COMPUTER SCIENCE

B. Sc. II (CBCS) Semester-IV 2023-2024

2CS2 : RDBMS and Core Java

Unit-IV : Introduction to Java

PROF. V. V. AGARKAR

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

Unit – IV

Introduction to JAVA: History and evolution, Feature, JRE, JDK, JVM, Tokens of Java, Data types and Literals, Operators, Structure of Java Program, Access controls, modifiers, type conversion and casting, Control of Flow: Selection Statements, Iteration Statements. Command Line Argument, Arrays.

Introduction to JAVA

The Java programming language is a general-purpose, object-oriented language. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing, and unsafe.

James Gosling and colleagues at Sun Microsystems developed java language in the early 1990s, later it was acquired by Oracle Corporation. Unlike conventional languages which are generally designed either to be compiled to native (machine) code, or to be interpreted from source code at runtime, Java is intended to be compiled to a *bytecode*, which is then run by a Java Virtual Machine (JVM).

The Java language is mainly used for: mobile applications (especially Android apps), desktop applications, web applications, web servers, application servers, games, database connection and much more.

Java is a ***platform-neutral*** language. Java is the first programming language that is not tied to any particular hardware or operating system. Program developed in Java can be executed anywhere on any system.

History and Evolution of Java

James Gosling, Patrick Naughton, Chris Warth, Mike Sheridan, and Ed Frank initiated the Java language project in June 1991. The idea was to develop a language which was platform-independent and which could create embedded software for consumer electronic devices. It took 18 months to develop and had an initial name as ***Oak*** which was renamed to ***Java*** in 1995, due to copyright issues.

The popularity of Java language can be recognized to the popularity of World Wide Web and its capacity to create web pages. In 1995, Java version 1.0 was launched. In 1996, JDK 1.0 was released. In the year 1997, JDK 1.1 was released. Then the Java servlets developer kit was released. This resulted into increase of large number of Java developers.

In 1998, Java 2 with version 1.2 open source technology was launched. It gained more importance in 1998 because of the spread of the internet. In 1999, Java 2 Platform Standard Edition (J2SE) and Enterprise Edition (J2EE) were released. Then same year Java Server Pages (JSP) technology was released. The JSP deals with client and server side operations. Then in the year 2000, Java powered PDAs was launched.

In the year 2000, J2SE with SDK 1.3 was released. In the year 2002, J2SE with SDK 1.4 was released. In the year, 2003 mobile phones powered by Java were developed and used by the customers. In the year 2004, J2SE with JDK 5.0 was released. This is known as J2SE 5.0.

Feature of Java

The inventers of Java wanted to design a language which could offer solutions to some of the problems encountered in modem programming. They wanted the language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially describes Java with the following features:

1) Compiled and Interpreter
2) Platform Independent and Portable
3) Object Oriented
4) Robust and Secure
5) Distributed
6) Familiar, Simple and Small
7) Multithreaded and Interactive
8) High Performance
9) Dynamic and Extensible
10) Architectural Neutral

1) **Compiled and Interpreter:** Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two stage system. First, Java compiler translates source code into what is known is *bytecode* instructions. *Bytecode* are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program. Thus, Java is both a compiled and an interpreted language.

2) **Platform-Independent and Portable:** The most significant contribution of Java over other languages is its portability. Java programs can be easily moved from one computer system to another anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason why Java has become a popular language for programming on internet which interconnects different kinds of systems worldwide. We can download a Java applet from a remote computer onto our local system via internet and execute it locally.

   Java ensures portability in two ways. First, Java compiler generates *bytecode* instructions that can be implemented on any machine. Secondly, the sizes of the primitive data types are machine-independent.

3) **Object Oriented:** Java is a true object-oriented language. Almost everything in Java is an *object*. All program code and data reside within objects and classes. Java comes with an extensive set of *classes*, arranged in *packages*, which we can use in our programs by *inheritance*. The object model in Java is simple and easy to extend.

4) **Robust and Secure:** Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures serious errors and eliminates any risk of crashing the system.

   Security becomes an important issue for a language that made for programming on internet. Threat of viruses and abuse of resources are everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

5) **Distributed:** Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

6) **Familiar, Simple and Small:** Java is a small and simple language. Many features of C and C++ that are either redundant or sources of unreliable code are not part of Java. For example, Java does not use pointers, preprocessor header files, *goto* statement and many others. It also eliminates operator overloading and multiple inheritance.

   Familiarity is another striking feature of Java. To make the language look familiar to the existing programmers, it was modeled on C and C++ languages. Java uses many constructs of C and C++ and therefore, Java code "looks like a C++" code, In fact, Java is a simplified version of C++.

7) **Multithreaded and Interactive:** Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another. For example, we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer. This feature greatly improves the interactive performance of graphical applications.

8) **High Performance:** Java performance is impressive for an interpreted language, mainly due to the use of intermediate *bytecode*. According to Sun, Java speed is comparable to the native C/C++. Java architecture is also designed to reduce overheads during runtime. The inclusion of multithreading enhances the overall execution speed of Java programs.

9) **Dynamic and Extensible:** Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

   Java support functions written in other languages such as C and C++. These functions are known as *native methods*. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at runtime.

10) **Architectural Neutral:** Architecture represents processor. Java programs are compiled into *bytecode* format which does not depend on any machine architecture but can be easily translated into a specific machine by a Java Virtual Machine (JVM) for that machine.

Java Development Kit (JDK)

JDK is an abbreviation for Java Development Kit. It is a software development environment used for developing Java applications and applets. It includes all the tools, executable and binaries required to compile, debug and execute a Java program. Some of the contents of JDK are the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

JDK is platform dependent i.e. there is separate installer for Windows, Mac, and UNIX systems. The version of JDK represents version of Java. It physically exists. Following Table 4.1 lists some of the JDK tools and their descriptions:

Table 4.1: Java Development Tools

Tools	Description
appletviewer	Enables to run Java applets (without using a Java-compatible browser).
javac	A Java compiler, which translate Java source code to bytecode files that the interpreter can understand.
java	A Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javap	A Java disassemble, which enables us to convert bytecode files into a program description.
javah	Produce header files for use with native methods.
javadoc	Creates HTML-format document from Java source code files.
jdb	Java debugger, which helps to find errors in the programs.

Java Run-time Environment (JRE)

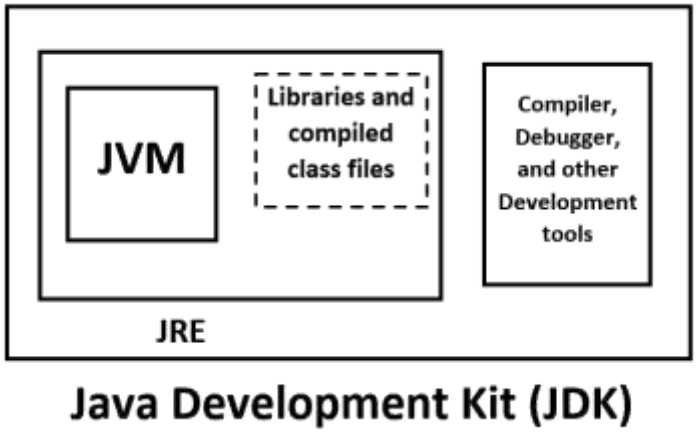
Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK). It is a freely available software distribution which has Java Class Library, specific tools, and a stand-alone JVM. It is the most common environment available on devices to run java programs. The Java source code gets compiled and converted to Java bytecode. To run this bytecode on any platform, it requires JRE. The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system.

Java Virtual Machines (JVM)

Java compiler produces ***bytecode*** for a machine that physically does not exist. This machine is called the **Java Virtual Machine** and it exists only inside the computer memory. JVM is a part of JRE (Java Run Environment). JVM makes java platform independent.

Java Virtual Machine (JVM) is an engine that provides runtime environment in which java ***bytecode*** can be executed (i.e. JVM is a program that enables a computer to run Java programs). JVM is a part of Java JRE. JVM is available for many hardware and software platforms. JVM analyze the *bytecode*, interprets it, and execute the same *bytecode* to display the output. That is, JVM converts Java *bytecode* into machines language.

Note That, following figure-4.1 shows the relationship between JDK, JRE and JVM:



[Fig. 4.1 : Relationship between JDK, JRE and JVM]

Java Tokens

A token is a smallest individual element of a program that is meaningful to the computer. During compilation of a program, the compiler scans the text in your source code and extracts individual tokens. The Java compiler uses it for constructing expressions and statements. Generally, a Java program is a collection of tokens, comments and white spaces. Java language includes five types of tokens. They are:

- Reserved Keywords
- Identifiers
- Literals
- Operators
- Special symbols

Keywords

Keywords are predefined; reserved words used in Java programming that have special meanings to the compiler. Java language has reserved 50 words as keywords. Table 4.2 lists these keywords.

Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods, objects or any other identifiers. All keywords are to be written in lower-case letters, since Java is case-sensitive.

Table 4.2 : Java Keywords

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

Note:- The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

Identifiers

Identifiers in Java are symbolic names used for identification. A Java identifier is a name given to a package, class, interface, method, object, array or variable. However, in Java, there are some keywords or reserved words; that cannot be used as an identifier.

- ***Rules for defining Java Identifiers***

There are certain rules for defining a valid Java identifier. All these rules must be followed; otherwise compile-time errors can occur. Following are the rules:

Rule 1 : An identifier can only contain alphabetic characters [A-Z] or [a-z], numbers [0-9], underscore (_) and a dollar sign ($).

Rule 2 : An identifier should not contain any blank space.

Rule 3 : Identifiers should not start with digits [0-9]. The starting character should be alphabet [A-Z] [a-z], dollar ($) or underscore (_).

Rule 4 : Java identifiers are case-sensitive.

Rule 5 : There is no limit on the length of the identifier in Java, but it is advisable to use an optimum length of 4 – 15 letters only.

Rule 6 : Reserved words (keywords) cannot be used as identifiers.

Variable

A variable is the basic unit of storage in a program. It is a data container which stores the data value during Java program execution. A variable is the name of allocated memory area. All the operations done on the variable affects the allotted memory location. Every variable is assigned a data type that designates the type and range of value it can hold. The value stored in a variable can be changed during program execution. In Java, all variables must be declared before use. Variable names are identifiers, so all the rules for naming identifiers are also applicable to naming variables.

- ***Declaring (or creating) Variables***

In Java, the variables have to be declared before it is using in the program. As soon as the compiler gets a variable declaration/initialization command, it reserves memory with the variable name mentioned in the program. Variable declaration does four things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. It reserves enough memory for the variable.
4. The place of declaration (in the program) decides the scope of the variable.

The general declaration form of a variable is:

```
type variablename1 [,variablename2, ...];
```

If declaring more than one variable, then variable names are separated by commas. A declaration statement must end with a semicolon.

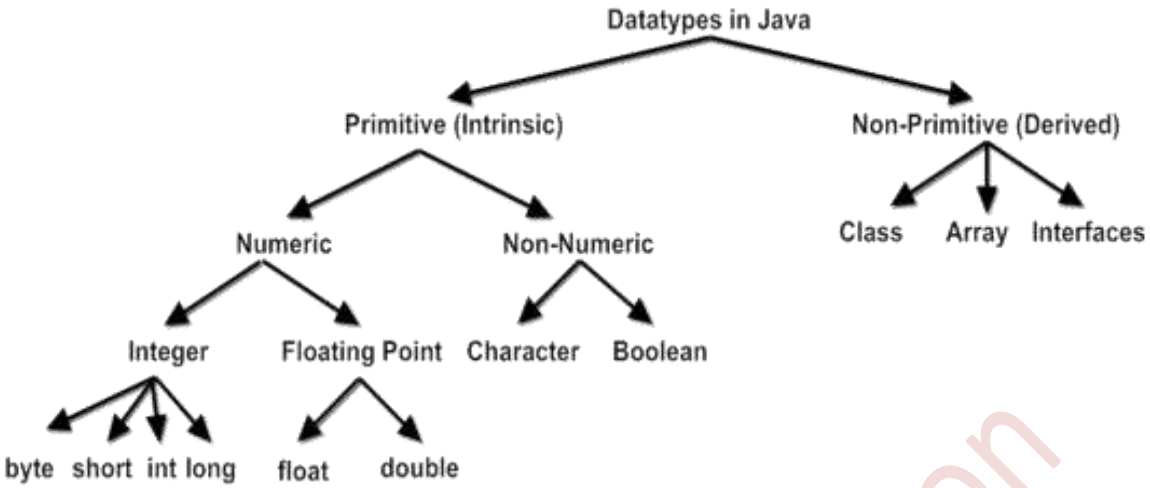
Example :-

Some valid declarations are:

```
int  count;
float  x, y;
char  c1, c2, c3;
```

Data types

Data types specify the size and type of values that can be stored. Every variable in Java has a data type. Data types in Java under various categories are shown in Fig. 4.4.



[**Fig. 4.4:** Data types in Java]

Data types in Java can be categorised mainly in two types:

1. Primitive (Intrinsic) Data Types
2. Non-Primitive (Derived) Data Types

1. Primitive Data Types

Primitive datatypes are predefined (built-in) by the Java language and named by a keyword. There are two types of primitive data types: Numeric and Non-numeric.

A) Numeric data types:

Numeric data types are further divided into Integer and Floating point data types.

a) Integer types

Integer types can hold whole numbers such as 512, -87, and 1025. The size of the values that can be stored depends on the selected integer data type. Java supports four types of integers: byte, short, int, and long. Java does not support the concept of unsigned types and therefore all Java values are signed meaning they can be positive or negative. Table 4.3 shows, the memory size and range of all the four integer data types.

Table 4.3: Size and range of Integer type

Type	Size	Minimum Value	Maximum Value
`byte`	1 byte	-128	127
`short`	2 byte	-32,768	32,767
`int`	4 byte	-2,147,483,648	2,147,483,647
`long`	8 byte	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

b) Floating point types

Floating point type holds numbers containing fractional parts such as 72.95 and -8.123. There are two kinds of floating point storage in Java: the **float** type values are single-precision numbers while the **double** types represent double-precision numbers. Table 4.4 gives the size and range of these two types.

Table 4.4: Size and range of Floating Point type

Type	Size	Minimum Value	Maximum Value
`float`	4 byte	3.4e-038	3.4e+038
`double`	8 byte	1.7e-308	1.7e+308

B) Non-Numeric data types:

There are two types of non-numeric data types in Java: Character and Boolean type.

a) Character type

To store character constants in memory, Java provides a character data type called `char`. The `char` type assumes a size of 2 bytes but, it can hold only a single character.

b) Boolean Type

Boolean type is used to test a particular condition during the execution of the program. There are only two values that a boolean type can take: `true` or `false`, Boolean type is denoted by the keyword `boolean` and uses only one bit of storage.

2. Non-Primitive (Derived) Data Types

Non-Primitive data types in Java are not pre-defined data types. It is a data type which has to be created by a programmer as needed. There are many types of Non-Primitive data type such as array, class, interface, etc.

a) Array

An *array* is a group of homogeneous data items that share a common name. Arrays offer a convenient means of grouping related information.

b) Classes

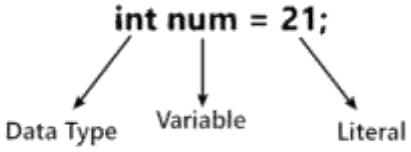
Class defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class.

c) Interface

An interface is basically like a class, but interfaces define only abstract method and final data items.

Literals

In Java, literal is a notation that represents a fixed value in the program. Literals are the constant values that appear directly in the program. It can be assigned directly to a variable. Thus literals are also known as **constants**.



- ***Types of Java Literals***

In Java, there are five types of literals: integer literals, floating-point literals, character literals, string literals and Boolean literals.

1) Integer Literals

An integer literal is a sequence of digits. There are four types of integer literal:

i) Decimal Literals

These are literals that consist of digits from `0` to `9`. It may have a positive (`+`) or negative (`–`). Note that between numbers commas and non-digit characters are not permitted. For example: `5678`, `+657`, `-89`, etc.

ii) Octal Literals

These are literals that consist of digits from `0` to `7` and proceeded by `0` (zero). For example: `045`, `05326`.

iii) Hexadecimal Literals

These are literals that consist of digits from `0` to `9` and character from `A` to `F` or `a` to `f`. Also these literals are preceded by `0x` or `0X` (zero-x). For example: `0x1A9`, `0XA8D`.

iv) Binary Literals

These are literals that consist of digits `0` and `1` and preceded by `0b` or `0B` (zero-b). For example: `0b1101`, `0B0101`.

2) Floating Point Literals

Java has two kinds of floating-point numbers: `float` and `double`. By default, every floating-point literal is of `double` type. But we can specify floating-point literal as `float` type by suffixed with `f` or `F`. We can specify explicitly floating-point literal as `double` type by suffixed with `d` or `D`. For Floating-point data types, we can specify literals in only decimal form, and we can't specify in octal and hexadecimal forms. For examples: `123.45`, `3.14e0`, `1.0e-6D`, `3.14F`.

3) Character Literals

A character literal represents a single character that is enclosed in a single quote (`''`). A character literal is that it must contain a single character enclosed within a single quote. A `char` data type is used to represent a character literal. For example:

```
char ch = 'A';
```

Java allows having certain non-graphic characters as character literals. Non-graphic characters are those that can't directly take from the keyboard; example backspaces, tabs, etc. These non-graphic characters can be represented as ***escape*** sequences.

4) String Literals

String literal is a sequence of characters that is enclosed between double quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc. For example: "Jack", "12345", "\n", etc.

5) Boolean Literals

Boolean literals allow only two values and thus have two literals– *true*: it represents a real Boolean value and *false*: it represents a false Boolean value. For example,

```
boolean b = true;
boolean d = false;
```

Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions. Java supports a rich set of operators. Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

1) Arithmetic operators

Arithmetic operators are used to perform arithmetic operations on variables and primitive data types. Java supports all the basic arithmetic operators. They are listed in Table 4.5:

Table 4.5: Arithmetic Operators

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Reminder)

Arithmetic operators are used as shown below:

```
a - b          a + b
a * b          a/b
a % b          -a * b
```

Here *a* and *b* may be variables or constants and are known as *operands*.

2) Relational operators

Relational operators are used to compare two quantities on either side. Java supports six relational operators in all. These operators and their meanings are shown in Table 4.6.

Table 4.6: Relational Operators

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	equal to
!=	not equal to

An expression such as

```
a < b      or      x > 20
```

containing a relational operator is termed as a ***relational expression***. The value of relational expression is either true or false.

3) Logical operators

Logical operators allow making a decision based on multiple conditions. Each operand is considered as a condition that can be evaluated to a true or false value. These values are then used to determine the overall value of the logical operator. Java supports three logical operators. These operators and their meanings are shown in Table 4.7.

Table 4.7: Logical Operators

Operator	Meaning
&&	logical AND
||	logical OR
!	logical NOT

An expression which combines two or more relational expressions is termed as a ***logical expression*** or a ***compound relational expression***.

Some examples of the usage of logical expressions are:

1. `if(age > 55 && salary < 1000)`
2. `if(number < 0 || number > 100)`

4) Assignment operators

Assignment operators are used to assign the value of an expression to a variable. We already have seen the usual assignment operator, '='. In addition, Java has a set of '*shorthand*' assignment operators which are used in the form:

```
v op= exp;
```

Where *v* is a variable, *exp* is an expression and *op* is a Java binary operator. The operator `op=` is known as the ***shorthand assignment*** operator. Some of the commonly used shorthand assignment operators are illustrated in Table 4.8:

Table 4.8: Shorthand Assignment Operators

Operator	Example	Meaning
+=	a += 10	a = a + 10
-=	a -= 5	a = a - 5
*=	a *= 10	a = a * 10
/=	a /= 2	a = a / 2
%=	a %= 2	a = a % 2

5) Increment Decrement operators

Java has two very useful operators that are the increment and decrement operators:

`++` and `--`

The operator `++` adds 1 to the operand while `--` subtracts 1. Both are unary operators and are used in the following form:

```
++m;        or    m++;
-- m;       or    m --;
```

`++m;` is equivalent to `m = m+1;` (or `m += 1;`)

`--m;` is equivalent to `m = m-1;` (or `m -= 1;`)

While `++m` and `m++` mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m  =  5;
y  =  ++m;
```

In this case, the value of `y` and `m` would be `6`. If we rewrite the above statement as:

```
m  =  5;
y  =  m++;
```

then, the value of `y` would be `5` and `m` would be `6`. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

6) Conditional operators

The character pair `?:` is a ternary operator available in Java. This operator is used to construct conditional expressions of the form :

```
exp1  ?  exp2  :  exp3
```

Where, *exp1*, *exp2*, and *exp3* are expressions.

The operator `?:` works as: *exp1* is evaluated first, if it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the conditional expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the conditional expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example:

```
a = 10, b = 15;
x = (a>b)? a : b;
```

In this example, `x` will be assigned the value of `b`. This can also be achieved using the `if…else` statement as follows:

```
if (a > b)
      x  =  a;
else
      x  =  b;
```

7) Bitwise operators

Java supports special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double, Table 4.9 lists the bitwise operators:

Table 4.9: Bitwise Operators

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

8) Special operators

Java supports special operators such as `instanceof` operator and member selection operator (.).

a) `Instanceof` Operator

The `instanceof` is an object reference operator and returns *true* if the object on the left-hand side is an instance of the class given on the right-hand side. This operator is used to determine whether the object belongs to a particular class or not. Example:

```
person instanceof student
```

is ***true*** if the object **person** belongs to the class **student**; otherwise it is ***false***.

b) `Dot` Operator

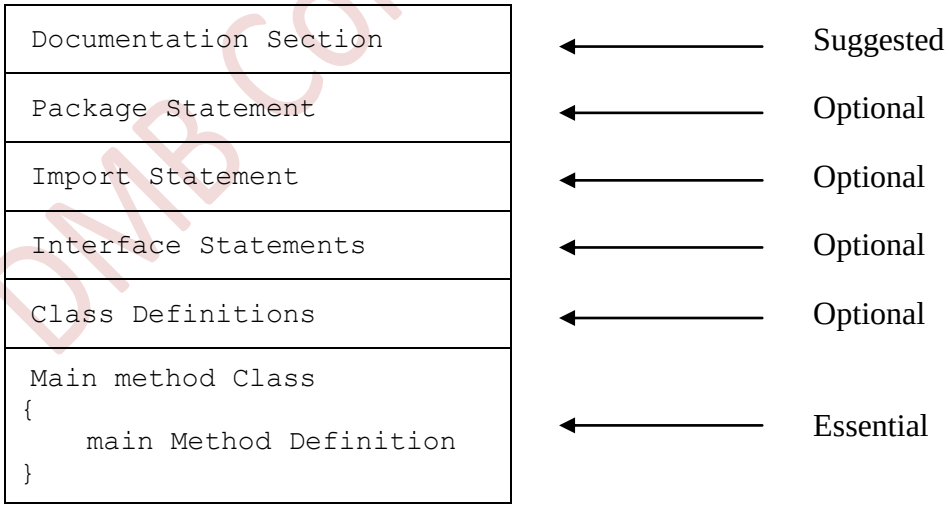
The dot operator (.) is used to access the instance variables and methods of class objects. Examples:

```
Person1.age          // Reference to the variable age
Person1.salary()     // Reference to the method salary( )
```

It is also used to access classes and sub-packages from a package.

Structure of Java Program

A Java program may contain many classes of which only one class defines a `main` method. Classes contain data members and methods that operate on the data members of the class. Methods may contain data type declarations and executable statements. To write a Java program, we first define classes and then put them together. A Java program may contain one or more sections as shown in Fig. 4.3.



[**Fig. 4.3:** General Structure of Java program]

Documentation Section

The documentation section is optional for a Java program. It includes basic information about a Java program. The information includes the author's name, date of creation, program name, and description of the program to understand code. It improves the readability of the program. Java compiler ignores whatever written in the documentation

section during the execution of the program. To write the statements in the documentation section, comments are used. The comments may be single-line (//), multi-line (/* … */), and documentation comments (/** … */).

Package Statement

Java that allows you to declare your classes in a collection called *package*. A package is a group of classes that are defined by a name. There can be only one package statement in a Java program and it has to be at the beginning of the code before any class or interface declaration. This statement is optional.

For example, following statement declares a package `student` and informs the compiler that the classes defined here belong to `student` package:

```
package student;
```

Import Statements

Many predefined classes are stored in packages in Java. An `import` statement is used to refer to the classes stored in other packages. An `import` statement is always written after the `package` statement but it has to be before any class declaration. Many classes can be imported in a single program and hence multiple `import` statements can be written. This is similar to the `#include` statement in C.

For example: Take a look at the examples given below to understand how import statement imports a specific class or all classes from a package:

1) `import java.util.Date;`
   This imports only the `Date` class from the java `util` package.
2) `import java.applet.*;`
   This imports all the classes from the java `applet` package.
3) `import student;`
   This imports the `student` class.

Interface Statements

This section is used to specify an interface in Java. It is an optional section which is mainly used to implement multiple ***inheritance*** in Java. An interface is a lot similar to a class in Java but it contains only constants and method declarations and cannot be instantiated. To declare an interface, `interface` keyword is used.

Class Definitions

A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. The `class` keyword is used to define the class. The class contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the `main()` method.

Main Method Class

Every Java program requires a `main()` method as its starting point i.e. the execution of all Java programs starts from the `main()` method. The `main()` method must be inside the class. Inside the `main()` method, objects can be created and call the methods. On reaching the end of `main()` method, the Java program terminates.

Access controls

Access Control in Java refers to the mechanism used to restrict or allow access to certain parts of a Java program, such as classes, methods, and variables. Access control determines which classes and objects can access specific codes or data within a program. By controlling access to different parts of the program, Java's access control mechanism promotes code encapsulation, and information hiding, and reduces the errors and security vulnerabilities in the program. Access control in Java can be implemented by using access control modifiers, which are keywords placed before the declaration of the class member.

Modifiers

Access control modifiers in Java are keywords that can be used to control access to classes, fields, and methods. Access control modifiers determine the level of access that other classes or objects have to a particular class, field, or method. There are two types of modifiers in Java: *access modifiers* and *non-access modifiers*.

The four ***access control modifiers*** in Java are:

1. **private**

   The `private` access control modifier in Java is used to restrict access to a class member to only within the same class. This means that a private member cannot be accessed from outside of the class, including from any subclass of the class.

2. **default**

   The access level of a `default` modifier is only within the same package. This means that a default member cannot be accessed from outside of the package. If you do not specify any access level, it will be the default.

3. **protected**

   The `protected` access control modifier in Java is used to provide access to a class member within the same class, any subclass of the class, or any class within the same package. This means that a protected member can be accessed from within the same class, any subclass of the class, or any class within the same package, but cannot be accessed from any class outside of the package, even if it is a subclass of the protected class.

4. **public**

   The `public` access control modifier in Java is used to provide unrestricted access to a class member from any other class, including classes that are not in the same package. This means that a public member can be accessed from within the class, outside the class, within the package and outside the package.

There are many ***non-access modifiers***, such as static, abstract etc.

Type conversion and casting

Typecasting is the process of converting the value of a data type into another data type. This conversion is done either automatically or manually. The compiler performs the automatic conversion, and a programmer does the manual conversion.

To use a variable in a particular way in automatic conversion, we need to explicitly tell the Java compiler to convert a variable from one data type to another data type.

- ***Types of Type Casting***

1. Widening Type Casting

Widening type casting is the process of converting a lower data type to a higher data type. It is also known to as ***implicit conversion*** or ***casting down***. This process is performed automatically and is safe, as there is no risk of data loss. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

```
byte → short → char → int → long → float → double
```

(From left to right: Lower data type to Higher data type)

2. Narrowing Type Casting

Narrowing type casting is the process of converting a higher data type to a lower one. It is also known as ***explicit conversion*** or ***casting up***. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

```
double → float → long → int → char → short → byte
```

(From left to right: Higher data type to Lower data type)

Syntax:

```
datatype variableName = (datatype)value;
```

Example:

```
double a = 166.66;
long b = (long)a;
```

Control of Flow

Java compiler executes the statements in the program sequentially from top to bottom, in the order that they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. Control flow statements, however, break up the flow of sequential execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. Java provides three types of control flow statements:

1. Selection (Decision Making) statements
2. Iteration statements
3. Jump statements

1) Selection (Decision Making) Statements

The Selection or decision-making statements decide which statement to execute and when. These statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. That is, a certain block of code is executed when the condition is fulfilled, otherwise another block is executed.

Java supports two selection or decision-making statements: **if** and **switch**.

a) If Statements

The `if` statement is used to decide which block of statements will execute depending on a condition. There are various types of `if` statement in Java:

i) if statement
ii) if-else statement
iii) if-else-if ladder
iv) nested if statement

i) if statement

The `if` statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not, i.e. if a condition is true then a block of statement is executed otherwise not.

The `if` statement checks a particular condition; if the condition evaluates to true, it will execute a statement or a set of statements. Otherwise, if the condition is false, it will ignore that statement or set of statements. The test expression of `if` statement must be of Boolean type. The syntax of simple `if` statement is:

```
if (condition)
        statement1;
```

Where, *statement1* may be a single statement or a compound statement enclosed in curly braces *i.e.* a *block*. The *condition* is any expression that returns a *Boolean* value.

First the *condition* is evaluated and if it is true, then *statement1* is executed. But if the *condition* evaluates to false, *statement1* will be skipped and control will exit the `if` statement. For examples:

1.
```
if(amount >= 5000)
        discount = 5;
```

2.
```
if(mark >= 33)
 {
    Grace = 3;
    cando = 4;
 }
```

ii) if-else statement

The `if-else` statement can perform two different operations, i.e., one is for the correctness of the condition, and other is for the incorrectness of the condition. The if-else statement executes a block (*say* `if-block`) of code if a specified condition is `true` and if the condition is `false`, another block (*say* `else-block`) of code can be executed. *In any situation the* `if` *and* `else` *block cannot be executed simultaneously.* The syntax of `if-else` statement is:

```
if (condition)
{
      Statement block-1;
}
else
{
      Statement block-2;
}
```

Where, *statement block-1* and *statement block-2* may be a single statement or a compound statement enclosed in curly braces *i.e.* a *block*. The *condition* is any expression that returns a *Boolean* value.

First the *condition* is evaluated if it is true, then *statement block-1* is executed. Otherwise, *statement block-2* is executed. In no case will both statements be executed.

Example:

```
if(age >= 18)
      { System.out.println("Eligible to vote"); }
else
      { System.out.println("Not eligible to vote"); }
```

iii) if-else-if ladder

The `if-else-if` ladder helps user to decide from multiple conditions. The `if` statements are executed from the top to down. As soon as one of the conditions in the `if` is true, the statement associated with that `if` is executed, and the rest of the `else-if` ladder is bypassed. If none of the conditions is `true`, then the final else statement will be executed. The syntax of `if-else-if` ladder is:

```
if(condition1)
        statement1;
else if(condition2)
        statement2;
else if(condition3)
        statement3;
...
else
        default-statement;
```

Where, the *condition1*, *condition2*, *condition3*, … are any expression that returns a *Boolean* value. The *statement1*, *statement2*, *statement3*, …, *default-statement* may be a single statement or a compound statement enclosed in curly braces *i.e.* a *block*.

Example:

```
if(mark >= 75)
      division = "Distinction";
else if(mark >= 60)
      division = "First class";
else if(mark >= 45)
      division = "Second class";
else if(mark >= 35)
      division = "Third class";
else
      division = "Fail";
```

iv) Nested-if statement

A nested `if` statement is an `if` statement placed inside another `if` statement as in the `if` body or the `else` body. Nested `if` statements are often used when you must test a combination of conditions before deciding on the proper action.

Example:

```
if (i == 10)
{
      if (j < 20)
            a = b;
}
else
{
      a = d;
      if(k > 100)
            c = d;
      else
            a = c;
}
```

b) Switch Statement

The `switch` statement in Java language is a multi-way branch statement. It tests the value of a given variable (or expression) against a list of `case` values and when a match is found, a block of statements associated with that `case` is executed. The `switch` statement works with `byte`, `short`, `int`, `long`, `enum` types, `char` and `String` data. The general form of `switch` statement is as follows:

```
switch (expression)
{
      case value-1:
                  block-1
                  break;
      case value-2:
                  block-2
                  break;
      ...
      ...
    [ default:
                  default-block
                  break; ]
}
```

The `expression` is evaluated once. Basically, the `expression` can be a `int`, `byte`, `short`, `char`, or `String` data types. There can be *one* or *n* number of case values *value-1*, *value-2*, …. for a switch statement. The case value must be of switch `expression` type only. The case value must be literal or constant. It doesn't allow variables. The case values must be unique. If case values are duplicates, it renders compile-time error. *block-1*, *block-2*, ….. are statement lists and may contain zero or more statements.

Each `case` statement can have a `break` statement which is optional. When control reaches to the `break` statement, it jumps the control after the `switch` statement. If a `break` statement is not found, it executes the next `case`. The `default` is as optional case. When present, it will be executed if the value of the expression does not match with any of the `case` values.

When the switch is executed, the value of the `expression` is successively compared against the values *value-1*, *value-2*, …. If a `case` is found whose value matches with the value of the `expression`, then the block of statement that follows the case are executed.

Example :

```
int day;
switch (day)
{
      case 1:
                  System.out.println("Monday");
                  break;
      case 2:
                  System.out.println("Tuesday");
                  break;
      case 3:
                  System.out.println("Wednesday");
                  break;
      case 4:
                  System.out.println("Thursday");
                  break;
      case 5:
                  System.out.println("Friday");
                  break;
      case 6:
                  System.out.println("Saturday");
                  break;
      case 7:
                  System.out.println("Sunday");
                  break;
      default:
                  System.out.println("Invalid day");
                  break;
}
```

2) Iteration Statements

The process of repeatedly executing a block of statements is known as *looping* and this can be done using iteration statements. The statements in the block may be executed any number of times, from *zero* to *infinite* number. If a loop continues forever it is called an *infinite loop*. The Java language provides following three iterative statements:

a) for statement
b) while statement
c) do statement

a) The *for* Statement

The `for` loop is an *entry-controlled* loop and provides a concise way of writing the loop structure. The `for` loop is used to iterate a block of code several times. If the number of iterations is fixed, it is recommended to use `for` loop. The `for` statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter and easy structure of looping. The syntax of the `for` loop is:

```
for {initialization;  test condition;  increment)
{
     Statement(s);    //Body of the loop
}
```

The `for` loop consists of four parts:

1. **Initialization**: It is the initialization of loop-control/counter variable. It is executed only once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable.
2. **Test Condition**: It is the condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either `true` or `false`.
3. **Increment/Decrement**: It increments or decrements the control/counter variable value. It is executed each time after the statement (body of loop) has been executed.
4. **Statement**: The statement(s) of the loop is executed each time until the ***condition*** is false.

While executing the ***for*** statement, firstly the *initialization* of the *loop-control/counter* variable is done. Then the value of the control variable is tested using the *test condition*. If the condition is true, the body of the loop is executed. After execution of last statement in the body of loop, the control is transferred back to the ***for*** statement. Now, the *loop-control/counter* variable is *incremented/decremented* and the new value of the control variable is again tested in the test condition. If the condition is true, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition. When the test condition becomes false, the loop is terminated and the execution continues with the statement that immediately follows the loop.

Examples:

1) Consider the following segment of a program

```
for(x=0; x<=9; x++)
{
      System.out.println(x);
}
```

This for loop is executed 10 times and prints the digits 0 to 9 in one line.

2) Consider the following segment of a program

```
for(x=9; x>=0; x--)
     System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9.

***Note that**, braces are optional when the body of the loop contains only one statement.*

- **Additional Features of `for` Loop**

  Java's additional feature of the ***for*** loop allows two or more variables to control ***for*** loop, multiple statements can be included in both the *initialization* and *increment* portions of the ***for*** statement. Each such use is separated by a comma. Example:

```
for(a=1,b=4;  a<b;  a++,b--)
{
      System.out.println("a = " + a);
      System.out.println("b = " + b);
}
```

b) The *while* Statement

The `while` loop is an *entry-controlled* loop statement. The `while` loop is used to iterate a block of code several times. If the number of iterations is not fixed, it is recommended to

use `while` loop. It repeatedly execute a body of loop as long as the test condition is true. As soon as the test condition becomes false, the loop automatically terminates. The syntax of the `while` loop is:

```
while(test condition)
{
      Body of the loop
}
```

While executing the ***while*** statement, firstly the *test condition* is evaluated. If the condition is true, the body of the loop is executed. After execution of last statement in the body of loop, the control is transferred back to the ***while*** statement. Now, again the test condition is evaluated. If the condition is satisfied, the body of the loop is again executed. This process continues till the test condition is true. When the test condition becomes false, the loop is terminated and the execution continues with the statement that immediately follows the loop. For example:

```
sum = 0;
n = 1;
while(sum <= 50)
{
      sum = sum + n;
      n = n+1;
 }
System.out.println("Sum = " + sum);
```

The body of the loop is executed number of times for n = 1, 2, 3, ... till the sum becomes 50 or greater than 50.

c) The *do* Statement

The `do-while` loop is called an *exit control* loop. Therefore, unlike `while` loop and `for` loop, the `do-while` check the condition at the end of loop body. The `do-while` loop is used to iterate a block of code several times. If the number of iterations is not fixed and wants to execute the body of the loop at least once, it is recommended to use `do-while` loop. It repeatedly execute a body of loop as long as the test condition is true. As soon as the test condition becomes false, the loop automatically terminates. The `do-while` loop is executed at least once because condition is checked after loop body. The syntax of the `do-while` loop is:

```
do
{
  body of the loop
}
while(test condition);
```

While executing the ***do-while*** statement, on reaching the ***do*** statement, the program proceeds to evaluate the *body of the loop* first. At the end of the loop, the *test condition* in the ***while*** statement is evaluated. If the condition is true, the program jumps back on ***do*** statement and continues to evaluate the body of the loop once again. This process repeatedly continues as long as the *test condition* is true. As soon as the test condition becomes false, the loop will be terminated and the control goes to the next statement immediately after the ***do-while*** statement.

Example:

```
int n=1, sum=0;
do
{
      sum = sum + n;
      n = n+1;
 }
while (sum <= 50)
System.out.println("Sum  =   "   +  sum);
```

Command Line Argument

A Java application can accept any number of arguments from the command line. The Java command-line argument is an argument i.e. passed at the time of running the Java program from the console (i.e. command prompt) and it can be used as an input.

The command-line arguments are directly passes to the `main()` method. The string or primitive data types such as `int`, `double`, `float`, `char`, etc can be passes as command-line arguments. When the command-line arguments are passed, they are converted to strings and stored in String array `args[]`. The arguments have to be passed as space-separated values.

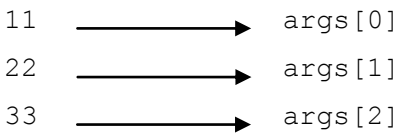
Example:

```
class Test
{
      public static void main(String args[])
      {
            System.out.println(args[0]);
            System.out.println(args[1]);
            System.out.println(args[2]);
      }
}
```

Consider the command line of above code:

D:/DCPS> `java Test 11 22 33`

This command line contains four arguments, that are `11`, `22`, `33` and these values are stored in an array ***args*** as follows:



The individual elements of an array are accessed by using an index or subscript like `args[`*i*`]`. The value of *i* denotes the position of the elements inside the array. For example, `args[2]` denotes the third element. Output of the last program segment is:

Output : `11 22 33`

Arrays

An *array* is a group of homogeneous or related data items that share a common name. A specific element in an array is accessed by its *index* or *subscript*. Arrays offer a convenient means of grouping related information.

In Java, array is an object of a dynamically generated class, which contains elements of a similar data type. Any primitive values can be store in an array in Java. Also single dimensional or multidimensional arrays can be created in Java. Array in Java is index-based, the first element of the array is stored at the **0th** index, second element is stored on **1st** index and so on.

1) One-Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional array*. Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

a) Declaration
b) Creation
c) Initialization

a) Declaration of One-Dimensional Arrays

One-dimensional arrays in Java are declared in two forms as follows:

Form 1

```
type  arrayname[];
```

Form 2

```
type[]  arrayname;
```

An array declaration has two components: the ***type*** and the ***name***. The ***type*** declares the element type of the array. The element type determines the data type of each element that comprises the array. The ***arrayname*** determines the name of array and it is any valid identifier. Also note that the size of the array is not given in the declaration. For example:

```
int counter[];
float average[];

int[] counter;
float[] average;
```

Note that, above declarations declares the `counter` and `average` arrays, but no array actually exists. In fact, the value of these arrays is set to null, which represents an array with no value. To link these arrays with an actual, physical array of integer and float, user must create these arrays.

b) Creation of One-Dimensional Arrays

After the array is declared, it must be created in memory. Creation of array gives required memory locations to the array. Arrays are created using `new` operator, as below:

```
arrayname = new type[size];
```

Here, ***type*** specifies the data type of array, ***size*** specifies the number of elements in the array, and ***arrayname*** is the name of the array that is declared in the declaration. The elements in the array allocated by `new` will automatically be initialized to zero. Example:

```
counter = new int[5];
average = new float[10];
```

These lines create necessary memory locations for the arrays `counter` and `average` and designate them as `int` and `float` respectively. Now, the variable `counter` refers to an array of 5 integers and `average` refers to an array of 10 floating point values.

It is also possible to combine the two steps declaration and creation into single step:

```
int counter[]    =  new int[5];
float average[]  =  new float[10];
```

c) Initialization of One-Dimensional Arrays

The final step is to put values into the array created. This process is known as *initialization*. This is done using the array subscripts as shown below:

```
arrayname[subscript] = value;
```

Examples:

```
counter[0] = 11;
counter[1] = 22;
counter[4] = 55;
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, us shown below:

```
type arrayname[] = {list of values};
```

The array initializer is a list of values separated by commas and surrounded by curly braces. Note that no size is given. The compiler allocates enough space for all the elements specified in the list. For example:

```
int counter[] = {35, 40, 20, 57, 19};
```

It is possible to assign an array object to another. For example:

```
int a[] = {1, 2, 3};
int b[];
b = a;
```

are valid in Java. Both the arrays will have the same values.

- **Array Length**

In Java, all arrays store the allocated size in a variable named `length`. This information will be useful in the manipulation of arrays when their sizes are not known. The length of the array `Test` can be obtain using `Test.length`. For example:

```
int TestSize = Test.length;
```

2) Two-Dimensional Arrays

Two-dimensional array in Java is the simplest form of multi-dimensional array. The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns and we can access the record using both the row index (subscript) and column index (subscript).

a) Declaration of Two-Dimensional Arrays

Two-dimensional arrays in Java are declared as follows:

```
type arrayname[][];
```

Here, *type* declares the data type of the array. The *arrayname* is any valid identifier.

Example:

```
int myArray[][];
```

Above declarations declares the two-dimensional array **myArray**.

b) Creation of Two-Dimensional Arrays

Two-dimensional array is created as shown below:

```
arrayname = new type[row-size][column-size];
```

Here, *type* specifies the data type of array, *row-size* and *column-size* specifies the number of elements in the array, and *arrayname* is the name of the array that is declared in the declaration. For example:

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

or

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

This creates a two-dimensional array that can store 6 integer values, two rows and three columns.

c) Initialization of Two-Dimensional Arrays

The final step is to put values into the array created. This process is known as initialization. This is done as shown below:

```
type arrayname[][] = {list of values};
```

The initialization is done row by row. For example:

```
int myArray[][] = {1, 2, 3, 4, 5, 6};
```

This initializes first row as 1, 2, 3 and second row as 4, 5, 6. The above initialization statement can be written as:

```
int myArray[][] = {{1, 2, 3}, {4, 5, 6}};
```

by surrounding the elements of each row by braces.

We can also initialize two-dimensional array in the form of matrix:

```
int myArray[][] = {
                    {1, 2, 3},
                    {4, 5, 6}
                  };
```

■ ■ ■ ■ ■

Sample Questions

1. Explain features of Java.
2. Describe structure of Java program with example.
3. Explain with example:
   (i) Keyword (ii) Variable (iii) Literals
4. State and explain selection statements in Java.
5. Describe command line arguments with suitable example.
6. What are command line arguments? How are they useful?
7. What is one dimensional array? How to declare and initialize array? Explain with syntax.
8. What is an array? How is one dimensional declared, created and initialized in Java?

■ ■ ■ ■ ■

References:

1) The Complete Reference JAVA2 by Herbert Schildt (Tata McGraw)
2) The Complete Reference JAVA by Patrik Noughton
3) Programming with JAVA - A Primer: By E.Balguruswamy (Tata McGraw)
4) Programming in JAVA : By S. S. Khandare (S. Chand)
5) Teach Yourself 'Java' in 2 Hrs : By Sams
6) Java for You : By P. Koparkar
7) Internet

■ ■ ■ ■ ■