Unit I
Chapter 1 Java Fundamentals

The Origins of Java

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- It was initially named as "Oak" but renamed as "Java" in 1995.
- It is platform-independent language that could be used to create software to be embedded in consumer electronic devices such as toasters, microwave ovens and remote controls.
- It is portable and cross-platform language that produces code which can run on a variety of CPUs under different environments.
- It is portable and hence well suited for World Wide Web.

How Java Relates to C and C++

- Java is directly related to both C and C++. It inherits syntax from C and C++. This makes it easy and familiar to programmers.
- Its object model is adapted from C++.
- It is influenced by C++ but not the extended version of C++.

How Java Relates to C#

- Many features of C# are common to Java
- Both Java and C# share general syntax of C++
- Both support distributed programming and utilize the same object model.
- Java and C# are optimized for two different types of computing environments.

Java's Contribution to the Internet

Java had a profound effect on the Internet. Java innovated new type of network program called applet. Java addressed the portability and security issues associated with the Internet.

- **Java Applets**: An Applet is a special kind of program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible browser. If the user clicks a link that contains an Applet, the applet will be automatically downloaded and run in the browser. Applet is a small program that is used to display data provided by the server, to handle user inputs or to provide simple functions such as EMI calculator that execute locally. There are two categories of objects that are transmitted between the server and the client: passive information and active or dynamic. Reading an e-mail is example for passive data. Applet is a dynamic self-executing program, yet it is initiated by the server.
- **Security**: A normal program that is downloaded may contain Viruses, Trojan horses or other harmful code which can gain an unauthorized access to the recourses. To download and execute applets safely some protection mechanism is needed. This

protection is achieved by confining an applet to the Java execution environment and not allowing it access to the other parts of the computer.

- **Portability**: Portability is a major aspect of the Internet because there are different types of computers and operating systems connected to it. A Java program need to be executed on any computer connected to the Internet. It is not practical to have different versions of applet for different computers. The same code must work in all computers.

Java's Magic: The Bytecode

The key that allows Java to solve both security and portability problems is the output of a Java compiler is not executable code. It is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called Java Virtual Machine (JVM). JVM is an interpreter for bytecode. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only JVM needs to be implemented for each platform. Details of the JVM differ from platform to platform but all understand the same bytecode.

The fact that a Java program is executed by the JVM makes it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside the system. The bytecode is highly optimized and the use of bytecode enables the JVM to execute programs much faster.

The Java Buzzwords (Features of Java)

Simple	Java has a concise set of features that makes it easy to learn and use
Secure	Java provides a secure means of creating Internet applications
Portable	Java contains the object-oriented programming philosophy
Object-Oriented	Java encourages error-free programming by being strictly
Robust	Java provides strict compile time and run time checking of data types, exception handling and garbage collection mechanism
Multithreaded	Java can handle multiple tasks simultaneously by using multithreaded programs
Architecture-neutral	Java is not tied to a specific machine or operating system
Interpreted	Java supports cross-platform code through the use of Java bytecode
High Performance	Java bytecode is highly optimized for speed of execution
Distributed	Java is designed with the distributed environment of the Internet in mind
Dynamic	Java is capable of dynamically linking in the new class libraries, methods and objects.

Object Oriented Programming

The centre of Java is OOP. All OOP languages including Java have three features in common: encapsulation, polymorphism and inheritance.

1. **Encapsulation**: This mechanism binds together the code and the data it manipulates and keeps both safe from outside interference and misuse. Java's basic unit of encapsulation us the class. It specifies both the data and the code that will operate on that data. Objects are instances of a class. The code and data that constitute a class are called members. Data is referred to as member variables and the code is referred to as methods.
2. **Polymorphism**: It allows one interface to a general class of action. This mechanism is referenced by the phrase "one interface, multiple methods". It is possible to define a generic interface to a group of related activities. It is the job of the compiler to select the specific action based on the situation.
3. **Inheritance**: It is the process by which one object can acquire the properties of another object. It supports the concept of hierarchical classification. Without the use of hierarchies, each object needs explicit definition of all its characteristics. Using inheritance, an object need only define those qualities unique within its class. It can inherit its general attributes from its parent.

A Simple Java Program

```
class Example{
   public static void main(String args[]) {
     System.out.println("This is my First Java Program");
  }}
```

For most computer languages, the name of the file that holds the source code is arbitrary. However, this is not the case with Java. The Java compiler requires that a source file use the **.java** extension. So the name of the above program must be Example. In Java, all code must reside inside a class. The name if the main class should match the name of the file that holds the program.

To compile the program, execute the compiler, **javac**, specifying the name of the source file on the command line as

```
javac Example.java
```

The javac compiler creates a file called Example. Class that contains the bytecode version of the program. Bytecode is not executable code. Bytecode must be executed by a Java Virtual Machine. To run the program, Java interpreter should be used as follows: `java Example`

`public static void main(String args[])`

All Java applications begin execution by calling **main()** method. The **public** keyword is an access modifier which determines how the program can access the members of the class. If the keyword is public, then it can be accessed by code outside the class in which it is declared. In this case, main() must be declared as public, since it must be called by code

outside its class when the program is started. The keyword **static** allows main() to be called before an object of the class has been created. This is necessary because main() is called by the JVM before any objects are made. The keyword void tells the compiler main() does not return any value.

Any information that is needed to be passed to a method is received by variables specified within the set of parenthesis that follow the name of the method. In main() there is only one parameter **args[]** of type array of **String**. In this case args[] receives any command-line arguments present when the program is executed.

```
System.out.println("This is my First Java Program");
```

This line outputs the string "This is my First Java Program" on the screen. println() is the method used to display output, **System** is a predefined class that provides access to the system and **out** is the output stream connected to the console. Thus **System.out** is an object that encapsulates console output.

Java Keywords

Fifty keywords are currently defined in Java. Keywords cannot be used as names for variable, class or method.

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				

In addition to the keywords, Java reserves the following: **true**, **false** and **null**.

Identifiers in Java

- An identifier is a name given to a method, a variable or any other user-defined item.
- Identifier can be from one to several character long.
- It may start with an alphabet, an underscore or a dollar sign
- It cannot contain space or any other special character
- Keywords cannot be used as identifiers. Names of any standard methods like println cannot be used as identifier

The Java Class Libraries

Java class library contains set of pre-defined classes that contain many built-in methods. These methods provide support for I/O, string handling, networking and graphics.

Chapter-2 Data Types and Operators

Java's Primitive Data Types

Type	Meaning
boolean	Represents true/false values
byte	8-bit integer
char	Character
double	Double-precision floating point
float	Single-precision floating point
int	Integer
long	Long integer
short	Short integer

Java strictly specifies a range and behaviour for each primitive type, which all implementations o the JVM must support. This is because Java programs must be portable. For example, an int is same in all execution environments. This allows programs to be fully portable.

Integers: Java defines four integer types: byte, short, int and long. The range allows both positive and negative values. Java does not support unsigned integers.

Type	Width	Range
byte	8 Bits	-128 to 127
short	16 Bits	-32,768 to 32,767
int	32 Bits	-2,147,483,648 to 2,147,483,647
long	64 Bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Floating Point Types: Floating-point types can represent numbers that have fractional components. There are two kinds of floating-point type float and double which single- and double-precision numbers respectively are. Type float is 32-bit wide and double is 64-bit wide. Of the two, double is most commonly used because all math functions in Java's class library use double values.

Characters: Java uses Unicode which defines a character set that can represent all the characters found in all human languages. In Java, char is an unsigned 16-bit type having a range of 0 to 65,536. A character variable can be assigned value by enclosing the character in single quotes. For example, to assign value X to variable ch we use the following Java statement: `char ch='X';`

The Boolean Type: The Boolean type represents true/false values. Example: `boolean b= true;`

Literals

- Literals refer to fixed values. They are also called as constants. Integer literals are specified a numbers without fractional part. For example -100, 999 are valid integer literals. By default integer literals are of type int. If long integers are to be used, an L or l is to be appended. Example: 12L. 12 is of type int and 12L is of type long.

- By default, floating point numbers are of type double. To specify a float literal, F or f is to be appended to the constant. Example: 10.3 is of type double and 10.3F is of type float.
- A hexadecimal literal must begin with 0x or 0X where as an octal literal always begins with 0. Example hex=0X45C; oct=011;
- Java also supports sting literals. A string is a set of characters enclosed within double quotes. For example "Good Morning"

Escape sequences: Java supports some special backslash character constants that are used in output methods. A list of such backslash character constants are given below. Note that each of them represents one character, although they consist of two characters. These characters combinations are known as escape sequences.

'\b'	backspace	'\t'	horizontal tab
'\f'	form feed	'\"	Single quote
'\n'	new line	'\"'	double quote
'\r'	carriage return	'\\'	backslash

Variables

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things.

- It tells the compiler what the variable name is
- It specifies what type of data the variable will hold
- The place of declaration decides the scope of the variable
- A variable must be declared before it is used in the program. This is necessary because the compiler must know what type of data it contains.
  Ex: **int** count; **float** height; **boolean** flag **short** sno=123;

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

Example:

```
// Demonstration of dynamic initialization
class DynInit
{ public static void main(String args)
  { double  radius=4, height=5;
     double volume = 3.1416 * radius * radius * height; // volume is
initialized dynamically
     system.out.println("Volume = " +volume); }  }
```

In the above program, there are three local variables namely height, radius and volume. The first two are initialized by a constant and the volume is initialized dynamically.

The Scope and Lifetime of Variables

Java allows variables to be decaled within any block. Scope determines the visibility of objects to the other parts of the program. It also determines the lifetime of those objects. In Java variables can be declared within any block. Variables declared inside the block are not visible to the code which is defined outside the block.

Example:

```
// Demonstration of block scope
```

```
Class ScopeDemo{
public static void main (String args[]){
int x=10; // known to all code within main
if (x==10){ // Start a new scope
int y=20; // Known to only this block
System.out.println("X  =" +x+ "Y ="+y); // Both x and y are visible
here4
}
Y=100; // Error!! Y is not known here
System.out.println("X  =" +x); // x is still known here } }
```

Operators

An operator is a symbol that tells the compiler to perform a specific mathematical or logical manipulation. Java has four general classes of operators: arithmetic, bitwise, relational and logical. Java also defines some additional operators that handle certain special situations.

Arithmetic Operators:

Operator	Meaning	Operator	Meaning
+	Addition (also unary plus)	%	Modulus
-	Subtraction(also unary minus)	++	Increment
*	Multiplication	--	Decrement
/	Division		

These can be applied to any built-in numeric data types. They can also be used on objects of type char. When / is applied to integers, the result will be an integer. So 10/3 will be 3. The increment operator ++ adds 1 to its operand and the decrement operator – subtracts 1. Both increment and decrement operators either prefix or follow the operand. There is no difference whether the increment is applied as prefix or postfix. However, when they are a part of an expression, there is a difference.

```
x=10;                                    x=10;
y=++x; // In this case y becomes 11      y=x++; // y= 10 but x= 11
```

Relational and Logical Operators

Relational Operators		Logical Operators	
Operator	**Meaning**	**Operator**	**Meaning**
==	Equal to	&	AND
!-	Not equal to	|	OR
>	Greater than	^	XOR (Exclusive OR)
<	Less than	||	Short circuit OR
>=	Greater than or equal to	&&	Short circuit AND
<=	Less than or equal to	!	NOT

Relational operators can be applied to all numeric as well as char data types. For logical operations, operands must be of type Boolean and the result of logical operation is of type Boolean.

Logical operators work according to the following truth table:

P	Q	P & Q	P | Q	P ^ Q	!P
False	False	False	False	False	True
False	True	False	True	True	True
True	False	False	True	True	False
True	True	True	True	False	False

Short circuit operators work the same way as their normal counterparts. Only difference is that the normal operands will always evaluate each operand but short circuit versions will evaluate the second operand only when necessary.

The Assignment Operator

The assignment operator is =. The general form is `var=expression`; the type of var must be compatible with the type of expression.

Example:
```
int x, y,z;
        x=y=z=100; // will set 100 to  variables x,y and z
```

Shorthand Assignments

An expression of the kind `x=x+10`;can be written as `x+=10;` The operator pair += tells the compiler to assign x+10 to the variable x. Java has the following arithmetic and logical shorthand operators.

+=	-=	*=	/=
%=	&=	|=	^=

Type Conversion in Assignments

When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side. Not all types are compatible in Java. For example, int and boolean are not compatible.

When one type of data is assigned to another type of variable, an automatic type conversion will take place if

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

For example, int type is always large enough to hold all valid byte values, and both int and byte are integer types, so an automatic conversion byte to int will take place.

Example:

```
//To demonstrate type conversion
class TypeDemo{
public static void main(String args[]) {
{ long L, double D;
L=123456L;
D=L; // Automatic conversion from long to double;
D=123456.0;
L=D; //Illegal. No automatic type conversion from double to long  }
}
```

Casting Incompatible Types

A cast is an instruction to the compiler to convert one type into another. It has the following form:

(target-type) expression;

Here, target-type specifies the desired type to convert the specified expression to.

Example:

```
double x,y;
// --------
(int)(x/y);
```

Here, even though x and y are of type double, the caste converts the outcome of the expression to int. The parentheses surrounding x/y are necessary. Otherwise, the cast to int would apply only to x. The cast is necessary here because there is no automatic conversion from double to int.

When a cast involves a narrowing conversion, information might be lost. For example, when casting a long into short, information will be lost if the value held in long type of variable is greater than the range of short because higher order bits are removed. When floating point value is cast to an integer type, the fractional component will be lost due to truncation. If 1.23 is assigned to an integer, the resulting value will be 1.

Operator Precedence

++ (Postfix), --(Postfix)	Highest
++ (Prefix) --(Prefix) ~ ! + (Unary) – (Unary) (type-cast)	
* / %	
+ -	
>> >>> <<	
> > = < <= instance of	
== !=	
&	
^	
|	
&&	
||	
?:	
= += -= *= /= %= &= |=	Lowest

Type Conversion in Expressions

It is possible to mix two or more different types of data in an expression. When different types of data are mixed, they are converted to the same data type. This is through Java's type promotion rules. Char, byte , short values are promoted to int. If one of the operand is long, the whole expression is promoted to long. If one operand is of type float, the entire expression is promoted to float. If any of the operand is double, the result is double. Type promotion only affects the evaluation of expression.

Example:

```
//To demonstrate type conversion in expressions
class Demo{
public static void main(String args[]) {
{
```

```
byte b; int i;
b=10;
i=b*b;
b=10;
b=(byte)(b*b);
}
}
```

No cast is needed because result is already evaluated to int.

Cast is needed because result is already evaluated to int

I/O Basics

Most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java's Abstract Window Toolkit (AWT) or Swing for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming.

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output. Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. To use the stream classes, **java.io**. must be imported
The abstract classes **InputStream** and **OutputStream** define several key methods thatthe other stream classes implement. Two of the most important are **read( )** and **write( )**,which, respectively, read and write bytes of data. Both methods are declared as abstractinside **InputStream** and **OutputStream**. They are overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.

The abstract classes **Reader** and **Writer** define several key methods that the other streamclasses implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

The Predefined Streams

All Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of the program and without reference to a specific **System** object.

System.out refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console. As you will see, you can wrap these within character based streams, if desired.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing so is not recommended. The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**. To obtain a character based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. You can not construct a Buffered Reader directly fron S**ystem.in**. Instead, you must first convert it into a character stream. To do this, you will use **InputStreamreader** object that is linked to **System.in**, use the constructor shown next:

InputStreamReader(InputStream *inputStream*)

Next, using the object produced by **InputStreamReader,** construct a **BufferedReader** using the constructor shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.* Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

Reading Characters

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is
int read( ) throws IOException

Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered.

The following program demonstrates **read( )** by reading characters from the console until the user types a "q" Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead
 {
public static void main(String args[]) throws IOException
{
char c;
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
c = (char) br.read();
System.out.println(c);
} while(c! = 'q');
} }
```

Reading Strings
To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

```
String readLine( ) throws IOException
```

It returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
public static void main(String args[])
throws IOException {
```

```
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop")); }
}
```

Writing Console Output

Console output is most easily accomplished with **print( )** and **println( )**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section. Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here: void write(int *byteval*) This method writes to the stream the byte specified by *byteval.* Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
public static void main(String args[]) {
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

Program Control Statements

Input Characters from the Keyboard

To read a character from the keyboard, System.in.read() can be used. System.in is the input object attached to the keyboard. The read() method waits until the user presses a kay and then returns the result. The character is returned as integer. So it must be cast into char to assign it to a variable of type char. By default, console input is line buffered. The term buffer here refers to a small portion of memory that is used to characters before they are read by a program. When the user presses ENTER key, the character is returned to the program.

Example:

```
// Read a character from the keyboard
class KbInput{
public static void main(String args[]) throws java.io.IOException
{
char ch;
System.out.println("Type a character !!!! ");
ch=(char) in.read(); // This reads a character from the keyboard
System.out.println("Your Key is " +ch); } }
```

Because System.in.read() is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors.

The simple if Statement

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It takes the following form:

```
if(test-expression)
{
statement-block;
}
statement-x;
```

The statement-block may be a single statement or a group statements. If the `test-expression` is true, the `statement-block` will be executed; otherwise the statement-block will be skipped and the execution will jump to `statement-x`;

Example:

```
if (category==SPORTS)
  marks=marks+bonus_marks;
System.out.println("Marks = " +marks);
```

The if ..... else Statement

```
if(test-expression)
{   True-block statements;   }
else
{   False-block statements;   }
statement-x;
```

If the test-expression is true, then the true-block statements are executed; otherwise, the false-block statements are executed. In either case, either true-block or false-block will be executed, not both. In both the cases, the control will be transferred subsequently to statement-x.

Example:

```
if (code==1)
  boy=boy+1;
else
   girl=girl+1;
```

The Nested if Statement

A nested if statement is an if statement that is the target of another if or else.

Syntax:

```
if (test condition-1)
{ if (test condition-2)
  { statement-1; }
  else
  { statement-22; }
else { statement-3;}
statement-x;
```

If condition-1 is false, then statement-3 will be executed; otherwise it checks condition-2. If condition-2 is true, statement-1 will be executed; otherwise statement-2.

Example:

```
if(gender='F')
{ if (balance>5000)
     bonus=0.05 * balance;
  else
      bonus=0.02 * balance; }
else
      bonus=0.03 * balance;
balance=balance+bonus;
```

The if .... else if Ladder

```
if(condition-1)[
 statement-1;
else if (condition-2)
 statement-2;
.....
else if (condition-n)
 statement-n;
else
   default-statement;
statement-x;
```

This is known as the else if ladder. The conditions are evaluated from top to downwards. As soon as true condition is found, the statement associated with it is executed and the control is transferred to statement-x. When all conditions become false, then the final else containing the default statement will be executed.

Example:

```
// Demonstrate if-else-if ladder
class Ladder {
public static void main (String args[]){
for(int x=0; x<6; x++) {
if(x==1)
```

Output:

x is one
x is two
x is three
x is four
x is not between 1 and 4

```
  System.out.println("x is one");
else if (x==2)
  System.out.println("x is two");
else if (x==3)
  System.out.println("x is three");
else if (x==4)
  System.out.println("x is four");
else
  System.out.println("x is not between 1 and 4); }}}
```

The switch Statement

The test expression is continuously tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general format of switch statement:

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

The expression controlling the switch must be of byte, short int or char. Each value specified in the case statement must be unique. If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The break statement signals the end of particular case and transfers the control to statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place when all matches fail and the control goes to statement-x.

Example:

```
//Demonstrate switch statement
class SwitchDemo{
public static void main(String args[]){
for(int i=0; i< 10; i++)
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;
case 4 : System.out.println("i is four");
      break;
default: System.out.println("i is five or more ");  } } }
```

Output:

```
i is zero
i  is one
i  is two
i  is three
i  is four
i is five or more
i is five or more
i is five or more
i is five or more
i is five or more
```

Technically break statement is optional. When included within a case, it causes the program flow to exit from the entire switch statement and resume at the next statement outside the switch. However, if break is not present with a case, then all statements following the matching case will be executed until a break or end of switch is encountered.

Example:

```
//Demonstrate switch statement without break
class NoBreak
{
public static void main(String args[])
{
for(int i=0; i<=5; i++)
{
switch(i)
 {
case 0 : System.out.println("i is less than 1");
case 1 : System.out.println("i is less than 2");
case 2 : System.out.println("i is less than 3");
case 3 : System.out.println("i is less than 4");
case 4 : System.out.println("i is less than 5");
}
System.out.println();
 }
 } }
```

Output:

```
i is less than 1
i  is less than 2
i  is less than 3
i  is less than 4
i  is less than 5

i is less than 2
i is less than 3
i is less than 4
i is less than 5

i is less than 3
i is less than 4
i is less than 5

i is less than 4
i is less than 5

i is less than 5
```

We can have empty cases as shown in the example below

```
switch(i)
 { case 0 :
  case 1 :
  case 2 : System.out.println("i is 0 1 or 2");
           break;
  case 3 : System.out.println("i is 3");    }
```

The for Loop

The general form of for loop for repeating a single statement is

```
    for(initialization;condition;iteration) statement;
```

For repeating a block, the general form is

```
    for(initialization;condition;iteration)
     { statement- block ; }
```

The initialization is usually an assignment statement that sets the initial value of the loop control variable. The condition is a Boolean expression that determines whether or not the loop will repeat. The iteration expression defines the amount by which loop control variable will change each time the loop is repeated. These three sections must be separated by semicolons. The for loop will continue to execute as long as the condition evaluates to true. Once the condition becomes false, loop will be terminated and control will be transferred to the statement following for.

Example:

```
//Demonstration of for loop
class ForDemo{
public static void main(String args[]) {
int i;
for(i=0; i<5; i++)
 System.out.print(i+ " ");} }
```

Output: 0 1 2 3 4

For loop can proceed in a positive or negative fashion.

Example:

```
//Demonstration of for loop
class NegFor{
public static void main(String args[]) {
int i;
for(i=10; i>5; i--)
 System.out.print(i+ " ");} }
```

Output: 10 9 8 7 6

In case of for loop, the condition always gets tested in the beginning. This means that, the code inside the loop may not be executed at all if the condition is false in the beginning.

Example:

```
 for(count=10; count <5; count++)
  x+=count; // This statement will not get executed at all
```

Some Variations on the for Loop

We can use multiple loop control variables in a for loop. Here commas separate the two statements.

Example:

```
class MulFor{
public static void main(String args[])
{
int i,j;
for(i=0, j=10; i<j; i++,j--)
 System.out.print("i= " +i +" j= "+j);
} }
```

Output
i=0 j=10
i=1 j=9
i=2 j=8
i=3 j=7
i=4 j=6

The condition controlling the loop can be any valid Boolean expression. It does not need to involve the loop control variable.

Example:

```
// Loop until 'S' is pressed
class ForTest
{
public static void main(String args[]) throws java.io.IOException{
int i;
for(i=0, (char)System.in.read()!='S'; i++)
 System.out.print("i= " +i);
} }
```

Missing Pieces

In Java, it is possible for any or all of the initialization, condition or iteration portions of the for loop to be blank.

Example-1:

```
//Demonstrate for loop with some empty parts
class Empty
{
public static void main(String args[]) {
int i;
for(i=0; i<10;) { // Iteration expression is missing here
System.out.println("Pass "+i); i++;
 }}}
```

Example-2:

```
//Demonstrate for loop with some empty parts
class Empty2
{
public static void main(String args[]) {
int i=0;
for(; i<10;)
 { // Initialization and iteration expressions are missing
System.out.println("Pass "+i); i++;
}}}
```

Placing initialization outside the loop is done only when initial value is derived through complex process which can not be contained in the for loop.

The Infinite Loop

An infinite loop can be created using for by leaving conditional expression empty. The structure of an infinite loop is as follows:

```
for(;;) // Intentionally infinite loop
{ .......
  }
```

This loop will run for ever. Use of break statement inside such loop can terminate the loop.

Loops with No Body

In Java, the body associated with a for loop can be empty.

```
//Demonstrate for loop with no body
class Empty3{
public static void main(String args[]) {
int i, sum=0;
for(i=0; i<=10; sum+=i++);
System.out.println("Sum = "+sum); }}
```

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. When this is the case, it is possible to declare the variable inside the initialization portion of the for loop.

Example:

```
// Declare a loop control variable inside the for.
class ForVar {
public static void main(String args[]) {
int sum=0,fact=1;
for(int i=1;i<=5;i++) // i is declared inside the loop
{ sum+=i;
  fact*=i; // i is available only till here }
System.out.println("Sum = " + sum); // i is not known to this part
System.out.println("Factorial = " + fact); } }
```

When a variable is declared inside a for loop, the scope of that variable ends when the for statement does. Outside the for loop, the variable will cease to exist. If the variable is to be used elsewhere in the program, it has to be declared outside the for loop.

The while Loop

Syntax: `while(condition) statement;`

Where the statement may be a may be a single statement or a block of statements. Condition is a valid Boolean expression that controls the loop. Loop repeats while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Example:

```
//Demonstration of while loop
class WhileDemo
{
public static void main(String args[]) {
char ch= 'a';
while (ch<= 'z'){
 System.out.print (ch + "  ");
ch++;
} }  }
```

The do-while Loop

In do... while loop, condition is always tested at the bottom of the loop. This means that do-while loop will always execute at least once. The general form of do-while is

```
do
{
    statements;
} while(condition);
```

Braces are not necessary when only one statement is present, but they are used to increase the readability. The do-while loop executes as long as the conditional expression is true.

```
// Demonstrate the do-while loop
class DoWhileDemo
{
public static void main(String args[])throws java.io.IOException {
char ch;
```

```
do {
  System.out.println("Enter a character");
  ch=(char) System.in.read();
   } while (ch!='q');
} }
```

Use of break to Exit a Loop

It is possible to force an intermediate exit from a loop, bypassing any remaining code in the body of the loop using break statement. When a break statement is encountered inside a loop, the loop is terminated and the program control resumes at the next statement following the loop.

Example:

```
//Using break to exit the loop
class BreakDemo{
public static void main(String args[]) {
int num=100;
for(int 1=0;i<num;i++){
  if(i*i>=num) break; // terminate loop if i*i >=100
System.out.println(i + " "); } }
```

Break statement can be used with any of Java's loops including infinite loop.

```
//Read input until a q is received
class Break2{
public static void main(String args[]) {
char ch;
for(;;){
  ch=(char) System.in.read();
  if(ch=='q') break; // terminate loop if ch is q }
System.out.println("q is pressed "); } }
```

Labelled break Statement (break as a form of goto)

In Java, we can give a label to a block of statements. A label is any valid Java variable name. Label is given to a loop by placing it before the loop with a colon at the end.

```
loop1: for(.........)
  { ......................;
        break loop1;
}
```

//using break with a label

```
class Break4
{
Public static void main(String args[])
{
  int i;
  for(i=1;i<4;i++)
  {
        one:{
```

```
        two:   {
        three:     {
                        system.out.println("\n i is  "+i);
                      if(i==1) break one;
                        if(i==2) break two;
                        if(i==3) break three;

                // this is never reached
                System.out.println("won't print");
                  }
                System.out.println("After block three.");
             }
          System.out.println("After block two.");
           }
        System.out.println("After block one.");
     }
    System.out.println("After for");
}
}
```

The output is,

```
i is 1
After block one.

i is 2
After block two.
After block one.

i is 3
After block three.
After block two.
After block one.
After for.
```

Precisely where you put a label is very important-especially when working with loops. For example:

```
class Break6{
public static void main(String args[])
{
 int x=0,y=0;
Stop1: for(x=0;x<5;x++){
          for(y=0;y<5;y++)  {
                if(y==2) break stop1;
                System.out.println("x     and
y: "+ x + " "+  y);
          }
  }
     System.out.println();
// now, put label immediately before
{
  for(x=0;x<5;x++)
stop2: {
          for(y=0;y<5;y++)  {
          if(y==2) break stop2;
                System.out.println("x     and
y: "+ x + " "+y);        }}  }}
```

The output is,

```
x and y : 0 0
x and y  : 0  1

x and y : 0 0
x and y  : 0  1
 x and y : 1 0
x and y  : 1  1
x and y : 2 0
x and y  : 2  1
 x and y : 3 0
x and y  : 3  1
 x and y : 4 0
x and y  : 4  1
```

The continue Statement

The continue statement forces the next iteration of the loop to take place skipping any code between itself and the conditional expression that controls the loop. The following program uses continue to print even numbers from 1 to 100:

Example:

```
//Use of continue
class contDemo {
public static void main(String args[]) {
for(int i=1;i<=100;i++) {
if((i%2)!=0) continue;
System.out.println( i); } } }
```

Output:
2
4
6
8 ..... .100

Use of continue and break statements

```
// Demonstration of labelled continue
class BreakCont{
public static void main(Sting args[]) {
LOOP1: for(int i=1;i<=100;i++) {
 System.out.println( " ");
if(i>=6) break;
for(int j=1; j<100;j++) {
System.out.print( " * ");
if(j==i) continue LOOP1 } } } }
```

Output:
*
* *
* * *
* * * *
* * * * *

Nested Loops

Writing one loop inside the scope of other loop is called nesting of loops.

```
// Demonstration of nested loops
class Nest{
public static void main(String args[]) {
for(int i=2;i<=10;i++) {
System.out.print("Factots of " + i +": ");
for(int j=2; j<i;j++)
if(i%j==0) System.out.print(j+ " ");
System.out.println();} } }
```

Output:
Factors of 2 :
Factors of 3 :
Factors of 4 : 2
Factors of 5 :
Factors of 6 : 2 3
Factors of 7 :
Factors of 8 : 2 4
Factors of 9 : 3
Factors of 10: 2 5