a specific microprocessor needs to be converted into machine language instructions which that microprocessor understands. Thus, for the same program, corresponding machine language instructions would be different for different microprocessors. Hence, if a program is compiled for one microprocessor it may not work on another microprocessor. To make it work on another microprocessor it would have to be compiled for that microprocessor again. This is shown in Figure 1.1.

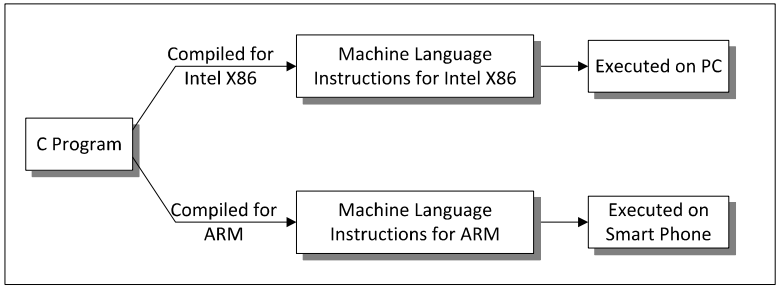

Figure 1.1

Any running program needs to make use of services of an Operating System (OS) during its execution. These include services like performing input/output, allocating memory, etc. You must be aware of the fact that on the same microprocessor, different OS can be used. For example, suppose there are two laptops having same Intel Pentium microprocessor. On one laptop one can run Windows whereas on the other, one can run Linux. But since the way these OSs' offer different services is different, during conversion to machine language these changes have to be accommodated. So for the same program, machine language instructions for Intel + Windows combination would be different than those for Intel + Linux combination. This is shown in Figure 1.2.

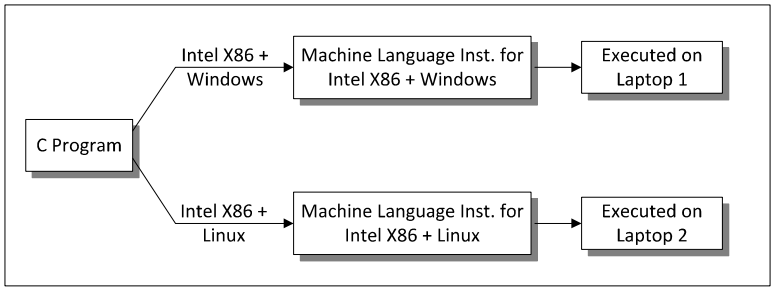

Figure 1.2

Figure 1.1 and Figure 1.2 depict a scenario called "write once, compile anywhere". It means to make the same program work on a different microprocessor + OS combination we are not required to rewrite the program, but are required to recompile the program for that microprocessor + OS combination. This is the approach taken by popular traditional languages like C and C++.

How is Java Different?

Java takes a different approach than the traditional approach taken by languages like C and C++. It lets application developers follow a "compile once, run anywhere" scenario. This means that once a Java program is compiled, it can get executed on different microprocessors + OS combinations without the need to recompile the program. This makes Java programs immensely portable across different microprocessors + OS combinations. The microprocessor + OS combination is often called "Platform". Hence Java is often called a platform-independent language or architecturally neutral language. Java programs are considered portable since they can be used on different microprocessor + OS combination without making any changes in them.

Java achieves this "compile once, run anywhere" and platform independence magic through a program called Java Virtual Machine (JVM). When we compile Java programs, they are not converted into machine language instructions for a specific microprocessor + OS combination. Instead, our Java program is converted into bytecode instructions. These bytecode instructions are similar to machine code, but are intended to be interpreted by JVM. A JVM provides an environment in which Java bytecode can be executed. Different JVMs are written specifically for different host hardware and operating systems. For example, different JVMs are written for Intel + Windows combination, ARM + Linux combination, etc.

During execution, the JVM runtime executes the bytecode by interpreting it using an Interpreter program or compiling it using a just-in-time (JIT) compiler. JIT compilers are preferred as they work faster than interpreters. During interpretation or JIT compilation, the bytecode instructions are converted into machine language instructions for the microprocessor + OS combination on which the program is being executed. This perfectly facilitates executing Java programs on different machines connected to Internet.

A Java program is typically stored in a .java file, and the bytecode is usually stored in a .class file. A complex program may consist of many .class files. For easier distribution, these multiple class files may be packaged together in a .jar file (short for Java archive). The working of a Java program discussed above is shown in Figure 1.3.

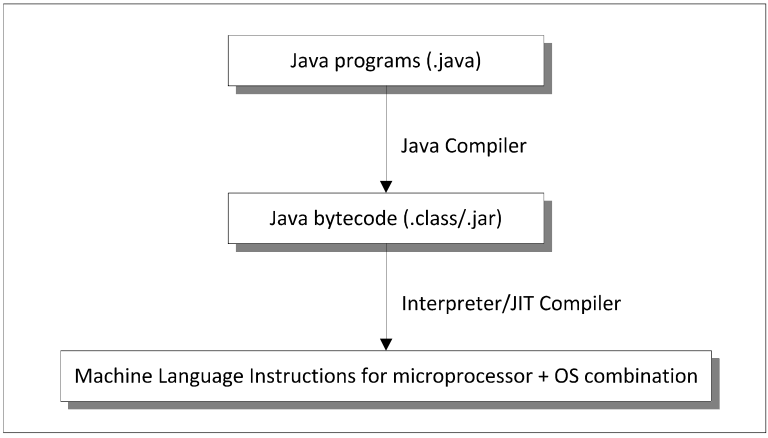

Figure 1.3

How Java addresses Security?

Let us first understand what typically happens when we use some web application on the Internet. Through browser on our PC/Laptop/Tablet/Smartphone we use a URL to reach the application present on some web server on the Internet. The web application sends HTML that gets rendered in our browser. However, except for the simplest of web applications, along with the HTML some executable Java program is also sent to our browser. This program is often small and is called Applet. The purpose of the applet is to make the web application more responsive. For example, if we enter a password, it should be possible to check whether it follows rules for password creation or not right there within the browser using the downloaded applet, rather than sending the password to server

and get it verified. This certainly improves user experience. This is because the check is being performed on the machine itself rather than on the server machine. This saves a roundtrip to the server.

But when we download an applet, there is always a possibility that the applet may contain malicious code like a Virus or Trojan horse that would cause harm to our machine. JVM prevents this from happening by restricting the applet code from accessing other resources of your machine, other than what it is supposed to. This makes applets secure. Thus JVM solves two dicey issues in one shot—portability as well as security.

Java or C++?

After learning C, it is often a question whether one should migrate to Java or C++. Answer is both; and that too in any sequence that you want. Though both are Object Oriented programming languages, neither is an advanced version of the other. Learning one before the other would naturally help to learn the second.

It is important to note that both address different sets of problems. C++ primarily addresses complexity, whereas Java addresses portability and security. In my opinion, both languages would continue to rule the hearts of programmers for years to come.

As you start learning Java, you would find that there are many features in it that are similar to C and C++. This is not by accident, but by intent. Java designers knew that they had to provide a smooth transition path to learners of Java language. That is why Java uses a syntax which is similar in many ways to that of C and it follows many of the object oriented features of C++, though in a refined fashion.

The Java Environment

We know that JVM contains an Interpreter / JIT that converts bytecode into microprocessor + OS specific machine language instructions. Since instruction sets vary from microprocessor to microprocessor, there exist different JVMs for different platforms. Thus, though any JVM can run any Java program, JVMs themselves are not portable.

JVM is distributed along with a set of standard class libraries that implement the Java Application Programming Interface (API). The Java APIs and JVM together form the Java Runtime Environment (JRE). If your need is only to execute Java programs on your machine, all that you need is JRE. For example, if you wish to play a Java-based game on your machine, you need to install only JRE on your machine for the game to run.

However, if you wish to also develop programs on your machine, you need Java Developer Kit (JDK). JDK contains tools needed to develop the Java programs, as well as JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc. Compiler converts Java code into bytecode. Java application launcher opens a JRE, loads the class, and calls its **main( )** method. Figure 1.4 shows all these pieces of Java environment.

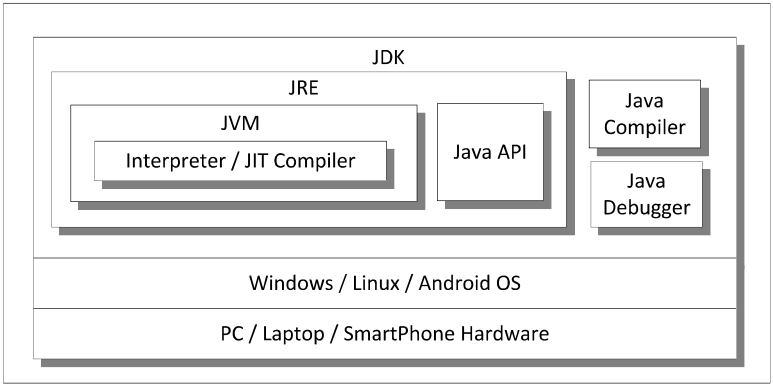

Figure 1.4

Tools of the Trade

To create and run Java programs you need to install two software on your PC. These are

(a) Java Development Kit (JDK)
(b) Integrated Development Environment like NetBeans or Eclipse

JDK is often also called Standard Edition Development Kit or Java SE 19 JDK or simply Java 19, where 19 is the version number. As years roll by, the version number would go on increasing. JDK contains JVM, JRE, Java compiler and debugger. A compiler is needed to convert the Java program into its equivalent bytecode. A debugger is needed to detect, analyze and eliminate bugs in the program. Java 19 can be downloaded using the link

https://www.oracle.com/java/technologies/download

On this download page, select the appropriate verion number, OS (Windows/Mac/Linux) and the Microprocessor (x86 or X64) and download the JDK. Next step is to install the downloaded JDK. This is a fairly simple job and I am sure you would be able to do this easily. You simply have to double click the downloaded installer file **jdk-19_windows-x64_bin.exe**

(assuming Windows and 64-bit machine configuration), and the installer would guide you through the installation process.

When you are developing a Java program you need an editor to type the program. Small Java programs can be typed in one file. But more sophisticated programs may be split across multiple files. To let you type the program, manage multiple files of your program, compile it and debug it, you need a tool that can let you carry out these tasks in a visual and user-friendly manner. This tool is often called an Integrated Development Environment (IDE). Some of the popular IDEs used for building Java programs include NetBeans, Eclipse and IntelliJ. All programs in this book have been created using NetBeans IDE. It can be downloaded using the link www.netbeans.org.

Online Java Compilers

With ubiquitous availability of Internet, if you wish, you can completely avoid installation of JDK and IDE on your machine. Using a browser, you can connect to any of the following to type, compile and execute your Java programs:

https://www.onlinegdb.com/
https://www.tutorialspoint.com/compile_java_online.php

The limitation of using online compilers is that you need a steady Internet connection while you are using them. Most of these compilers compile our program using the gcc compiler.

While using the online compilers like the one at onlinegdb.com you have to remember to choose the language (Java in our case) from the dropdown. Next, we have to type our program and click the Run/Execute button. When we do so our typed program is sent to the web server where it is compiled and executed. The output created on execution is then sent back and displayed in the browser. If any errors are found in the program during compilation they are also relayed back and displayed in the browser.

Onlinegdb.com also has provisions to create multi-file Java programs, supply command-line arguments, debug a program using the debugger, create folders and store multiple files in it. From the point of view of security, online compiler is not a preferred choice for serious software development in Java. Nevertheless, a good option with zero installation and configuration headaches when you are learning Java.

We are now on surer grounds. We now have the historical perspective of what led to creation of Java, what problems it primarily attempts to solve, and what tools we need to install to begin Java program development. It

would be a good idea to attempt the exercise on the next page to help you fix these ideas, before we formally begin learning Java language from next chapter onwards.

[A] Match the following:

(a) Creator of Java	(1) Provides security and portability
(b) JRE	(2) Platform dependent
(c) Java Program	(3) Bjarne Stroustrup
(d) JVM	(4) Contains compiler and debugger
(e) NetBeans	(5) Needed for executing Java programs
(f) Creator of C++	(6) IDE
(g) JDK	(7) Platform independent
(h) Stuctured Lang.	(8) Converts Bytecode into m/c language
(i) OOP Language	(9) Java
(j) JVM	(10) James Gosling
(k) Java Compiler	(11) C
(l) Java Interpreter	(12) Converts Java program into Bytecode
(m) Java API	(13) Library of classes

[B] State which of the following statements are True or False:

(a) Different microprocessors use different Instruction sets.

(b) Same JVM is used for all microprocessor + OS combination.

(c) We can get by just installing JRE on a machine on which we intend to only execute Java programs.

(d) NetBeans is just an IDE and doesn't have a Java compiler built in it.

(e) The Java compiler converts instructions in Java into machine language instructions.

(f) JRE and JDK both are part of JVM.

(g) The way I/O and memory management is done is same across different OSs.

(k) To run a Java program you need to install JDK.

(l) Java was conceived to create portable programs for the consumer electronic devices.

(m) Multiple .class files can be packaged together to create a .jar file.

(n) During execution Java Interpreter converts Bytecode instructions into machine language instructions.

(o) Instruction set for Intel x86 and Arm microprocessor are different.

[C] Which of the following is highlighting feature of C, C++ and Java?

(a) Structured
(b) Object Oriented
(c) Portable
(d) Secure
(e) Suitable for Internet programming
(f) Simple syntax
(g) Architecturally neutral
(h) Management of complexity

[D] Pick up the correct alternative for each of the following questions:

(a) Which of the following is CORRECT about Java programs?
   (1) Compile often, run once
   (2) Write once, compile anywhere
   (3) Write often, compile anywhere
   (4) Compile once, run anywhere

(b) In traditional programming languages, if a program has been compiled for Intel x86 + Windows combination, what should be done to make the program work on Intel x86 + Linux combination?
   (1) The program should be rewritten
   (2) The program should be recompiled
   (3) The program should be reassembled
   (4) No need to do anything, the same program would work on the new combination

(c) At what stage does byte code get converted into machine language instructions?
   (1) during compilation
   (2) during assembly
   (3) during preprocessing
   (4) during execution

(d) Java is
   (1) microprocessor-independent language
   (2) platform-independent language
   (3) OS-independent language
   (4) library-independent language

(e) For the same Java program to run on different devices we need
   (1) appropriate JVM for those devices
   (2) appropriate compiler for those devices
   (3) appropriate interpreter for those devices
   (4) appropriate preprocessor for those devices

(f) If we are to merely execute a Java program on a machine, then which of the following must be installed on that machine?
   (1) JVM
   (2) JRE
   (3) JDK
   (4) Java API

(g) Which of the following statement is CORRECT?
   (1) Different microprocessor use same instruction sets
   (2) Different platforms use same JVM
   (3) To execute a Java program we need to install JDK
   (4) To execute a Java program we need to install JRE

[E] Answer the following:

(a) Why programs written for Linux on a x86 machine do not work on a Windows x86 machine?

(b) If on a machine we wish to develop as well as execute Java programs, should we install JDK or JRE?

(c) Why are programs written in Java considered to be more portable than those written in C and C++?

(d) Why is Java language considered to be architecturally neutral?

(e) What were the pain points that led to creation of Java language?

(f) What are the different components of Java environment?

(g) What do you mean by a computing platform?

- 2 categories of software :
  - System software - OS, Compilers, Device Drivers
  - Application software - software for desktop/laptop, Web, Mobile
- Technologies used in Java world for different platforms :
  - Desktop - J2SE, Mobile - J2ME, Web - J2EE
- Reasons of popularity of Java :
  - Same language for varied applications
  - Rapid Application Development (RAD) possible
  - Easy development cycle
  - Easy to manage large projects
- Acronymns:
  - API = Application Programming Interface
  - JVM = Java Virtual Machine
  - JRE = Java Runtime Environment
  - JDK = Java Development Kit
- API = Library of classes in form of packages
- JVM = Memory Manager + Interpreter / Just In Time (JIT) compiler
- JRE = JVM + API
- JDK = JRE + Development tools like javac, java, debugger
- NetBeans, Eclipse are popular development environments
- Nebeans and Eclipse internally use javac, java, debugger
- Different JREs and JDKs have to be downloaded for different Hardware + OS combination
- For execution of Java programs only JRE is needed
- To create and execute Java programs JDK + NetBeans are needed

- In C / C++ our program on building is converted into machine language instructions.
- In Java on compiling our program is converted into ByteCode instructions.
- During execution of Java programs the ByteCode instruction are converted into machine language instructions and these insrtuctions are executed.
- Byte code instructions of a .java file are stored in corresponding .class file.
- For multiple .java files a .jar (Java Archive) file is created.
- To achieve portability C/C++ use Write once, Compile anywhere principle
- To further improve portability Java uses Compile once, Run anywhere principle

It is good to wet your feet, before you take a dip. See how to create a small program in Java…

Four important aspects of any language are the way it stores data, the way it operates upon this data, how it accomplishes input and output, and how it lets you control the sequence of execution of instructions in a program. We would discuss the first three of these building blocks in this chapter.

Java Data Types

Before we write our first Java program, it is important to understand how data is represented in Java. This is done using a data type. A data type specifies two things:

(a) What value can the data type take?
(b) What operations can be performed on the data type?

For example, an **integer** data type can take values in the range -2147483648 to +2147483647, and operations like addition, subtraction, multiplication, division, etc., can be performed on it. Similarly, a **boolean** data type can take a value true or false, and permits comparison operations on it.

Based on where a data type can be created in memory, it is called a **primitive type** (often also known as a **value type**) or a **reference type**.

Java forces primitive data types to get created only in stack and reference types only on heap. For example, an integer (like 2341) always gets created in stack, whereas a string (like "Quest") always gets created in heap. The guiding principle on the basis of which Java does this decision making is—all data types that are small in size are created in stack, and all those that occupy larger memory chunks are created in heap.

A primitive type as well as a reference type can be further categorized into pre-defined and user-defined categories. Here pre-defined means the data types that Java provides ready-made, whereas user-defined means those data types which a common user like us can create. For example, integer is a pre-defined primitive data type, whereas an Enumeration is a user-defined value type. Figure 2.1 shows the different categories of data types available in Java. Note that the pre-defined value types are often also called Primitives.

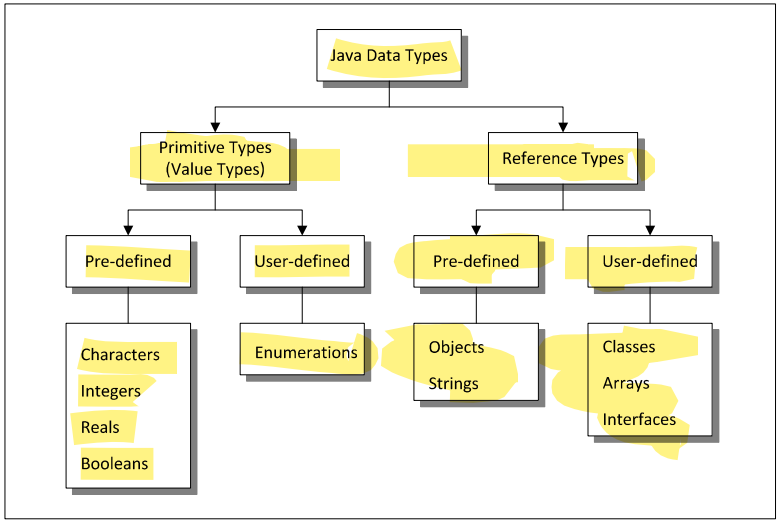

Figure 2.1

Amongst all the data types shown in Figure 2.1, to begin with, we would concentrate on the pre-defined value data types. To use the data types in a Java program, we have to create constants and variables. A constant is nothing but a specific value from the range of values offered by a data type, whereas a variable is a container which can hold a constant value. The container is typically a memory location and the variable is the name given to the location in memory. For example, if we use an expression **x = 5**, then the constant value 5 would be stored in a memory location and a name **x** would be given to that location. Whenever, we wish to retrieve and use **5**, we just have to use the variable name **x**. This is shown in Figure 2.2.

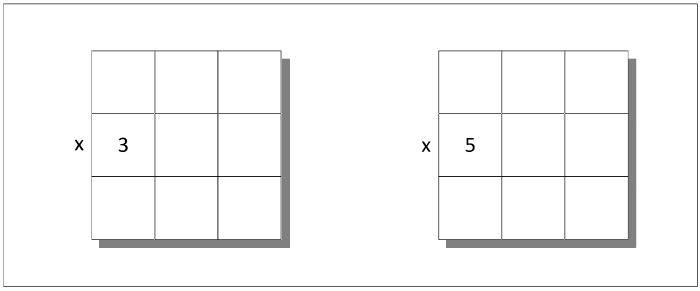

Figure 2.2

As the name suggests, value of a constant cannot change (fixed), whereas, value of a variable's can change (vary). A constant is often called a **literal**, whereas, a variable is also known as an **identifier**. Figure 2.3 gives list of commonly used pre-defined primitive data types along with the range of values that they can take and the numbers of bytes they occupy in memory.

Data Type	Range	Size in bytes
char	0 to 65535	2
int	-2147483648 to +2147483647	4
float	-3.4e38 to +3.4e38	4

Figure 2.3

There are certain rules that one needs to observe while creating constants and variables. These are discussed below.

Rules for Constructing Constants

(a) If no sign precedes a numeric constant, it is assumed to be positive.

(b) No commas or blanks are allowed within a constant.

(c) The bytes occupied by each constant are fixed and do not change from one compiler to another.

(d) Only a float constant can contain a decimal point.

(e) A float constant must be followed by a suffix **f**.

(f) A float constant can be expressed in fractional from (example 314.56f) or exponential form (example 3.1456e2).

(g) A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, 'A' is a valid character constant, whereas 'A' is not.

Given below are examples of some valid constants.

426	+78.23	-8000	-7605
true	'A'	'+'	'3'

Rules for Constructing Variable Names

(a) A variable name is any combination of alphabets, digits, underscores (_) and dollars ($).

(b) The first character in the variable name must be an alphabet, underscore or dollar.

(c) No commas or blanks are allowed within a variable name.

(d) Variable names are case-sensitive. So, abc, ABC, Abc, aBc, AbC are treated as different variables.

While creating variable names, conventions given below are commonly followed.

(a) A variable name usually begins with an alphabet. Ex. speed, average

(b) A variable representing money usually begins with $. Ex. $interest, $salary.

(c) If a variable name containing multiple words, the words are either connected using underscore or follow a camel-case notation. Ex. current_speed, currentSpeed, avg_salary, avgSalary.

While following these rules and conventions, one must avoid the temptation of creating long variable names as it unnecessarily adds to the typing effort.

The rules remain same for constructing variables of any type. Naturally the question follows—how is Java able to differentiate between these variables? This is a rather simple matter. Java compiler makes it compulsory for us to declare the type of any variable name that we wish to use in a program. Here are a few examples showing how this is done.

```
Ex.: int   si, m_hra ;
     float   bassal ;
     char   code ;
```

Since, there is no limit on maximum allowable length of a variable name, an enormous number of variable names can be constructed using the

above-mentioned rules. It is a good practice to exploit this enormous choice in naming variables by using meaningful variable names.

Thus, if we want to calculate simple interest, it is always advisable to construct meaningful variable names like **prin**, **roi**, **noy** to represent Principal, Rate of interest and Number of years rather than using the variables **a**, **b**, **c**.

Java Keywords

Keywords are the words whose meaning has already been explained to the Java compiler. When we make the declaration like the one shown below,

```
int   age ;
```

age is a variable, whereas **int** is a keyword. When this declaration is made, we are telling the compiler that the variable **age** be treated as a variable of type integer. But we don't have to be so elaborative, just **int age** conveys the same meaning. This is because the meaning of the keyword **int** has already been explained to the Java compiler.

The keywords cannot be used as variable names because if we do so, we are trying to assign a new meaning to the keyword which is not allowed. There are only 48 keywords available in Java. Figure 2.4 gives a list of these keywords for your ready reference. A detailed discussion of each of these keywords would be taken up in later chapters wherever their use is relevant.

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

Figure 2.4

The First Java Program

Armed with the knowledge of variables, constants and keywords, the next logical step is to combine them to form instructions. However, instead of this, we would write our first Java program now. Once we have done that we would see in detail the instructions that it made use of.

Before we begin with our first Java program, do remember the following rules that are applicable to all Java programs:

(a) Each instruction in a Java program is written as a separate statement. Therefore, a complete Java program would comprise a series of statements.

(b) Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.

(c) All statements are in small case letters.

(d) Every Java statement must end with a semicolon (**;**).

Let us now write our first Java program. It would simply calculate simple interest for a set of values representing principal, number of years and rate of interest.

```
// Calculation of simple interest
package calofsi ;
public class CalOfSi
{
    public static void main ( String[ ] args )
    {
        float  p, r, si ;
        int  n ;
        p = 1000.50f ;
        n = 3 ;
        r = 15.5f ;
        si = p * n * r / 100 ;
        System.out.println ( si ) ;
    }
}
```

Now a few useful tips about the program...

- Comment about the program should either be enclosed within /* */ or be preceded by //. For example, the first statement in our program is a comment.
- Though comments are not necessary, it is a good practice to begin a program with a comment indicating the purpose of the program, its author and the date on which the program was written.
- Sometimes it is not very obvious as to what a particular statement in a program accomplishes. At such times, it is worthwhile mentioning the purpose of the statement (or a set of statements) using a comment. For example

  ```
  /* formula for simple interest */
  si = p * n * r / 100 ;
  ```

- A comment can be split over more than one line, as in,

  ```
  /* This is
     a multi-line
     comment */
  ```

  Such a comment is often called a multi-line comment.
- A Java program is a collection of one or more packages. Each package can contain multiple classes. Each class may contain multiple functions. Each function can contain multiple instructions. This typical organization of a Java program is shown in Figure 2.5.
- Every instruction used in a Java program should belong to a function. Every function must belong to a class and every class must belong to a package. In our program, there is a package called **calofsi**, a class called **CalOfSi** and a function called **main( )**. **package** and **class** both are keywords.

  Right now we do not want to go into the details of package and a class. We would learn about packages and classes in later chapters. But it would be a good time to get introduced to the concept of a function.

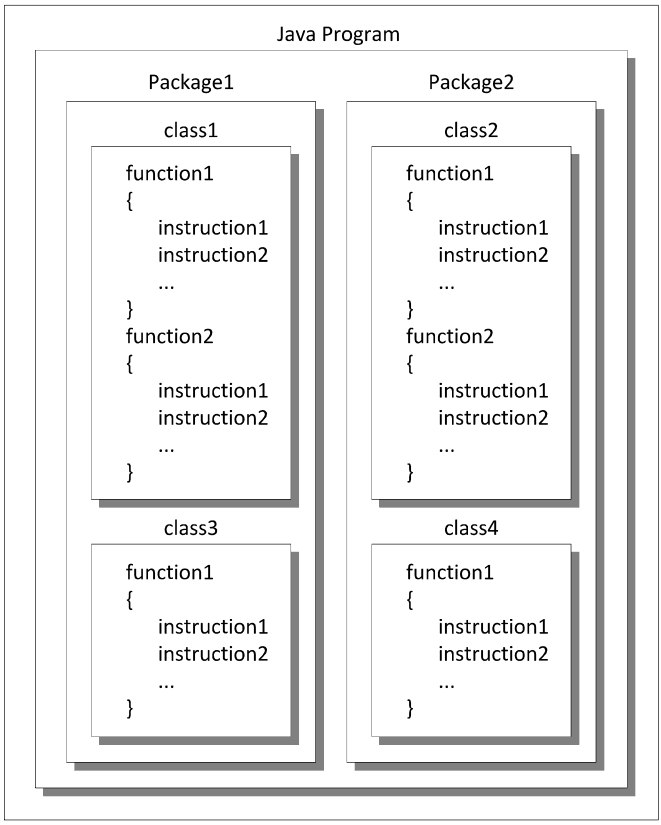

Figure 2.5

– **main( )** is a function. A function contains a set of statements. Though a Java program can contain multiple functions, to begin with, we would concentrate on only those programs which have only one function. All statements that belong to **main( )** are enclosed within a pair of braces { } as shown below.

```
public static void main ( String[ ] args )
{
    statement 1 ;
    statement 2 ;
    statement 3 ;
}
```

- The way functions in a calculator return a value, similarly, functions in Java also return a value. Since we do not wish to return any value from **main( )** function, we have to specify this using the keyword **void** before **main( )**. **main( )** is always preceded by the keyword **static**. The purpose of this keyword and detailed working of functions would be discussed in Chapters 9 and 7 respectively.

- Any variable used in the program must be declared before using it. For example,

```
int  p, n ;      /* declaration */
float  r, si ;   /* declaration */
si = p * n * r / 100 ;   /* usage */
```

- Any Java statement always ends with a semicolon (;). For example,

```
float  r, si ;
r = 15.5f ;
```

- In the statement,

```
si = p * n * r / 100 ;
```

***** and **/** are the arithmetic operators. The arithmetic operators available in Java are **+**, **-**, ***** and **/**. Java is very rich in operators. There are totally 41 operators available in Java.

- Once the value of **si** is calculated, it needs to be displayed on the screen. Unlike other languages, Java does not contain any instruction to display output on the screen. All output to screen is achieved using ready-made library functions. One such function is **println( )**. We have used it to display the value contained in **si** on the screen.

- Actually, **println( )** is a function of **PrintStream** class, and **out** is a **static** object defined in a **System** class. We would learn classes, objects and static members in Chapter 8. As of now, let us just use the syntax **System.out.println( )** whenever we wish to display output on the screen.

- If we wish we can print multiple values using **println( )** function. This is shown below.

```
System.out.println ( si + " " + p + " " + n + " " + r ) ;
System.out.println ( "Simple interest = Rs. " + si ) ;
System.out.println ( "Principal = " + p + " Rate = " + r ) ;
```

The output of these statements would look like this...

```
465.2325 1000.5 3 15.5
Simple interest = Rs. 465.2325
Principal = 1000.5 Rate = 15.5
```

Compilation and Execution

We need to carry out the following steps to create, compile and execute our first Java program. It is assumed that you have installed JDK and NetBeans as per the instructions in Chapter 1.

(a) Start NetBeans from Start | All Programs.

(b) Select File | New Project from the File menu. Select 'Java' from 'Categories' list box and 'Java Application' from 'Projects' box. Click on 'Next' button.

(c) Give a proper name for the project in 'Project Name' text box (say **CalOfSi**). Choose suitable location for the project folder, then click on Finish.

(d) NetBeans would provide a skeleton program that would contain a **package** statement, a public class **CalOfSi** and a function **main( )**, all defined in a file called **CalOfSi.java**. Note that the name of the file and the name of the public class are same in all Java programs.

(e) Type the statements of our simple interest program in **main( )**.

(f) Save the program using **Ctrl+S**.

(g) Compile and execute the program using **F6**.

One More Program

We now know how to write an elementary Java program, type it, compile it and execute it. These are the steps that you will have to carry out for every program. So to help you fix your ideas, here is one more program. It calculates and prints average value of 3 numbers.

```
/* Calculation of average */
package calofavg ;
public class CalOfAvg
```

```
{
    public static void main ( String[ ] args )
    {
        int  x, y, z, avg ;
        x = 73 ;
        y = 70 ;
        z = 65 ;
        avg = ( x + y + z ) / 3 ;
        System.out.println ( avg ) ;
    }
}
```

[A] Which of the following is invalid variable name and why?

BASICSALARY	_basic	basic-hra
#MEAN	group.	422
population in 2006	over time	mindovermatter
SINGLE	hELLO	queue.
team'svictory	Plot # 3	2015_DDay

[B] Point out the errors, if any, in the following Java statements:

(a) int = 314.562f * 150 ;

(b) name = 'Ajay' ;

(c) varchar = '3' ;

(d) 3.14f * r * r * h = vol_of_cyl ;

(e) k = (a * b) (c + (2.5fa + b) (d + e) ;

(f) m_inst = rate of interest * amount in rs ;

(j) k = ((a * b) + c) (2.5f * a + b) ;

(k) a = b = 3 = 4 ;

(l) count = count + 1 ;

(m) date = '2 Mar 11' ;

[C] Pick up the correct alternative for each of the following questions:

(a) Which of the following is the correct way to write a comment?

(1) // This is a comment
(2) / This is a comment
(3) /* This is a comment
(4) /* This is a /* comment */ */

(b) The maximum value that an integer constant can have is:

(1) -2147483647
(2) 2147483647
(3) 3.4×10^{38}
(4) -3.4×10^{38}

(c) A Java variable cannot start with:

(1) An alphabet
(2) A number
(3) A special symbol other than underscore
(4) Both (2) and (3) above

(d) Which of the following is odd one out?

(1) +
(2) -
(3) /
(4) **

[D] Answer the following:

(a) Assume a suitable value for Ramesh's basic salary. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a Java program to calculate his gross salary.

(b) Assume a suitable value for distance between two cities (in km.). Write a Java program to convert and print this distance in meters, feet, inches and centimeters.

(c) Assume suitable values for marks obtained by a student in five different subjects are input through the keyboard. Write a Java program to find out the aggregate marks and percentage marks obtained by the student. Assume that the maximum marks that can be obtained by a student in each subject is 100.

(d) Assume a suitable value for temperature of a city in Fahrenheit degrees. Write a Java program to convert this temperature into Centigrade degrees and print both temperatures.

(e) Assume suitable values for length and breadth of a rectangle, and radius of a circle. Write a Java program to calculate the area and perimeter of the rectangle, and the area and circumference of the circle.

- Constants = Literals -> Cannot change
  Variables = Identifiers -> May change
- Data Types : 1) Primitives (value types) 2) Reference types
- Primtive types :
  - Char - 2 bytes
  - Integers - byte, short, int, long (sizes - 1, 2, 4, 8 bytes)
  - Real - float, double (sizes - 4, 8 bytes respectively)
  - Boolean - true / false (1 bit)
- Derived types (classes) :
  - Library : String, System, Exception, etc.
  - User-defined : CalOfSi, CalOfAvg, etc.
- Variable names are case-sensitive and should begin with an alphabet
- Total keywords = 48. Example : int, char, float
- A Java program is a collection of one or more packages
- Each package can contain multiple classes

- Each class may contain multiple functions
- A variable must belong to either a function or a class
- No global functions or variables in Java
- public class is accessible from outside the package
- public function is accessible from outside the class
- 3 types of comments :
  - Single line - //
  - Multiline - /* ... */
  - Documentation - /** .... */