Easy JAVA

(An Interactive Programming Language)

2nd Edition

Vanam Mallikarjun

Author Resident Address

V. Mallikarjun
S/o V. Ramanaiah,
H.No: 11-22-405,
Kashibugga,
Warangal - 506 002.
Telangana State.
Cell : 9849671334
Email : easybooksapexcomputers@gmail.com

Rs. **350 /-**

About the Author

The author obtained his Degree in Bachelor of Business Management (B.B.M) from Kakatiya University, Warangal and Masters in Computer Science from Maharshi Dayanand University, Rohtak, New Delhi.

The author is well known for his excellent teaching in the subjects related to computer science like **C, C++, Data Structures, VC++, JAVA (Core, Adv, J2EE), VB, ORACLE, .NET, PHP** and **Android** To his credit, he has few more publications on **Easy C** and **Easy CPP**.

He has his own Institute **APEX COMPUTER EDUCATION** in warangal, a prime learning centre for computer science subjects for many of the Polytechnic, Engineering, M.sc.(CS) and MCA students in and around warangal city.

At present, the author is the secretary cum correspondent on the management of **APEX ENGINEERING COLLEGE,** Vishwanadhapuram(Vill.), Geesugondal(Mdll.), Warangal(Dist.), T.S. which is Approved by AICTE, New Delhi & Affiliated to JNTU, Hyd.

Preface

Java technology is a high-level programming and a platform independent language. Java is designed to work in the distributed environment on the Internet. Java has a GUI features that provides you better "look and feel" over the C++ language, moreover it is easier to use than C++ and works on the concept of object-oriented programming model. Java enable us to play online games, video, audio, chat with people around the world, Banking Application, view 3D image and Shopping Cart. Java find its extensive use in the intranet applications and other e-business solutions that are the grassroots of corporate computing. Java, regarded as the most well described and planned language to develop an applications for the Web.

Java is a well known technology which allows you for software designed and written only once for an "virtual machine" to run on a different computers, supports various Operating System like Windows PCs, Macintoshes, and Unix computers. On the web aspect, Java is popular on web servers, used by many of the largest interactive websites. Java is used to create standalone applications which may run on a single computer or in distributed network. It is also be used to create a small application program based on applet, which is further used for Web page. Applets make easy and possible to interact with the Web page.

This book has been written keeping in mind the difficulties of the students which I came to realize whih teaching the subject during the last 10 years. Infact, it is in response to a long standing demand from the students I have tried to lay emphasis on fundamental concepts, classes and objects, inheritance, interfaces, packages, exceptional handling, multithreading, applets and swings. Efforts have been made to present the subject matter in simple langauge and a systamatic way so that even a fresher can easily follow.

We do not claim this hand book to be a complete text book on a particular subject. I have tried to ensure that there are no major technical or grammatical errors in this book. But in a work of this magnitude some small errors are bound to creep in. As a reader you are like to go through every line of this book. If you notice some errors do let me know through the feedback.

Do share your views and suggestion to enable me to make this book a better learning tool. I will be glad to receive any suggestions. The readers must feel free to send feedback to the mail id **easybooksapexcomputers@gmail.com**

I would like to wish you success in your determined efforts to master computers.

Good Luck and Happy Learning.

Vanam Mallikarjun

CONTENTS

Easy JAVA 2nd Edition

Chapter - 1

Basics of Java

Basics of Java

History of JAVA

Back in 1990, a gentleman by the name of James Gosling was given the task (assigned job) of creating programs to control **consumer electronics** such as TV, VCR, Lights, Phone, refrigerator etc. **James Gosling** and his team of people (**Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan**) at **Sun Microsystems** started designing their software using **C++**, the language that most programmers were praising as the next big thing because of its object-oriented nature. Gosling, however, quickly found that C++ was not suitable for the projects that he and his team had in mind. They ran into trouble with complicated aspects of C++ such as **multiple inheritance** of classes and with program bugs such as **memory leaks**. Gosling soon decided that he was going to have to come up with his own, simplified computer language that would avoid all the problems he had with C++.

Although Gosling didn't care for the complexity of languages such as C++, he did like the basic syntax and object-oriented features of the language. So, he decided to design his new language, he used C++ as its model, removing away all the features of C++ that made that language difficult to use with his consumer-electronics projects. When Gosling completed his language-design project, he had a new programming language that he named **Oak**. He got the name Oak when Gosling seen out his office window at an Oak tree.

Oak language was first used in something called the Green project, wherein the development team attempted to design a **control system** for use in the home. This control system would enable the user to manipulate a list of devices, including **TVs**, **VCRs**, **lights**, and **telephones**, all from a hand-held computer called ***7** (Star Seven). The *7 system featured a touch-sensitive screen that the owner used to select and control the devices supported by the control.

The next step for Oak was the **video-on-demand** (VOD) project, in which the language was used as the basis for software that controlled an interactive television system. Although neither ***7** nor the **VOD** project led to actual products, they gave Oak a chance to develop and mature. By the time Sun discovered that the name "Oak" was already claimed (given name for other language) and they changed the name to **Java**, and they had a powerful, yet simple, language on their hands.

More importantly, Java was a **platform-neutral** language, which meant that programs developed with Java could run on any computer system with no changes. This platform independence was attained by using a special format for compiled Java programs. This file format, called **byte-code**, could be read and executed by any computer system that has a **Java interpreter**. The Java interpreter, of course, must be written specially for the system on which it will run.

In 1993, after the World Wide Web had transformed the text-based Internet into a graphics-rich environment, the Java team realized that the language they had developed would be perfect for Web programming. The team came up with the concept of Web applets which are small programs that could be included in Web pages and even went so far as to create a complete Web browser (now called HotJava) that demonstrated the language's power.

In the second quarter of 1995, Sun Microsystems officially announced Java. The "new" language was quickly got popular as a powerful tool for developing Internet applications. Netscape Communications who are the developer of popular Netscape Navigator Web browser, added support for Java to its new Netscape Navigator 2.0 onwards. Other Internet software developers are sure to use java, including Microsoft, whose Internet Explorers offers Java support. After more than five years of development, Java has found its home.

By now, you may be curious why Java is considered such a powerful tool for Internet development projects. Java is a simplified version of C++. C++ added so much to the C language that even professional programmers often have difficulty making the transition.

According to Sun Microsystems, Java is ***"simple, object-oriented, statically typed, compiled, interpreter, architecture neutral, platform independent, portable, multi-threaded, garbage collected, robust, secure, extensible and well-understood."***

Features (or) characteristics of Java

Simple: Java's developers removed many of the unnecessary features of other high-level programming languages. For example, Java does not support pointers, implicit type casting, structures or unions, operator overloading, templates(but introduced from java 1.5), friend functions and multiple inheritance.

Object-oriented: Just like C++, Java uses classes to organize code into logical modules. At runtime, a program creates objects from the classes. Java classes can inherit from other classes, but multiple inheritance is not allowed.

Statically typed: All objects used in a program must be declared before they are used. This enables the Java compiler to locate and report data type conflicts (mismatch).

Compiled: Before we can run a program written in the Java language, the program must be compiled by the Java compiler. The compilation results in a "**byte-code**" file which is an intermediate language that is not either English or Machine language which can be executed under any operating system that has a **Java interpreter(JVM)**. This interpreter reads in the byte-code file and translates the byte-code commands into machine-language commands that can be directly executed by the machine that's running the Java program. We could say, then, that the **Java is both a compiled and interpreted language**.

Architecture Neutral: Java was designed in such a way that, programs written in Java can be executed on any machine irrespective of its architecture and called java is architectural neutral language. This is achieved because of **ByteCode** and **JVM**(Java Virtural Machine).

Plantform Independent: Java bytecode programs execute in different operating systems provided that they should contain **JVM**. If the operating system does not contain JVM then the java programs does not execute.

Portability : A Java program once written can be executed on any machine (computers) that differes in operating systems and hardware. Therefore java programs are portable programs.

Multi-threaded: Java programs can contain multiple threads of execution, which enables programs to handle several tasks concurrently. For example, a multi-threaded program can display an image on the screen in one thread while continuing to accept keyboard input from the user in the other thread. All applications have at least one thread, which represents the program's **main** path of execution.

Garbage collected: In Java, waste memory is de-allocated by the garbage collector. The garbage collector is a special program which is a part of JVM when executeds come into memory and remove all unwanted objects memory so that the memory can be used for other purpose. Therefore the programmer

can only concentrate on allocation of memory and need not to concentrate on de-allocation. Advantage of garbage collection is no wastage of memory.

Robust: Since the Java interpreter checks all system access performed within a program, Java programs cannot crash the system i.e. does not make computer struck or stop working. Instead, when a serious error is discovered, Java programs create an exception (error). This exception can be captured and managed by the program without any risk of making the computer struck or stop. Robust programs are written in java using exceptional handling mechanism.

Secure: Java programs are secured programs because web applets of Java executes only in browsers and can't access any resources of computer such as files on hard disk, data in RAM etc,. Virus programs do not enter into computer through web applets. Java does not support pointers therefore java programs cannot gain access to areas of system for which they have no authorization.

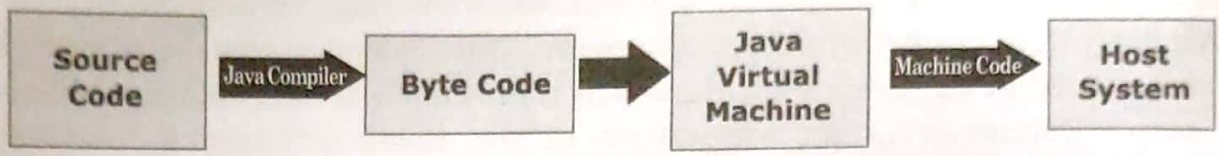
Extensible: Java programs support **native methods**, which are functions written in another language, usually C++. Support for native methods enables programmers to write functions that may execute faster than the equivalent functions written in Java. Native methods are dynamically linked to the Java program; that is, they are associated with the program at runtime. As the Java language is further refined for speed, native methods will probably be unnecessary.

Well-understood: The Java language is based upon technology that's been developed over many years. For this reason, Java can be quickly and easily understood by anyone with experience with modern programming languages such as C++.

What is Java Virtual Machine?

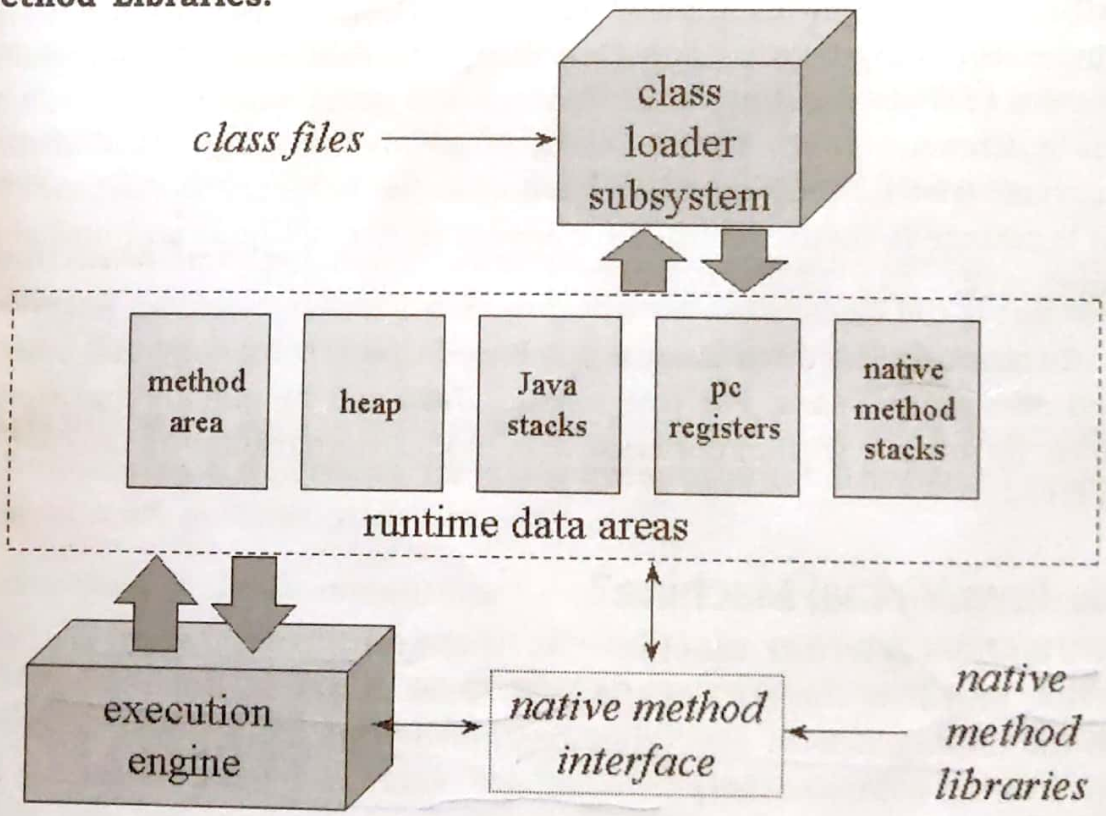
JVM(**J**ava **V**irtual **M**achine) is the main component of Java architecture and it is the part of the **JRE** (**J**ava **R**untime **E**nviroment) . It provides the cross platform functionality to java. This is a software process that converts the compiled Java byte code to machine code. Byte code is an intermediary language between Java source and the host system. Most programming language like C and Pascal converts the source code into machine code for one specific type of machine as the machine language vary from system to system . Mostly compiler produce code for a particular system but Java compiler produce code for a virtual machine . JVM provides security to java.

The programs written in Java or the source code translated by Java compiler into byte code and after that the JVM converts the byte code into machine code for the computer one wants to run. JVM is a part of Java Run Time Environment that is required by every operating system requires a different JRE .



The Architecture of the Java Virtual Machine

The following block diagram of Java Virtual Machine shows different components that plays different roles in execution of java bytecode file. When a java bytecode file(.class) is submitted to JVM, it uses different components to execute the bytecode. The different componenets of JVM are: 1**) Class loader sub-system** 2) **Memory Areas** and 3) **Execution engine**. 4) **Native Method Libraries.**



1) Class Loader Subsystem

The Java Virtual Machine contains **a *class loader***, which loads class files from both the program and the **Java API**. Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine.

When the java class file(bytecode) is subitted to **JVM** for the execution of program. While execution of java program, when the **JVM** come across class name for use then the class loader subsystem loads the class into memory and then the objects are defined in heap. This component is one of the main subsystems of **JVM** that loads types(classes and interfaces) into method area memory as and when required for the **JVM**. This mechanism of loading classes into memory is known as **class loading**, and at the time of class loading the static variables are defined in memory.

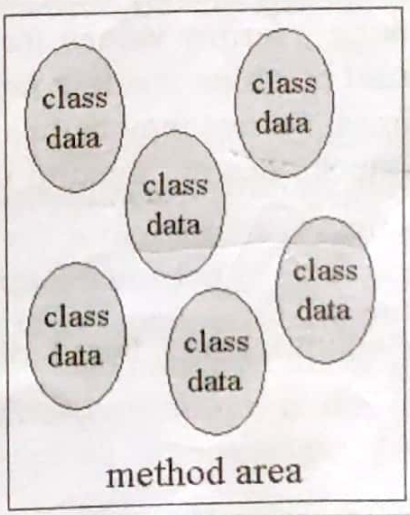
2) Runtime data areas

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, threads and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into five *runtime data areas*: **Method area**, **Heaps**, **Java Stacks**, **PC registers**, and **Native Method Stacks**.

2.1. Method Area

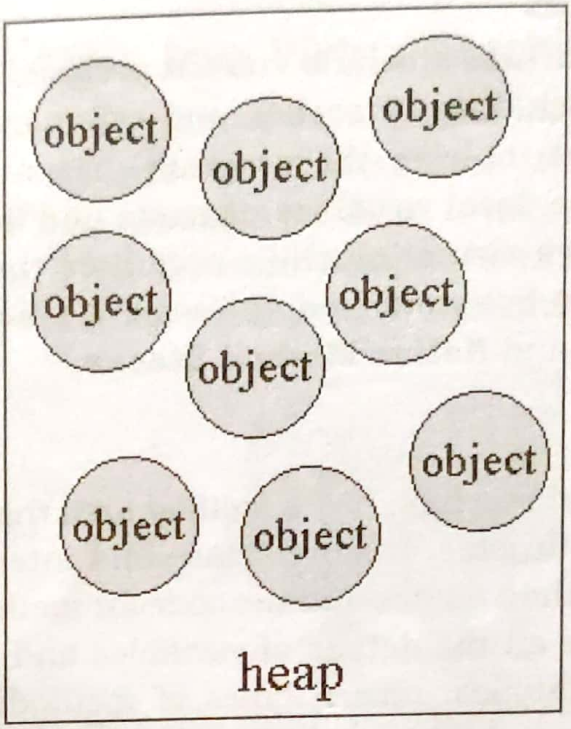
The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. It stores class and interface structures such as field names and method names and the code for methods and constructors used in class. It store all the details of variables and methods in the class such as modifies, datatypes, return types of methods, superclass names, number of parameters and their type of methods etc.

The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.



2.2. Heap

The Java virtual machine has a *heap* memory that is shared among all Java virtual machine threads. The heap is the runtime data area on which memory for objects and arrays are allocated. The heap is created on virtual machine start-up. The objects on the heap are reclaimed (deallocated) by a ***garbage collector***; objects are never explicitly deallocated. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.



2.3 Java Virtual Machine Stacks or Java Stack

Each Java virtual machine thread has a private ***Java virtual machine stack***, created at the same time as the **JVM** start-up. A Java virtual machine stack stores **frames**(A *frame* is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions.) . The Java virtual machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a Java virtual machine stack does not need to be contiguous.

Example:

```
class Example1
{
 public static int staticmethod(int i, long l, float f, double d, Object o, byte b)
 {
        return 0;
 }
```

```
public int nonstaticmethod(char c, double d, short s, boolean b)
{
        return 0;
}
}
```

The Java stack created for the above two methods of Example1 class is:

Stack of staticmethod()

index	type	parameter
0	int	int i
1	long	long l
2	float	float f
3	double	double d
4	reference	Object o
5	int	byte b

Stack of nonstaticmethod()

index	type	parameter
0	reference	hidden this
1	int	char c
2	double	double d
3	int	short s
4	int	boolean b

2.4. The pc Register

The Java virtual machine can support many threads of execution at once. Each Java virtual machine thread has its own pc (program counter) register. At any point, each Java virtual machine thread is executing the code of a single method, the current method for that thread. If that method is not native, the pc register contains the address of the Java virtual machine instruction currently being executed. If the method currently being executed by the thread is native, the value of the Java virtual machine's pc register is undefined. The Java virtual machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.

2.5 Native Method Stacks

An implementation of the Java virtual machine may use conventional stacks called "C stacks" to support native methods, methods written in a language other than the Java programming language. Native method stacks may also be used by the implementation of an interpreter for the Java virtual machine's instruction set in a language such as C. Java virtual machine implementations that cannot load native methods and that do not themselves rely on conventional stacks need not supply native method stacks. If supplied, native method stacks are typically allocated per thread when each thread is created.

3) Execution Engine

The execution engine is one of the main components of JVM which converts the byte code of java program into native language and executes in the computer. The execution engine contains an interpreter that interprets the bytecode into native code one statement after the other and executes in the machine. This way of interpreting the code is much slower. To improve the performace of interpreter another translator was introduced into the execution engine of JVM which is known as **Just-In-Time(JIT) compiler**.

A just-in-time compiler can be used as a way to speed up execution of bytecode. At the time the bytecode is run, the just-in-time compiler will compile some or all of it to native machine code for better performance. This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just-in-time"), and then cached and reused later without needing to be recompiled.

For example, The JIT compiler translates the bytecodes of a method to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked. Therefore the same method if executed for the second time and so on, it need not to be translated into native code, where as the interpreter of JVM interprets the method code each and every time it come across the method.

4) Native Method Libraries

When running on a Java Virtual Machine that is implemented in software on top of a host operating system, a Java program interacts with the host by invoking ***native methods***. In Java, there are two kinds of methods: **Java and native**.

A Java method is written in the Java language, compiled to bytecodes, and stored in class files. A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor. Native methods are stored in **a dynamically linked library** whose exact form is platform specific.

While Java methods are platform independent, native methods are not. When a running Java program calls a native method, the virtual machine loads the dynamic library that contains the native method and invokes it through native method interface as shown in the architecture of JVM.

The Java Keywords

There are 50 reserved keywords currently defined in the jdk1.6. Keywords are part of java language and they should be used in the same context in which they are given. The meanings of these keywords are already explained in the java, since they should be used in the same manner in which they are present. These keywords cannot be used as names for a variable, class or method.

The keywords are:

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

In addition to java keywords, it have three reserve values **true, false** and **null.** These are the values of variables, classes and so on. These three values are not considered as keywords.

Data types in Java

Java defines 8 simple types of data; **byte**, **short**, **int**, **long**, **char**, **float**, **double** and **boolean**. These are grouped into four.

1. **Integer category:** This group includes **byte**, **short**, **int** and **long** which are for whole-valued signed numbers.
2. **Float category:** This group includes **float** and **double** which represents numbers with fractional precision.
3. **Characters:** This group includes **char**, which represents a unicode character.
4. **Boolean:** This group includes **boolean** which is a special type for representing **true/false** values.

Data type name	Size(bytes)	Range
byte	1	-128 to 127
short	2	-32768 to 32767
int	4	-2147483648 to 2147483647
long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	- 3.4e-038 to 3.4e+038
double	8	-1.7e-308 to1.7e+308
char	2	0 to 65535
boolean	1	true or false

Java is strongly typed language

Java is a strongly typed language. The rules and regulations of language are strictly followed; otherwise errors are raised by the compiler at compilation time.

Every variable has a type, every expression has a type and every type is strictly defined. The data types of parameters in the called function should match with the data types of arguments from the function call. Any mismatche of errors must be corrected before the compilation of class.

Output Statment: System.out.print ( )

System is a predefined class and **out** is an object of PrintStream class which is defined in the **System** class. The **print()** and **println()** methods are present in the PrintStream class which prints the given values on the monitor.

```
class System
{
        :
        public static final PrintStream out;
        :
}
class PrintStream
{
        :
        public void print();
        public void print(boolean);
        public void print(char);
        public void print(int);
        public void print(long);
```

```
    public void print(float);
    public void print(double);
    public void print(char[]);
    public void print(java.lang.String);
    public void print(java.lang.Object);
    public void println();
    public void println(boolean);
    public void println(char);
    public void println(int);
    public void println(long);
    public void println(float);
    public void println(double);
    public void println(char[]);
    public void println(java.lang.String);
    public void println(java.lang.Object);
          :
}
```

The **print()** methods prints values on the same line where as **println()** methods prints values on different lines, because **println()** after printing the values it prints a new line character (**\n**).

Example:

```
System.out.print ( " Hello world" ) ;
system.out.print ( " Java welcomes U " );
```

Output:

```
1  Hello world Java welcomes  U
2
3
```

Example:

```
System.out.println("Hello world");
System.out.println("Java welcomes U");
```

Output:

```
1  Hello world
2  Java welcomes U
3
```

Other examples:

```
int a=100;
System.out.println(a);
System.out.println("a= " + a);
```

Output:

```
100
a=100
```

The **print()** and **println()** methods evaluates the values from left to right. In the above statement **System.out.print("a=" + a);** in which the string **"a="** is combined with **a(10)** and the result value it becomes is **"a=10"** as string and which is printed on the monitor.

The + operator in java have two meanings **1. Addition 2. Concatenation [Combine]**. If + is operating on two numbers then addition is perfomed.

Examples:

```
5 + 12  => 17
3.5 + 20 => 23.5
```

If one or two values are strings then + operator performs concatenation operation(i.e. combines two values).

Examples:

```
15 + "AB" => "15AB"
"AB" + 15 => "AB15"
"AB" + "XY" => "ABXY"
```

Examples:

```
int a=10, b=20;
System.out.println(a+b);
System.out.println("sum="+a+b);
System.out.println("sum="+(a+b));
```

Output:

```
30
sum=1020
sum=30
```

In the statement **System.out.println("sum="+a+b);** where the evaluation takes like this

```
"sum=" + a(10)      => "sum=10"
"sum=10" + b(20)    => "sum=1020"
```

In the statement **System.out.println("sum="+(a+b));** where brackets are given first preference and the evaluation takes like this

```
"sum="+(10+20)
"sum=" +30  =>  "sum=30"  which prints on screen.
```

About the Java Software

Java software is available in different versions like **JDK 1.0**, **JDK 1.1**, **J2SE 1.2** and so on **J2SE 1.6**. If this software is loaded in to the system, the software resides in the **program files** directory in this way

C:\program files\java\jdk1.6.0\bin

The bin directory contains large number of executable files and among which the two important files like **javac.exe** and **java.exe** are used to compile and run the java programs.

The **javac.exe** program compiles the **.java** program into bytecode and stores in the **.class** file.

Compilation Statement:

C:\program files\java\jdk1.6.0\bin> javac firstexample.java

The **javac** compiler converts **firstexample.java** into bytecode and stores into **firstexample.class** file.

Execution Statement:

The **java.exe** program runs the **.class** file by interpreting the bytecode into machine language.

Example:

C:\program files\java\jdk1.6.0\bin> java firstexample

The **java** translates **firstexample.class** bytecode into machine language and executes in the computer.

Note: Don't specify the **.class** extenstion when used with **java** command.

Rules of Java Program

1. A Java program should contain minimum of one class.
2. One of the classes should contain **main** function with String array argument.
3. Main method should be defined with **public** and **static.**
4. The source file name and class name that contain **main** method should have same names but it is not compulsory.

Note: Java programs can be written in any editors such as **Edit**(Dos), **Notepad**(Windows), **Vi**(Unix/Linux), etc,.

In this book, java programs are written using **Edit** command.

First Java Program

Writing the java program.

C:\program files\java\jdk1.6.0\bin> edit firstexample.java

```
// This is first java program
class firstexample
{
    public static void main (String argv[])
    {
       System.out.print("This is my first program in Java");
    }
}
```

[save and exit the editor]

Compiling the java program
C:\program files\java\ jdk1.6.0\bin> javac firstexample.java

We get **firstexample.class**, if **firstexample.java** doesn't contain errors.

Executing the java program
C:\program files\java\ jdk1.6.0\bin> java firstexample

Output:

```
This is my first program in Java
```

Example Programs

1. A Java program to print the sum of two numbers.

Bin> edit sum.java

```
class sum
{
    public static void main(String argv[])
    {
          int a,b,c;
          a=100;
          b=200;
          c=a+b;
          System.out.print("\n a="+a+"\n b="+b+"\n c="+c);
    }
}
```

Bin>javac sum.java

Bin>java sum

a=100
b=200
c=300

2. A Java program to print the biggest of two numbers.

Bin> edit big.java

```
class big
{
    public static void main(String argv[])
    {
          int a,b;
          a=100;
          b=200;
          if (a>b)
                System.out.print("\n biggest : " + a);
          else
                System.out.print("\n biggest : " + b);
    }
}
```

Bin>javac big.java

Bin>java big

biggest : 200

3. A Java program to print 1 to 10 numbers.

Bin> edit loop.java

```
class loop
{
     public static void main(String argv[])
     {
          int i;
          for(i=1; i<=10; i++)
          {
                System.out.print("\t" + i);
          }
     }
}
```

```
Bin>javac loop.java
Bin>java loop
1 2 3 4 5 6 7 8 9 10
```

Note: All branching and looping statements in Java are similar to **C**, **C++**.

Type Conversion or Type Casting

Conversion from one data type value to another data type value is known as type conversion or type casting. Java supports type conversion in two ways.

1. Implicit Conversion
2. Explicit Conversion

Implicit Conversion

Converting from one datatype value to another datatype value internally by the java language is known as implicit conversion. Java automatically converts one type value to another provided the following two conditions are satisfied.

1. **Destination and source variables types should be type compatible.** (i.e. the data types of variables should belong to same data type category.)

2. **Destination variable data type size should be larger than source variable data type size.**

Example:

1.
```
int x;
short y=20;
x=y;      //Correct : because x size 4 bytes and y size
          //2 bytes and both are type compatible.
```

2.
```
int x=30;
short y;
y=x;      // Error : because size of y is 2 bytes and x size is 4
          // bytes. Therefore destination variable(y) size is
          // not larger than source(x).
```

3.
```
int a=3000;
float f;
```

```
f = a ;  // Error : because the data types of variables are not
         // compatible i.e f belongs to float data type
         // category and a belongs to integer data type
         // category.
```

Explicit type conversion

When the above two conditions are not satisfying, then the java does not allow implicit conversion. Therefore we have to convert the source data type value to the destination data type explicitly and this conversion is known as explicit conversion. This can be done by using the type casting method.

Syntax: *destination_variable = (destination_variable_type) source_variable;*

Example:

```
int x=300;
byte y;
y= (byte) x;
      ↳ Explicit type casting.
```

In the above example, value of **x** is converted into 1 byte and stores into **y**. This conversion method is known as explicit type casting method.

When java does not convert automatically, then we should convert explicitly as shown above.

Automatic type promotion in expressions

In addition to assignments, there is another place where certain type conversions may occur, that is in **expressions.**

The type promotion Rules:

In addition to the elevation, **bytes** and **shorts** are promoted to **int.** Java defines several **type promotion rules** that apply to expressions. They are as follows:

First, all **byte** and **short** values are promoted to **int.**
If one operand is **long,** the entire expression is promoted to **long.**
If one operand is **float**, the entire expression is promoted to **float** .
If any operand is **double**, the result is **double**.

Example:

```
byte b=42;
short c=5;
int  j=12;
float x=10.55;
double y=20.66;
```

```
int h;
h=b+c*j;
```

In the above statement byte(**b**), short(**c**) variables values are automatically promoted to **int.** The resultant value obtained from the expression (**b+c*j**) will be **integer** value and assigns to integer **h** variable.

```
x=(j+c)+x;
```

In the above statement **(j+c)** evaluates to an integer which is added with float **x** and the resultant value will be float and assigns to **x** which is float variable.

```
h=(j+x)*y;  // error
```

The resultant value will be double and it can not be assigned to an integer variable **h** which is an error. To assign the expression result to **h** we have to use explicit type casting.

```
h=(int) ( (j+x)*y);
```

In the above statement, the resultant value double is type casted into integer and is assigned to **h** integer variable.

Arrays in Java

An array is a group of elements of same type and is referred by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

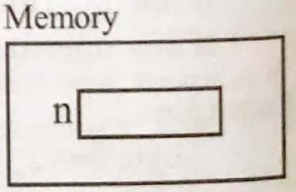
To declare one dimensional array in java, the array should be declared of specific data type and then the memory to the array should be allocated dynamically using **new** operator.

Syntax:

```
datatype arrayname[ ];
```

Example:

```
int n[ ];
```



In the above statement, **n** is the array name (known as reference name of array) of integers is declared but memory is not allocated. In java, memory should be allocated to the array only at runtime using **new** operator.

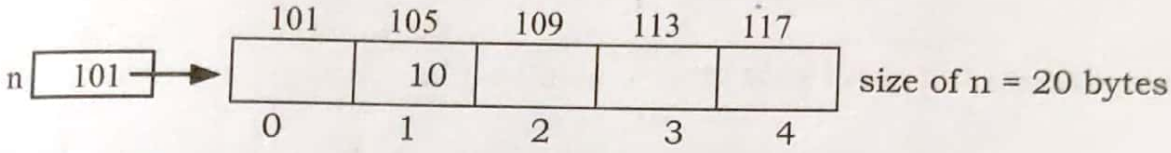
Syntax:

```
array_name=new datatype[no_of_elements];
```

Example:

```
n=new int[5];
```

In the above statement **5** elements of **int** memory is allocated to the array name **n** as shown below.



To access elements of array, we can use subscript/index numbers.

Example:

```
n [ 1 ] =10;   //stores 10 into n[1] element as shown above.
System.out.print("n [ 1 ] = " + n [ 1 ] );
```

output: n[1]=10

It is also possible to allocate memory to array when array is declared.

```
int n[ ]=new int[5];
```

Note: int n[5]; ⟶ error: can't define array with size.

Initialization of Arrays

Arrays can be initialized at the time of declaration of array. Based on the number of values initialized those number of elements of memory is allocated at run-time.

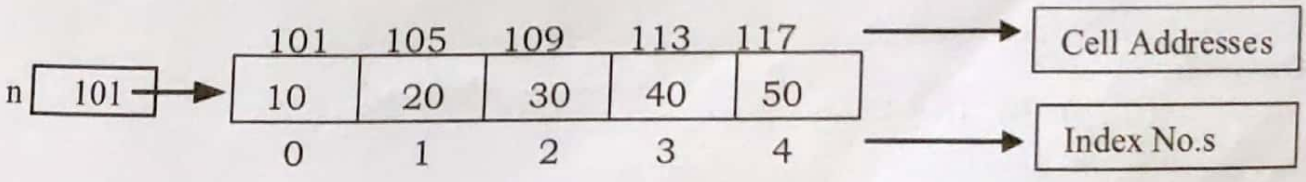
Syntax:

```
type array_name[]={value1, value2, ....};
```

Example:

```
int n [ ] ={ 10,20 ,30,40,50 };
```

In the above initialization, array **n** is allocated with five elements of integer memory and are stored with the values given in flower braces as shown below.



Example programs:

Bin> edit array1.java

```
class array1
{
    public static void main(String argv[ ])
    {
          int a[ ]={10,20,30,40,50};
          int i;
          for(i=0;i<5;i++)
          {
                System.out.print("\t" + a[ i ]);
          }
    }
}
```

Output:

10 20 30 40 50

Bin> edit array2.java

```
class array2
{
    public static void main(String argv[])
    {
          int a[ ], i;
          a=new int[5];
          a[0]=10;
          a[1]=20;
          a[2]=30;
          a[3]=40;
          a[4]=50;
          for(i=0;i<5;i++)
          {
                System.out.print("\t"+ a[ i ]);
          }
    }
}
```

Output:

10 20 30 40 50

Double (or) Two Dimensional Arrays

Defining double dimensional arrays are similar to single dimensional arrays, the only difference is, a two-dimensional array has two dimensions.

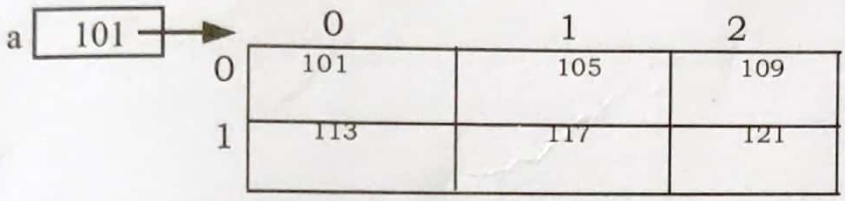
Syntax:

```
type array_name[][]=new data type [rows] [cols];
```

Example:

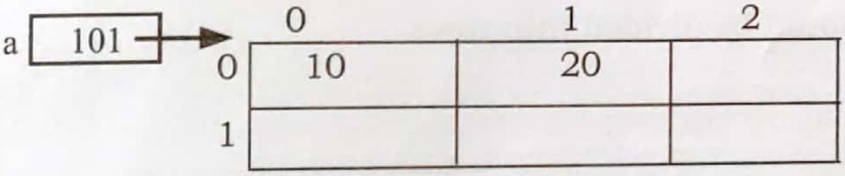
```
int a[ ][ ] =new int [2][3];
```

In the above example, array **a** is defined with **2 X 3** matrix as shown below.



To access the elements of two-dimensional array, we have to use row and column indexes.

```
a[0][0]=10;
a[0][1]=20;
```



It is also possible to allocate memory after the declaration of two-dimentional array as.

```
int a[ ][ ];
a=new int[2][3];
```

Bin> edit array3.java

```
class array3
{
    public static void main(String argv[ ])
    {
        int a[ ][ ]=new int[2][2];
        int i,j;
        a[0][0]=10;
        a[0][1]=20;
        a[1][0]=30;
        a[1][1]=40;
```

```
System.out.print("\n Values of a matrix...\n");
            for(i=0; i<2; i++)
            {
                for(j=0; j<2; j++)
                {
                    printf("\t " + a[i][j]);
                }
                System.out.print("\n");
            }
    }
}
```

Output:

Values of a matrix...

10	20	30
40	50	60

Initialization of Two-Dimentional Arrays

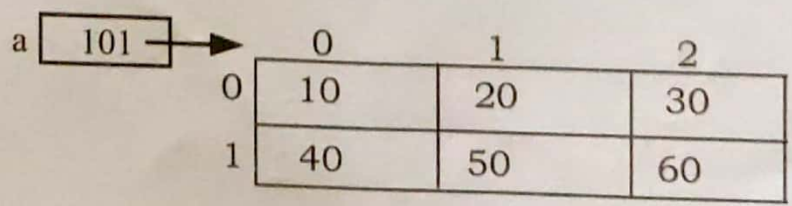
Two-dimentional arrays are initialized in the same way as single dimensional arrays but the values should be divided into rows.

Syntax:

```
type arrayname[ ][ ]= {
                        {value1, value2, . . },
                        {value3, value4, . . },
                                :
                      };
```

Example:

```
int a[ ][ ] = {
                {10,20,30},
                {40,50,60}
              };
```



Bin> edit array4.java

```
class array4
{
    public static void main(String argv[ ])
    {
          int a[ ][ ]={
                              { 10, 20, 30 },
                              { 40, 50, 60 }
                        };
          int i,j;

          System.out.print("\n Values of a matrix...\n");
          for(i=0; i<2; i++)
          {
                for(j=0; j<3; j++)
                {
                      System.out.print ("\t " + a[i][j]);
                }
                System.out.print("\n");
          }
    }
}
```

Output:

Values of a matrix...

10	20	30
40	50	60

Structure of Java Program

To write programs in java we have to follow the following structure of java program.

Comments
Package Declaration
Import Statements

Classes and Interfaces definitions

```
class <program_name>
{
        :
        public static void main(String argv[ ])
        {
            : // local variables definition
            : // programming statements
            :
        }
}
```

Comments

Comments are used for documenting the statements in the program. Comments are used for describing the statements in the program. Comments are ignored by the compiler and are un-executable statements. Java supports both single and multi-line commentation.

Single line comments:

```
// comment line
```

Example:

```
a=10; // a is assigned with 10
```

Multi-line comments:

```
/*
   comment line1
   comment line2
        :
        :
   comment lineN  */
```

Example:

```
/*
   This program prints addition of two numbers.
   Written on Dt=4-6-2010*/
```

Classes and Objects

Java is an object oriented programming language, therefore programs should be developed using classes and objects.

Class : A class is a collection of **instance variables** and **methods**.

Object : An object is an instantiation of class.

Example Program

Bin> edit ex2.java

```
class sample
{
    private int x;
    public void setx(int a) //sets value to x
    {
        x=a;
    }
    public void printx() // prints value of x
    {
        System.out.print("\n x=" + x);
    }
}
class ex2
{
    public static void main(String argv[])
    {
        sample obj=new sample();
        obj.setx(100);
        obj.printx();
    }
}
```

Bin> java exp2.java
Bin>java exp2

x=100

Explanation:

In the main, the statement

sample obj=new sample();

Here **obj** is the reference name of **sample** class and the **new** operator allocates memory of **sample** class to **obj**. The object contains **x** private variable of sample class. In this way, in java memory is allocated at runtime using **new** operator.

The statement **obj.setx(100)** calls the method present in the sample class and **100** is passed to **a** and then **a** is assigned to **x**.

The statement **obj.printx( )** calls the method in class where it prints the value of **x** on monitor as shown in output.

public static methods

The **public static** methods of a class can be accessed outside the class using class name without using object name. The **public** access specifier specifies that the method can be accessed outside the class. The **static** modifier specifies that the method can be accessed using classname.

Example program

Bin>edit static1.java

```
class A
{
        public static void show()
        {
                System.out.print("\n show of A..");
        }
}
class static1
{
        public static void main(String argv[])
        {
                A.show();
        }
}
```

Bin>javac static1.java
Bin>java static1
show of A..

Explanation:

The **A** class contain **show()** method which is **public static** method. In the **main()**, the **show()** method is called using **A** classname because public static methods can be accessed using classname.

Why main() method is declared as public and static

When a java program is executed at the command prompt using the java command as

Bin>java static1

The JVM takes the **static1** class name and using this class name it calls the **main()** method as **static1.main()**. Like this **main()** method is called from out side the class using class name therefore **main()** method should be declared as **public** and **static**.

Command-Line Arguments

It is also possible to pass values to **main()** method and it can be passed from command line because **main()** method is called from command-line. Passing arguments from command line to the **main()** method is known as command line arguments.

Values passed from command-line exist in the form of strings therefore the argument in the **main()** is defined as an array of String type.

public static void main(**String argv[ ]**)

argv is an array of String that can store any number of values passed from command-line

Example Program:

This program takes values from command-line and prints on the monitor.

Bin>edit cline.java

```
class cline
{
   public static void main(String argv[ ])
   {
     int i;
     System.out.print("\n No. of arguments="+argv.length);
     System.out.print("\n Argument values...");
     for(i=0; i<argv.length; i++)
     {
          System.out.print("\n"+argv[i] );
     }
   }
}
```

Bin>javac cline.java
Bin>java cline 100 apex 20.5

```
No. of arguments =3
Argument values...
100
apex
20.5
```

Explanation:

When the above program is executed with **java cline 100 apex 20.5** the **JVM** converts this execution statement as **cline.main ("100","apex","20.5")** the values are passed to main() as strings to **argv[]**.

In the main() method, **argv.length()** returns **3** because **argv** array contains three elements. In the for loop, **argv[i]** prints values of array one element after the other and we get the output shown above.

A program to print addition of two numbers using command-line arguments.

Bin>edit add1.java

```
class add1
{
    public static void main(String argv[])
    {
          int a,b,c;
          a=Integer.parseInt( argv[0] );
          b=Integer.parseInt( argv[1] );
          c=a+b;
          System.out.print("\n a=" +a + "\t b=" +b +"\t c="+c);
    }
}
```

Bin>javac add1.java

Bin>java add1 100 200

a=100 b=200 c=300

Explanation:

When the above program is executed, **100** and **200** values are passed to **argv[]** array. **100** and **200** values are stored in **argv[0]** and **argv[1]** as strings. In the main(), the statement

```
a=Integer.parseInt( argv[0] );
```

converts **argv[0]** value **"100"** into integer and assigns to **a**. Similarly, **argv[1]** value **"200"** converts into integer and assigns to **b**. Variables **a** and **b** are added to **c** and finally prints values of variables on the monitor as shown in above output.

Package declarations

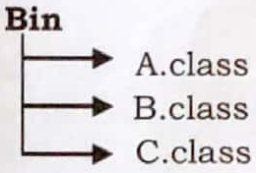
When a java program is compiled we get the **.class** files for each class present in the program. For example consider the following program

Bin>edit A.java

```
class A
{
        main(..)
        {
           ...
        }
}
class B { ... }
class C { ... }
```

Bin> javac A.java

The java compiler compiles the A.java program and generates three **.class** files. They are **A.class**, **B.class** and **C.class**. These .class files are stored in the working directory **Bin.**



Now, if we execute the program using the **A.class** file then the program executes successfully.

Bin> java A

No. error

We write another program in the same Bin directory.

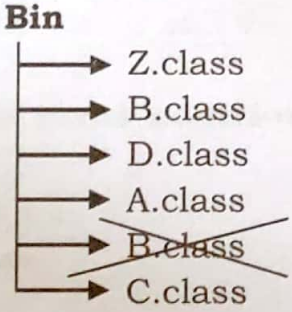
Bin>edit Z.java

```
class Z
{
        main(..)
```

```
            {
                ...
            }
        }
class B { ... }
class D { ... }
```

Bin> javac Z.java

The java compiler compiles the **Z.java** program and generates three **.class** files. They are **Z.class**, **B.class** and **D.class**. These **.class** files are stored in the working directory **Bin.**



Bin directory already consists of **B.class** file therefore old file is deleted and the latest **B.class** file generated from **Z.java** is stored. Therefore the **z.java** program executes successfully where as **A.java** program does not execute because the **B.class** file belong to **A.java** was deleted.

Bin> java Z

No. error, executes successfully

Bin>java A

Error: B.class of A.java was deleted.

As explained above when different java programs are compiled their may be a chance of two or more class files have same names and this is known as ***name space collision.*** In this process the older .class files are deleted and new files are stored, as a result the old java programs when re-executed they does not execute. This is the problem that arises in java.

To overcome the above problem packages concept can be used in java. For example consider the following programs.

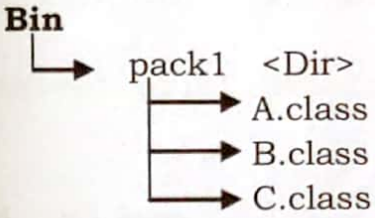
Bin>edit A.java

```
package pack1;
```

```
class A
{
      main(..)
      {
            ...
      }
}
class B { ... }
class C { ... }
```

Bin> javac –d . A.java

The java compiler compiles the **A.java** program and generates three **.class** files (**A.class**, **B.class** and **C.class**). These .class files are placed in **pack1** directory and this directory is stored in the working directory **Bin** as shown below.



Now, if we execute the program it executes successfully.

Bin> java pack1.A

//No error: Program executes successfully.

We write another program in the same Bin directory.

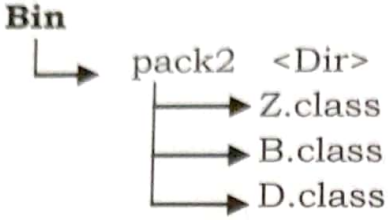
Bin>edit Z.java

```
package pack2;
class Z
{
      main(..)
      {
            ...
      }
}
class B { ... }
class D { ... }
```

Bin> javac –d . Z.java

The java compiler compiles the **Z.java** program and generates three **.class** files (**Z.class**, **B.class** and **D.class**). These .class files are placed in

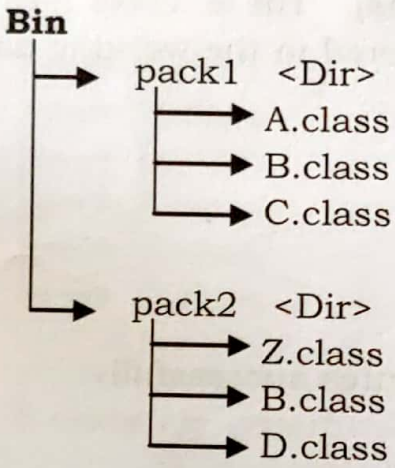
pack2 directory and this directory is stored in the working directory **Bin** as shown below.



Now, if we execute the program it executes successfully.

Bin> java pack2.Z

// No error: Program executes successfully.



As **B.class** files are stored in different directories (packages) as shown above, therefore their will be no name space collision.

Definition: A package is a container of classes.

Advantages:

1. Packages eliminate name space collision.
2. Two or more .class files with same name can be stored in different packages.
3. The classes can be classified and can be organized in hierarchical order.
4. The classes in packages can be reused or shared in other java programs.

Syntax:

```
package packagename1;
```

Package programs should be complied with the following syntax.

javac –d <path> pack_prog.java

-d parameter creates a directory on the name of package present in the pack_prog.java.

<path> specifies where the package directory should be placed.

Package programs should be executed using the following syntax.

java pack_name.classname

Example:

Bin>edit p1.java

```
package pack1;
public class p1
{
   public static void main(String argv[])
   {
      System.out.print("\n This is sample package program.");
   }
}
```

Bin>javac -d . p1.java

```
└→ pack1   <Dir>
     └→ p1.class
```

Bin>java pack1.p1

This is sample package program.

Explanation:

When the above program is compiled in Bin directory as

Bin>javac –d . p1.java

The java creates a new directory on the name of packagename **pack1** given in the program and the **.class** files(**p1.class**) are stored in the pack1 directory. This pack1 directory will be placed in **Bin** directory that is specified using **.** (current directory).

The above package program can be executed in the Bin directory as

Bin> java pack1.p1

The jvm executes the **p1.class** file present in pack1 directory and the output we get as

This is sample package program.

Note:

1. The classes in packages should be public otherwise the classes cannot be accessed outside the package.
2. With javac above we are using • (**dot**) which represent the working directory. It can also be complied as

   Bin> javac –d c:\program files\java\jdk1.6.0\Bin p1.java
3. The package statement should be the first statement in the program.
4. If the package directory already exist then the same directory is used for storing the .class files without creating the new directory.

Importing package classes

The classes present in packages can be used in other java programs by importing the classes of packages.

Syntax:

```
import packagename.classsname;
                or
import packagename.*;
```

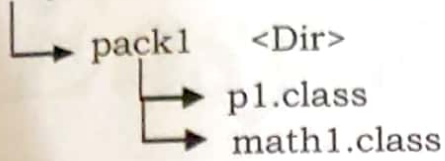
The first syntax imports only one class where as second syntax imports all classes (*) present in the package.

Example:

Bin> edit math1.java

```
package pack1;
public class math1
{
        public void sum(int x,int y)
        {
              System.out.print("\n sum="+(x+y));
        }
        public void square(int x)
        {
              System.out.print("\nsquare="+(x*x));
        }
}
```

Bin>javac -d . math1.java



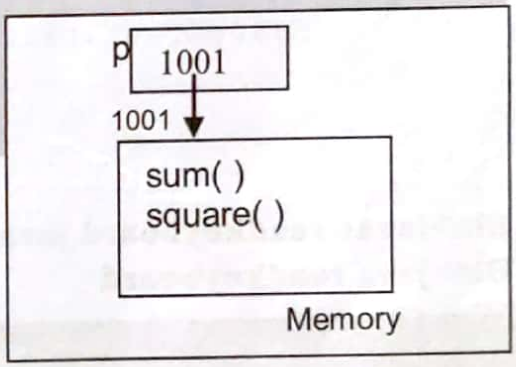
Explanation:

When the above program is compiled the jvm generates **math1.class** which is placed in **pack1** directory of **Bin**

Don't execute the **math1.class** file because it does not contain **main()**. This class can be used for importing into other programs.

Bin>edit m1.java

```
import pack1.math1;
  class m1
  {
      public static void main(String argv[])
      {
            pack1.math1 p=new pack1.math1();
            p.sum(5,6);
            p.square(5);
      }
  }
```



Bin> javac m1.java
Bin> java m1
sum=11
square=25

Explanation:

In the above program, the statement **import pack1.math1;** imports the **math1** class into the program. In the main(), an object **p** is instantiated to **math1** class present in **pack1** and using this object **sum()** and **square()** methods are executed and we get the output as shown above.

—— End of Structure of Java program ——

Reading values from keyboard

To read values from keyboard in a java program we can use InputStreamReader and BufferedReader classes of **java.io** package.

Note: For more information of InputStreamReader and BufferedReader classes refer page no.s 426, 428.

A program that reads two numbers from keyboard and prints its addition.

Bin>edit readkeyboard.java

```
import java.io.*;
class readkeyboard
{
   public static void main(String argv[])throws IOException
   {
      int a=0,b=0,c=0;
      InputStreamReader isr=new InputStreamReader(System.in);
      BufferedReader br=new BufferedReader(isr);
      try
      {
         System.out.print("\n Enter 2 numbers :");
         a=Integer.parseInt(br.readLine());
         b=Integer.parseInt(br.readLine());
         c=a+b;
         System.out.print("\nc="+c);
      }
      catch(NumberFormatException e)
      {
         System.out.print("\nError :"+e);
      }
   }
}
```

Bin>javac readkeyboard.java
Bin>java readkeyboard

```
 Enter 2 numbers :100
200

c=300
```

Explanation:

In the above program, the **InputStreamReader** class object **isr** is instantiated to **System.in** which is inputstream object of keyboard. The **isr** object reads values from keyboard. The **br** object of **BufferedReader** class reads data from the input stream reader object **isr**. The **readLine()** method of **BufferedReader** reads data from buffer which exist in string is converted to integer by **Integer.parse()** method. Values reads from keyboard stores into **a** and **b** variables and stores their addition into **c**. Finally, prints **c** on the monitor.