1.1 INTRODUCTION TO JAVA

1.1.1 Overview Of Java

Java is a OOPS based programming language. Java was designed to be easy for the professional programmer to learn and use effectively. If you already understand the basic concepts of OOPS, learning Java will be even easier. The ability to create robust programs was given high priority in the design of Java. In a well written Java program, all run-time errors can and should be managed by your program. Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine.

1.1.2 Brief history of Java

Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridant Sun Microsystems in 1991.It took 18 months to develop first working version. This language was initially called Oak but was renamed as Java in 1995.The primary motivation for the development of Java was the need for a platform independent language that can be used to create software to be embedded in various electronic devices (eg: remote controllers).Many different types of CPUs are used as controllers. The trouble with C and C++ is that they are System dependent.. An executable file created in C and C++ on a particular processor and operating system can be executed only on that processor and OS. If either the processor or OS is changed it is not possible to execute code. So efforts were started to create a platform independent, portable language that could be used to produce programs that would run on a variety of CPUs under different environments. This effort ultimately led to the creation of Java. Another factor that made java inevitable was the emergence of World Wide Web. Internet is a global network of computers. So there will be different types of computers with different processors and different operating systems existing on internet. Even though many platforms are attached to internet, user should be able to run a program on all of them. So if software is developed using C/C++ languages, that software can be distributed on internet and people can download the .exe files. But they cannot run those files because of system dependency. Again the need for a platform independent language was necessary which will allow to run a program on any system was needed.

Computer languages evolve for two reasons. To adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted java was the need for platform independent programs destined for distribution on internet .Java enhances and refines the object oriented paradigm used by C++.Many of the features of java are inherited from its predecessors like C and C++. From C java derives its syntax. Many of java's object oriented features were influenced by C++.When Java is compiled, it is not compiled into platform *specific machine*, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

1.1.3 What is the Java Virtual Machine? What is its role?

Java was designed with a concept of 'write once and run everywhere'. Java Virtual Machine plays the central role in this concept. The JVM is the environment in which Java programs execute. It is software that is implemented on top of real hardware and operating system. When the source code (.java files) is compiled, it is translated into byte codes and then placed into (.class) files. The JVM executes these byte codes. So Java byte codes can be thought of as the machine language of the JVM. A JVM can either interpret the bytecode one instruction at a time or the bytecode can be compiled further for the real microprocessor using what is called a just-in-time compiler. The JVM must be implemented on a particular platform before compiled programs can run on that platform

1.1.4 Why is Java Important to the internet?

The introduction of internet pulled Java to the forefront of programming and java has a profound effect on internet. It simplified the web programming. Java also innovated a type of networked program called applet.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to java, cyberspace was effectively closed to half the entities that now live there, as you will see, java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet.

1.1.5 Java Applets and Applications

Java can be used to create two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. That is an application created by java is more or less like one created using C or C++. When used to create applications, java is a not much different from any other computer language. Rather, it is java's ability to create applets that makes it important.

Applets are small programs developed for internet applications. An applet located on a distance computer(server) can be downloaded via internet and executed on a local computer(client) using java compatible web browser. We can develop applets for doing everything from simple animated graphics to complex games and utilities. An applet is downloaded on demand without further interaction with user .If the user clicks on a link that contains applet, the applet will be automatically downloaded and run in browser. An applet is actually a tiny java program, dynamically downloaded across the network, just like an image, sound file, or video clip.

Creation of Applets changed internet programming because it expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data

until you execute it. By contrast applet is a dynamic, self-executing program. Such a program is an active agent on client computer and yet the server initiates it.

Security

As you are likely aware, every time that you download a "normal" program, you are risking a viral infection. Prior to java, most users did not download executable programs frequently, and those who did, scan them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system, java answer both of these concerns by providing a "firewall" between a networked application and your computer.

When we use a java compatible Web browser, you can safely download Java applets without fear of viral infection. Java achieves this protection by confining a java program to the java execution environment and not allowing it to access other parts of the computer. The ability to download applets without the fear of any security breach is considered as the most important aspect of java.

Portability

Many types of computers and operating systems are in use throughout the world - and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. Java was able to achieve this with the help of byte codes and Java virtual machine.

1.1.6 Java's Magic: The Byte code

The key that allows java to solve both the security and the portability problems just described is that the output of a java compiler is not executable code. Rather, it is byte code. Byte code is a highly optimized set of instructions designed to be executed by the java runtime system, which is called the java virtual machine (JVM). That is, in its standard form, the JVM is an interpreter for byte code, this may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted mostly because of performance concerns. However, the fact that a java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why.

Translating a java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform, all interpret the same java bytecode. If a java program were compiled to native code, the different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.

The fact that a java program is interpreted also helps to make it secure. Because the execution of every java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the java language.

When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with java, the difference between the two is not so great. The use of bytecode enables the java run-time system to execute programs much faster than you might expect.

Although java was designed for interpretation, there is technically nothing about java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire java program into executable code all at once, because java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code, as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation)is still in charge of execution environment.

1.1.7 Java Buzzwords

No discussion of the genesis of java is complete without a look at the java buzzwords. Although the fundamental forces that necessitated the invention of java are portability and security, other factors also played an important role in molding the final form of the language, the key considerations were summed up by the java team in the following list of buzzwords:

Simple
Secure
Portable
Object-oriented
Robust
Multithreaded
Architecture-neutral
Interpreted
High performance
Distributed
Dynamic

Simple

Java is a simple programming Language.Java was designed to be easy for the professional programmer to learn and use effectively. Java inherits the C/C++ syntax and many of the object-oriented features of C++.So most programmers have little trouble learning java.

Also, some of the more confusing concepts from C++ (like pointers, multiple inheritance) are either left out of java or implemented in a cleaner, more approachable manner. Automatic garbage collection has been added, thereby simplifying the task of Java programming but making the system somewhat more complicated. A common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection (periodic freeing of memory not being referenced) the Java language not only makes the programming task easier, it also dramatically cuts down on bugs. Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The Java interpreter and standard libraries have a small footprint. A small size is important for use in embedded systems and so Java can be easily downloaded over the net.

Object-Oriented

Java is a true object oriented language. The object-oriented facilities of Java are essentially those of C++.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 that we can use in our program by inheritance. The object model in java is simple and easy to extend.

Robust

The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of java. To gain reliability, java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, java frees you from having to worry about many of the most common causes of programming errors. Because java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is key feature of java.

Architecture Neutral

Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. To enable a Java application to execute anywhere on the network, the compiler generates an architecture-neutral object file format--the compiled code is executable on many processors, given the presence of the Java runtime system.

The Java compiler does this by generating bytecode instructions which have nothing to do with particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly. Their goal while developing java was "write once; run anywhere, anytime, forever." To a great extend this goal is accomplished.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. This means that we need not wait for the application to finish one task before beginning another. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. "High-performance cross-platform" is no longer an oxymoron.

Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra address- space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/ server programming.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

1.1.8 Lexical Issues

Now we will see atomic elements of java. Java programs are a collection of white space identifiers, comments, literals, operators, separators, and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In java, whitespace is a space, tab, or new line.

Identifiers

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, java is case-sensitive, so VALUE is a different identifier the Value. Some examples of valid identifiers are:

AvgTemp

count a4 $test this_is_ok

Invalid variable names include:

2count high-temp
Not/ok

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

The Java language has three styles of comments:

// text

All characters from // to the end of the line are ignored.

/ text */*

All characters from /* to */ are ignored.

*/** text */*

These comments are treated specially when they occur immediately before any declaration. They should not be used any other place in the code. These comments indicate that the enclosed text should be included in automatically generated documentation as a description of the declared item.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from sub packages and classes. Also used to separate a variable or method from a reference variable.

Java Keywords

There are 49 reserved keywords currently defined in the Java language These keywords, combined with the syntax of the operators and separators, form the definition of the Java language. These keywords cannot be used as names for a variable, class, or method.

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

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java only defines the keywords shown in Table 2-1. The **assert** keyword was added by Java 2, version 1.4

In addition to the keywords, Java reserves the following: **true, false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

1.2 DATA TYPES VARIABLES AND ARRAYS

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

1.2.1 Data Types

There are two data types available in Java:

1.Primitive Data Types
2.Reference/Object Data Types

The Primitive Types

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. They are **byte, short, int, long, char, float, double**, and **boolean**. These can be put in four groups:

- **Integers**: includes **byte, short, int**, and long which are whole valued sign numbers.
- **Floating-point numbers** :includes **float** and **double** which represent numbers with fractional precision
- **Characters** : includes **char**, which includes symbols in a character set like letters and numbers.
- **Boolean**: includes **boolean** which is a special type for representing true/false values.

byte

The byte data type is an 8-bit signed integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters.

Example : ***byte a = 100 , byte b = -50***

short

The short data type is a 16-bit signed integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

Example : ***short s= 10000 , short r = -20000***

int

The int data type is a 32-bit signed integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else. This data type will most likely be large enough for the numbers your program will use.

Example : ***int a = 100000, int b = -200000***

long:

The long data type is a 64-bit signed integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.

Example : ***int a = 100000L, int b = -200000L***

float:

The float data type specifies a single precision value that uses 32 bits of storage. Variables of type float are used when you need a fractional component but doesn't require high degree of precision. It's value ranges from 1.4E-45 to 3.4028235E+38

Example : ***float f1 = 234.5f***

double

When the accuracy of the floating point number is insufficient, we can use the double to define the number. It uses 64 bits to store a value. This data type is generally used as the default data type for decimal values. The double is same as float but with longer precision and takes double space (8 bytes) than float.

Example : ***double d1 = 123.4***

boolean:

The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. Default value is false. This data type represents one bit of information.

char

In java data type used to store characters is char. character type are the alphabets which are written in single quotes. Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all the characters found in all human languages. It is a unification of dozens of character sets like latin,greek,Arabic,Hebrew and many more. This requires 16-bits.So in java char is a 16-bit type. Range of char is from 0 to 65536.Since java is designed to allow applets to be written for world wide use, using Unicode to represent characters make sense. Standard 8-bit ASCII character set is a subset of Unicode and ranges from 0-127A character variable can be assigned a value by enclosing the character in single quotes.

eg:***char myChar = `Q';***

1.2.2 Literals

Literals are fixed values that are represented in their human readable form. For example 100 is a literal Literals are also commonly called constants. In Java programming language there are some special type of literals that represent numbers, characters, strings and boolean values

Integer Literal

Integer is the most commonly used type in a program. Any whole number value is an integer literal.eg:1,4,43. These all are decimal values, that means they are describing base 10 number. There are other two bases which can be used in integer literals. They are hexadecimal and octal. An octal digit ranges from 0-7, while a hexadecimal digit ranges from 0-15A through F are substituted for 10 through 15.An octal value is represented with a leading 0.Hexadecimal digit is represented by leading 0x.

Integer literal creates an integer value. But it is possible to assign an integer literal to one of java's other integer types, such as byte or long without causing a type mismatch error. When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of target type. An integer literal can be assigned to a long variable. But to specify a long literal, you need to explicitly specifiy that it is long literal by appending L or l to the literal.

Floating Point Literals

Floating point numbers represent a decimal value with a fractional component. They can be expressed in either scientific or standard notation .Standard notation has a whole number component followed by a decimal point followed by a fractional component eg:3.5788S scientific notation uses a standard floating point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number which can be positive or negative.Eg:65760.23E-2. Floating Point Literals in java default to double precision. To specify a float literal F or f should be appended to the constant. To explicitly specify a double literal append D or d.

Boolean Literals

The values true and false are also treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value.

Eg:*boolean chosen = true;*

Remember that the literal **true** is not represented by the quotation marks around it. The Java compiler will take it as a string of characters, if its in quotation marks.

Character Literals

Characters in Java corresponds to Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators such as addition and subtraction. We can specify a character literal as a single character in a pair of single quote characters such as 'a', '#', and '3'.For characters that are impossible to enter directly there are several escape sequence ,which allows you to enter the character you need.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\ddd	Octal(d is octal constant)
\uxxxx	Hexadecimal(xxxx is a hexadecimal constant)

String Literals

String literals are specified by enclosing a sequence of characters between a pair of double quotes. Examples are "HelloWorld", "two\nlines". The escape sequence and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One point to remember is that java strings must begin and end on same line.

1.2.3 Variables

A variable is a container that holds values that are used in a Java program. A java variable needs to be declared before it is used and it can be placed anywhere in the block where they are used. A variable could be declared to use one of the eight primitive data types: byte, short, int, long, float, double, char or boolean.

A variable is any combination of letter, number, underscore,$ sign. A variable must not begin with number. Java variables are case sensitive. Therefore variables sum and Sum are different.

Declaring Variables

In java all variables must be declared before its used. The basic form of variable declaration is :

type identifier[=value][,identifier[=value]…];

Here type is one of the primitive types. Identifier is the name of the variable. You can initialize a variable by specifying an equal sign and a value. Here the initialization expression must result in a value whose type should same as the type specified by the variable. To declare more than one variables of same type, use comma separated list.

eg; intd=54,e,f=34;

Dynamic Initialization

It is possible to initialize variables dynamically by using expressions which is valid at the time the variable is declared.

eg: double h=a*b;

Here initialization expression can be any value valid at the type of initialization, including calls to methods, other variables and literals.

Scope of a variable

Scope refers to the validity of a variable across the java program. All of the variables used have been declared at the start of the main( ) method. However, Java allows variables to be declared within any block. As explained in a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

1.2.4 Type Conversion and Casting

It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the **int** type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

eg: ***int a;***

byte b;

// ...

b =(byte) a;

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components.Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

1.2.5 Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

byte a = 40;

byte b = 50;

byte c = 100;

int d = a * b / c;

The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte or short operand to int when evaluating an expression. This means that the sub expression a * b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

The Type Promotion Rules

In addition to the elevation of bytes and shorts 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, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is **double**, the result is **double**.

```
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
```

In the first sub expression, f * b, b is promoted to a **float** and the result of the sub expression is **float**. Next, in the sub expression i / c, c is promoted to **int**, and the result is of type **int**. Then, in d * s, the value of s is promoted to **double**, and the type of the sub expression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

General Structure of a Java program

Documentation Section
Package Statements
Import Statements
Interface Statements
Class Classname **{** **public static void main(String args[])** **{** **}** **}**

Let us see now how the simple java program will look like. You can use any editor like notepad or any Java IDE for writing java programs.

```
/* This is a sample java program
Save this file as Welcome.java */
import java.lang.*;
class Welcome
  {
//  A java program will start from here.
public static void main(String args[])
 {
System.out.println("  Welcome to Java-Sample!!! ");
}
   }
```

It is a common practice to write a brief explanation of the program in the beginning which is called the comment line. Single line comment start with // followed by description. Java provides multiline comments which span over adjacent lines and are delimited by the character sequence /* and */.These lines are ignored by compiler as they are meant to be documentation for programmers

The next statement is a package statement which helps in organizing groups of files in large projects into a unit. Similar to a folder on a file system containing physical grouping of many files, a package organizes logically related Java classes. Package is helpful in identifying the required classes for developing application

Next to package is import statement which makes the classes in required packages available to current program. To make use of libraries, it is essential to import required packages. The user defined packages already created can be made available to program using import statement.

The interface statement follows the import statement. Interfaces are similar to classes without defining methods in it.

After this the class can be defined. Class definition includes class header where we use the keyword **class** to define a new class. It will be followed by class name. The class definition must begin with opening curly brace ({) and ends with closing curly brace (}). Within the braces main() method is defined.

public static void main(String args[])

Defines the method main. Every java application program must include the main() method. The program will start execution by calling this main method. Java application can have any number of classes but only one of them must include a main method to initiate the

execution

This line contains a number of keywords **public**, **static** and **void**. The keyword **public** is an access specifier that declares that the main method is accessible to all other classes. The keyword **static** is a kind of modifier, which declares this method as one that belongs to the entire class and not part of any objects of the class. The **main** must be always be declared as **static** since the interpreter uses this method before any objects are created. The keyword **void** means that the method main() does not return any value. As we had seen before all the java program will start execute by calling the **main** method. If we want to pass any information to a method, it will be received by the variables declared within the parenthesis called parameters. In a main() method there is only one parameter , String args[] . args[] is a name of the parameter that is an array of the objects of data type String. String store sequences of characters and args will receive the command line arguments.

Within the main or any other methods program logic is included. All the method in java must be start with opening curly brace ({) and ends with closing curly brace (}).

Running a Java Program

Suppose if you are entering your program in notepad, then save this file as Welcome.java. In a java program class containing the main() function should be given as the file name of the source code with .java extension. After we have written our program we need to compile and run the program. For that we need to use the compiler called javac which is provided by java. Go to the command prompt and type the file name as shown here.

c;\>javac Welcome.java

The javac compiler will create a class file called Welcome.class that contains only byte codes. These byte codes have to be interpreted by a Java Virtual Machine(JVM) that will convert the bytecodes into machine codes. Once we successfully compiled the program, we need to run the program in order to get the output. So this can be done by the java interpreter called java. In the command line type as shown here.

c:\>java Welcome

So the output will be displayed as

Welcome to Java-Sample!!!

As we had seen above, when the source code has been compiled, it creates a class file with an extension of .class. Since this class file contains the byte codes that can be interpreted by the JVM which can be resided at any platform. Remember that while running the program we are using only .class file not the .java file. So once you got the class file you can run the same java program at any platform instead of writing the program again and again. This is the very special feature about java that 'Write once and Run anywhere'

1.2.6 Arrays

An array is an ordered sequence of finite data items of same data type that are referred to by a common name. The common name is array name and individual data item is known as an element of array. Elements of array are stored in subsequent memory locations starting from the first memory location of the block of memory created for array. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information. Unlike in other languages, arrays in Java are actual objects that can be passed around and treated just like other objects.

One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables.

Declaring Array Variables

The first step in creating an array is creating a variable that will hold the array, just as you would any other variable. Array variables indicate the type of object the array will hold . General form: **type var-name[ ];**

type declares the base type of the array. All the elements in the array will be of data type **type. var-name** specifies name of the array.

The following are all typical array variable declarations:

String **difficultWords[ ];**
int **temps[ ];**
float **val[ ];**

An alternate method of defining an array variable is to put the brackets after the type instead of after the variable.

String **[ ] difficultWords;**
int **[ ] temps;**
float **[ ] val;**

Creating array

There are different ways to create a single dimensional array.

- We can declare one-dimensional array and directly store elements at the time of declaration by enclose the elements of the array inside braces, separated by commas.

  eg: int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };

Here each of the elements inside the braces must be of the same type as the variable that holds that array. An array whose size is equal to the number of elements you've included will be automatically created for you.

- Another way of creating one-dimensional array is by declaring the array first and then allocating memory by using new operator.

General form:

type var-name[ ] ;
var-name = new type[size];

Eg: ***int marks[];***
marks=new int marks[5];

These two statements can also be written by combining them into a single statement as

type var-name[] = new type[size];

Eg*: int marks[]=new int[5];*

This line creates a new array of integer with 5 elements and allocates memory for storing 5 integer elements into the array. When you create a new array object using new, you must indicate how many slots that array will hold.

When you create an array object using new, all its slots are initialized for you (0 for numeric arrays, false for boolean, '\0' for character arrays, and null for objects). You can then assign actual values or objects to the slots in that array.

To store elements in array we can use statements like these in program.

```
marks[0]=50;
marks[1]=60;
marks[2]=50;
marks[3]=60;
marks[0]=50;
```

Or, we can pass values from keyboard to the array by using a loop

```
for(int i=0;i<5;i++)
 marks[i]=Integer.parseInt(br.readLine());
```

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range.

Accessing Array Elements

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

int twoD[][] = new int[4][5];will allocate a 4 by 5 array and assigns it to twoD

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
Eg:int twoD[][] = new int[4][];
   twoD[0] = new int[5];
   twoD[1] = new int[5];
   twoD[2] = new int[5];
   twoD[3] = new int[5];
```

When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.

```
Eg: int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];
```

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.

```
Eg:double m[][] = {
                    { 0,1, 2, 3 },
                    { 4, 5, 6, 7 },
                  };
```

1.3 OPERATORS

Java provides a rich operator environment. Most of its operators can be divided into the following four groups:

arithmetic
bitwise
relational
logical

Java also defines some additional operators that handle certain special situations.

1.3.1 Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)

*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The basic arithmetic operations—addition, subtraction, multiplication, and division act on all numeric types. The minus operator also has a unary form which negates its single operand. When the division operator is applied to an integer type, there will be no fractional component attached to the result.

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the % can only be applied to integer types.).

eg: ***double y = 42.25;***

a=y%10;

Here value of ' a' will be 2.25

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

Eg :***a = a + 4;***

In Java, you can rewrite this statement as shown here: a += 4;

Eg: ***a = a % 2; which can be expressed as a %= 2;.In this case, the %= obtains the remainder of a/2 and puts that result back into a.***

The assignment operators provide two benefits. First, they save you a bit of typing, because they are "shorthand" for their equivalent long forms. Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand. In the

prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.

1.3.2 The Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand.

Eg: ~00101010 gives 11010101

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
  00101010      42
& 00001111      15
  ----------
  00001010      10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
 00101010      42
| 00001111     15
 --------------
 00101111      47
```

The Bitwise XOR

The XOR operator ^ combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
  00101010      42
^ 00001111      15
  -------------
  00100101      37
```

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << num

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num

The Right Shift (arithmetic shift)

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

value >> num

Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.

```
Eg:   00100011      35
       >> 2
      00001000       8
```

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

```
Eg:     11111000      –8
        >>1
        11111100      –4
```

Unsigned Right Shift (logical shift)

The >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric

value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit. It is same as the bitwise right shift (>>) operator except that it does not preserve the sign of the original expression because the bits on the left are always filled with 0.

Here, a is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111      –1 in binary as an int
 >>>24
00000000 00000000 00000000 11111111      255 in binary as an int
```

The >>> operator is often not as useful as you might like, since it is only meaningful for 32- and 64-bit values.

Bitwise Operator Assignments

All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in 'a' right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

1.3.3 Relational Operators

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements. Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

Point to be noted

In C/C++, these types of statements are very common:

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ... // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0)) ... // This is Java-style.
if(done != 0) ...
```

The reason is that Java does not define true and false in the same way as C/C++.In C/C++, true is any nonzero value and false is zero. In Java, true and false are nonnumeric values which do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

1.3.4 Logical Operators

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: !true == false and !false == true

The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the left one being true or false in order to function properly.

Eg: ***if (denom != 0 && num / denom > 10)***

Since the short-circuit form of **AND (&&)** is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single **&** version of **AND**, both sides would have to be evaluated, causing a run-time exception when denom is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule.

Eg**:** ***if(c==1 & e++ < 100) d = 100;***

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

1.3.5 Assignment Operator

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = expression;

Here, the type of var must be compatible with the type of expression it allows you to create a chain of assignments.

Eg: ***int x, y, z;***

x = y = z = 100;

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x.

1.3.6 The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?, and it works in Java much like it does in C, C++. The ? has this general form:

expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.

Eg: ratio = denom == 0 ? 0 : num / denom;

1.3.7 Operator Precedence

Following table shows the order of precedence for Java operators, from highest to lowest. Notice that the first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Parentheses are used to alter the precedence of an operation

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
> >=	< <=		
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

1.4 CONTROL STATEMENTS

Control statements cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **selection, iteration,** and **jump**. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.

1.4.1 Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

1.4.1.1 if statement

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition) statement1;
else statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The **else** clause is optional. Here if the condition is true, then statement1 is executed. Otherwise,statement2 (if it exists) is executed. In no case will both statements be executed.

For example, consider the following:

```
eg:   if(a < b) a = 0;
      else b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

1.4.1.2 Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
eg:   if(i == 10) {
        if(j < 20) a = b;
        if(k > 100) c = d; // this if is
        else a = c; // associated with this else
      }
      else a = d;
```

the final **else** is not associated with if(j<20), because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100), because it is the closest if within the same block.

1.4.1.3 else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. General Format is:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
```

```
...
else
statement;
```

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

Eg:
```
if (month == 1 )
    value = "A";
else if (month == 2)
    value = "B";
else if (month == 3)
    value = "C";
else if (month == 4)
    value = "D";
else
    value = "Error";
```

1.4.1.4 switch statement

The switch statement in Java provides a convenient method for branching a program based on a number of conditions. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

The expression must be of type **byte**, **short**, **int**, or **char**. Each of the values specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed. In **switch** the value of the expression is compared with each of the

literal values in the **case** statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no case matches and no **default** is present, then no further action is taken. The **break** statement is used inside the switch to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**.

Eg:
```
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
```

1.4.1.5 Nested switch Statements

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

Eg:
```
switch(count) {
 case 1:
       switch(target)
      {
        case 0:
          System.out.println("target is zero");
          break;
        case 1: // no conflicts with outer switch
          System.out.println("target is one");
          break;
      }
 break;
 case 2: //
```

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is only compared with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

There are three important features of the switch statement to note:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs.

1.4.2 Iteration Statements

Java's iteration statements are **for**, **while**, and do-**while**. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. Java has a loop to fit any programming need.

1.4.2.1 while statement

The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition)
 {
        // body of loop
 }
```

The condition can be any boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

eg:
```
int n = 10;
while(n > 0)
  {
        System.out.println("tick " + n);
        n--;
}
```

Here the value of n is checked and if it is true the statements inside the loop is executed. So the loop is executed 10 times.

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

Eg: ***int a = 10, b = 20;***

```
while(a > b)
System.out.println("This will not be displayed");
```

Here the call to println( ) is never executed.

1.4.2.2 do-while statement

If the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a while loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.

Eg:

```
do
{
    System.out.println("tick " + n);
    n--;
}
while(n > 0);
```

1.4.2.3 for statement

The general form of the for statement is:

```
for(initialization; condition; iteration)
{
    // body
}
```

If only one statement is being repeated, there is no need for the curly braces. In for loop, when the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. The initialization expression is executed only once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an

expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Eg:
```
int n;
for(n=10; n>0; n--)
    System.out.println("tick " + n);
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a for loop is only needed for the purposes of the loop and is not used elsewhere. In that case, it is possible to declare the variable inside the initialization portion of the **for** statement.

When you declare a variable inside a **for** loop the scope of that variable is limited to the **for** loop Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you need to declare it outside the **for** loop.

Some for Loop Variations

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the for loop

  Eg: for(a=1, b=4; a<b; a++, b--)

  In this example, the initialization portion sets the values of both a and b. The two comma-separated statements in the iteration portion are executed each time the loop repeats.

- The condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

  ```
  Eg: boolean done = false;
  for(int i=1; !done; i++)
  ```

  In this example, the for loop continues to run until the boolean variable done is set to true. It does not test the value of i.

- Either the initialization or the iteration expression or both may be absent.

  Eg: for(; !done;)

  In the above example the initialization and iteration expressions have been moved out of the **for**.

- You can intentionally create an infinite loop(a loop that never terminates) if you leave all three parts of the for empty.

  ```
  Eg: for( ; ; )
      {
          // ...
      }
  ```

This loop will run forever, because there is no condition under which it will terminate.

1.4.2.4 Nested Loops

Like all other programming languages, Java allows loops to be nested. The placing of one loop inside the body of another loop is called nesting. When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop.

Eg:
```
for (int i=1; i<=9; i++)
{
        System.out.println();
        for (int j=1; j<=i; j++)
        {
                System.out.print(j);
        }
}
```

1.4.3 Jump Statements

Java supports three jump statements: **break, continue**, and **return.** These statements transfer control to another part of your program.

1.4.3.1 break statement

In Java, the break statement has three uses. First, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of **goto**.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
Eg: for(int i=0; i<100; i++)
    {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
```

Here although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when i equals 10.

When used inside a set of nested loops, the break statement will only break out of their innermost loop.

Using break as a Form of Goto

Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner. This usually makes code hard to understand and maintain.. But there are a few places where the **goto** is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By

using this form of **break**, you can break out of one or more blocks of code. These blocks need not be part of a **loop** or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of break works with a **label**. As you will see, break gives you the benefits of a goto without its problems. The general form of the **labeled break** statement is shown here:

break label;

Here, **label** is the name of a label that identifies a block of code. When this form of **break** executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled break statement to exit from a set of nested blocks. But you cannot use break to transfer control to a block of code that does not enclose the break statement.

To name a block, put a **label** at the start of it. A label is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the end of the labeled block.

One of the most common uses for a **labeled break** statement is to exit from nested loops.

Eg:
```
outer: for(int i=0; i<3; i++)
{
    System.out.print("Pass " + i + ": ");
    for(int j=0; j<100; j++)
    {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
    }
    System.out.println("This will not print");

}
System.out.println("Loops complete.");
```

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete

when the inner loop breaks to the outer loop, both loops have been terminated.

1.4.3.2 continue statement

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Eg:
```
for(int i=0; i<10; i++)
{
    System.out.print(i + " ");
    if (i%2 == 0) continue;
```

```
        System.out.println("");
    }
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

1.4.3.3 return statement

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.