Chapter: 1

Java/C# Intro, Data types, Operators & Control St.

History, Basics & Keywords, data types, operators, Control statemens.

1.1 History of Java And C#

1.1.1 History of Java

Building upon the rich legacy inherited from ***C*** and ***C++***, ***Java*** adds refinements and features that reflect the current state of the art in programming. Responding to the rise of the online environment, Java offers features that streamline programming for a highly distributed architecture.

The primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices, such as toasters, microwave ovens, and remote controls. Java is a **cross-platform language** that could produce code that would run on a variety of CPUs under ***differing environments***. So we can create ***Platform-Independent (i.e. Portable)*** programs. The Web (The **World Wide Web : WWW**), demanded portable programs. Because, the Internet consists of a diverse, distributed universe populated with many types of computers, operating systems, and CPUs.

(Most computer languages were designed to be compiled for a specific target. For example, consider C++. Although it was possible to compile a C++ program for just about any type of CPU, to do so required a full C++ compiler targeted for that CPU. The problem, however, is that compilers are expensive and time-consuming to create.)

Java Related to C and C++: Java is directly related to both C and C++.

- Java inherits its ***syntax*** from C.
- Its ***object model*** is adapted from C++.

You can think of Java as simply the "**Internet version of C++.**" However, to do so would be a mistake. Java has significant practical and philosophical differences. Although Java was influenced by C++, it is not an enhanced version of C++, it is neither upwardly nor downwardly compatible with C++. Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems.

How Java Relates to C# : Microsoft developed the C# language. C# is closely related to Java. In fact, many of C#'s features directly parallel Java. There are, of course, differences between Java and C#, but the overall "***look and feel***" of these languages is very similar. Both Java and C# share

- The same general C++-style ***syntax***,
- Support ***distributed*** programming,
- Utilize the same ***object model***.

"Will C# replace Java?": The answer is No. ***Java*** and ***C#*** are optimized for two different types of computing environments. Just as ***C++*** and ***Java*** will coexist for a long time to come, so will ***C#*** and ***Java***.

1.1.2 C# History

C# is directly descended from C and C++. It is closely related to Java. We know the ***Modern Age of Programming*** begins with C and ***OOP*** started with C++ and finally when internet takes the computer world Java is emerged. However, Java solved two major issues of programming: Portability and security. JVM and Bytecode solved both

- If you wanted to run a C/C++ program on a different system, it needed to be recompiled to machine code specifically for that environment. Java achieved portability by translating a program's source code into an intermediate language called ***bytecode***. This ***bytecode*** was then executed by the ***Java Virtual Machine (JVM)***. Therefore, a Java program could run in any environment for which a ***JVM*** was available. Also, since the ***JVM*** is relatively easy to implement, it was readily available for a large number of environments.
- As all Internet users know, computer viruses constitute a serious and on-going potential threat. What good would portable programs be if no one could trust them? Who would want to risk executing a program delivered via the Internet? It might contain malicious code. Fortunately, the solution to the security problem is also found in the ***JVM*** and ***bytecode***. Because the **JVM** executes the **bytecode**, it has full control of the program and can prevent a Java program from doing something that it shouldn't.

(Prior to the mainstreaming of the Internet, most programs were written, ***compiled***, and targeted for a specific CPU and a specific operating system. While it has always been true that programmers like to reuse their code, the ability to easily port a program from one environment to another took a backseat to more pressing problems. However, with the rise of the Internet, in which many different types of CPUs and operating systems are connected, the old problem of portability became substantially more important. To solve this problem, a new language was needed, and this new language was Java.)

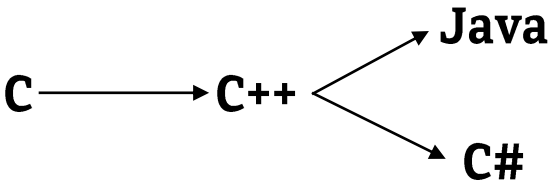
The Creation of C# : While Java has successfully addressed many of the issues surrounding portability and security in the Internet environment, there are still features that it lacks.

- One is ***cross-language inter-operability***, also called ***mixed-language programming***. This is the ability for the code produced by one language to work easily with the code produced by another. Cross-language interoperability is crucial for the creation of large, distributed software systems.
- Another feature lacking in Java is full **integration with the Windows platform**. Although Java programs can be executed in a Windows environment (assuming that the JVM has been installed), Java and Windows are **not closely coupled**.

To answer these and other needs, Microsoft developed C#.

The family tree for C#: The grandfather of C# is C. From ***C***, C# derives its ***syntax***, many of its keywords, and operators. ***C#*** builds upon and improves the ***object model*** defined by C++.

- C# and Java have a bit more complicated relationship. Like Java, C# is designed to produce portable code, and C# programs execute in a secure controlled runtime environment. However, C# is not descended from Java. Instead, C# and Java are more like cousins, sharing a common ancestry, but differing in many important ways.
- New features in C# 3.0 are language-integrated query (LINQ) and lambda expressions: LINQ enables you to write database queries using C# programming elements. Lambda expressions are often used in LINQ expressions. Other innovations include implicitly typed variables and extension methods.



1.2 JAVA : applet, bytecode and JVM

❒ ***Applet :*** Java innovated a new type of networked program called the ***applet***. An **applet** is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. The applet is a **dynamic, self-executing program**. Such a program is an active agent on the client computer, yet it is **initiated by the server**.

☞ An **applet** is downloaded on demand, without further interaction with the user. If the user clicks a ***link*** that contains an **applet**, the **applet** will be automatically downloaded and run in the browser.

☞ **Applets** are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server.

★ ***servlet :*** A **servlet** is a small program that executes on a server. Just as **applets** dynamically extend the functionality of a ***web browser***, **servlets** dynamically extend the functionality of a ***web server***.

[There are two very broad categories of objects that are transmitted between the server and the client: ***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.]

❒ ***The "bytecode" and "JVM" :*** However, networked programs present serious problems in the areas of ***security*** and ***portability***.

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

⇨ ***Bytecode and JVM :*** The output of a Java compiler is not executable code (not an .exe file). Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). In essence, the original JVM was designed as an interpreter for bytecode.

⇨ ***Reason to be "interpretive" :*** many modern languages are designed to be compiled into executable code because they runs faster than interpreter. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs.

➲ ***Portability:*** Translating a Java program into ***bytecode*** makes it much easier to run a program in a wide variety of environments because only the ***JVM*** needs to be implemented for each platform. Once the ***run-time package*** exists for a given system, any Java program can run on it.
NOTE: Although the **details of the JVM will differ** from platform to platform, ***all understand the same Java bytecode***.

➲ ***Security :*** Since the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

☛ ***Bytecode makes Java run faster :*** When a program is ***interpreted***, it generally runs slower than the same program would run if ***compiled*** to **executable code**. However, with Java, the differential between the two is not so great. Because ***bytecode*** has been highly optimized, the use of **bytecode** enables the **JVM** to execute programs much ***faster*** than you might expect.

☛ **JIT**: Java can compile of ***bytecode*** into ***native code (machine code of the User's machine)*** in order to boost performance. It can be achieved by using ***just-in-time (JIT) compiler*** for **bytecode**. When a ***JIT*** compiler is part of the ***JVM***, *selected portions of bytecode* are compiled into **executable code** in real time on a ***piece-by-piece, demand basis***.

- ***JIT*** partially compile a Java program (into several executable files), because Java performs various ***run-time checks*** that can be done only at ***run time***. So it is not practical to compile an entire Java program into executable code all ***at once***.
- A ***JIT*** compiler compiles code as it is needed, during execution. Furthermore, not all sequences of **bytecode** are compiled—only those that will benefit from compilation. The ***remaining code is simply interpreted***.

NOTE: ***Java doesn't support "pointers"***
The pointer feature of C++ can access resource outside of program itself and it can break security. For this reason Java doesn't support pointer.

1.3 Installing Java Development Kit (JDK), Netbeans IDE and compiling a program

Download and install JDK from oracle.com. The JDK supplies two primary programs.

- The first is ***javac***, which is the **Java compiler**.
- The second is ***java***, which is the **standard Java interpreter** and is also referred to as the **application launcher**.

❒ The JDK runs in the *command prompt* environment and uses *command-line* tools. It is not a windowed application. It is also ***not*** an integrated development environment (***IDE***).
NOTE : In addition to the basic command-line tools supplied with the JDK, there are several high-quality IDEs available for Java, such as ***NetBeans*** and ***Eclipse***. An IDE can be very helpful when developing and deploying commercial applications.

1.3.1 Write and compile a Java program:

Before we start the following points should be maintained

☞ ***Enter the program:*** Use simple **text editor** to write the program and crate the ***source-file***.

☞ ***Name of the source-file:*** A **source file** is called "***compilation unit***" in Java, and a Java compiler require the ***".java"*** file extension after the *source-file name*.

➢ In java all code must reside inside a class called "***main class***". By conversion, the name of the main class should match the name of the ***source-file***. Same name helps us to **maintain** and **organize** the codes easily.

➢ The ***main-class*** name and the ***source-file*** name must maintain the **capitalization**, because Java is case sensitive.

☞ ***Compiling the program:*** Before use the java compiler "***javac***" we have to add it in the "system-path".

▶ **Adding in system-path**:
$control\ panel \rightarrow sysem\ security \rightarrow system \rightarrow advanced\ system\ setting \rightarrow Environment\ variables(bottom)$
$user\ variavle \rightarrow path\ (add\ path)$ and $sytem\ variable \rightarrow path(add\ path)$
If path doesn't exist crate one. Before edit, make sure that you copied the desired path (folder location in a drive) of the program.
`C:\Program Files\Java\jdk1.8.0_231\bin;` is the location for ***Java compiler*** javac.
`C:\Windows\Microsoft.NET\Framework\v4.0.30319;` is the location for ***C# compiler*** csc.exe.

▶ **Compiling the program:** To compile the "***Example.java***" program, execute the **compiler**, ***javac***, specifying the name of the source file on the ***command line***, as shown here: `javac Example.java`

The ***javac*** compiler creates a file called "***Example.class***" that contains the **bytecode version of** the program. Remember, **bytecode** is ***not executable code***. Bytecode must be executed by a JVM (interpreter). Thus, ***the output of javac is not code that can be directly executed***.

▶ **Executing the application/program:** To actually run the program, you must use the **Java interpreter**, ***java***. To do so, pass the **class name** "***Example***" as a command-line argument, as shown here: `java Example`

Notice, no file extension "***.class***" is used.

1.2.3 The First Sample Program Line by Line:

```
                    /* This is a simple Java program. Call this file Example.java. */
class Example {     // A Java program begins with a call to main( ).
          public static void main(String args[]){ System.out.println("Java drives the Web.");
                                                  }
          }
```

⌦ ***Method:*** In Java any function/subroutine is called "**method**"

☑ The first line is a ***multiple line comment*** section.

☑ The next line is: **`class Example {`**
This line uses the keyword ***class*** to declare that a ***new class*** is being defined. ***Example*** is the ***name of the class***. The class definition begins with the opening curly brace "**{**" and ends with the closing curly brace "**}**". ***The elements between the two braces are members of the class***. All program activity occurs within this ***main-class***. On the right there is a ***single line comment***.

☑ The next line: **`public static void main (String args[]) {`**
This line begins the ***main()*** **method** (function/subroutine called method in Java). This is similar to C/C++'s **`main()`** ***function***, and some ***modifier*** and ***specifier*** are used in addition. This is the line at which the program will begin ***executing***. All Java applications ***begin execution*** by calling **`main()`**. Now let's see why the specifiers/modifiers are used :

- ⇨ ***public*** is an **access modifier** defined previously in C/C++. In this case, ***main()*** must be declared as ***public***, since it must be called by code outside of 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. [***Recall C/C++ 14.3: Static Class Members:*** **A** ***static*** *member variable exists before any object of its class is created*]
- ⇨ The keyword ***void*** simply tells the compiler that ***main()*** does not return a value.
- ⇨ In ***main()*** there is only one parameter, ***String args[]***, which declares a parameter named ***args***. This is an array of objects of type ***String***. Objects of type ***String*** store sequences of characters. In this case, ***args*** receives any ***command-line arguments*** present when the program is executed [***Recall 5.4 Pass Arguments to main()*** of C/C++, the command line argument]. This program does not make use of this "***String args[]***" information, but other programs will.
- ⇨ The last character on the line is the ***{***. This signals the start of ***main()***'s body. All of the code included in a ***method*** will occur between the ***method's*** opening curly brace and its closing curly brace.

☑ The next line of code is the statement: **`System.out.println("Java drives the Web.");`**

- ⇨ This line outputs the string "**Java drives the Web.**" followed by a new line on the screen. Output is done by the built-in ***println()*** method.
- ⇨ The line begins with ***System.out***. "***System***" is a predefined class that provides access to the ***system***, and ***out*** is the output stream that is ***connected to the console***. Thus, ***System.out*** is an object that encapsulates ***console output***.

NOTE: [1] All statements in Java end with a **semicolon**.
[2] The Java compiler will compile classes that do not contain a ***main()*** method. But, the Java interpreter would report an error because it would be unable to find the ***main()*** method.
[3] In Java there is no " ***return 0;*** " statement. Program execution terminates when "**}**" is reached. The ***{*** and ***}*** exist only in your program's source code. Java does not, per se, execute the ***{*** or ***}***.

1.2.4 println() and print(): ***println()*** outputs the data in the "***new line***" after each call. The ***print()*** method is just like ***println()***, except that it does not output a ***new line*** after each call.

☞ Using the + operator, you can chain together as many items as you want within a single ***println()*** statement. Eg:

```
System.out.println("var1 contains " +x);
```

In this statement, the **plus** "**+**" sign causes the value of ***x*** to be displayed after the string that precedes it.

☞ To print a ***blank line***, simply call ***println()*** without any arguments. i.e. "**println**():"

1.3 Variable Declarations, Data-types, Operator Basic

Variable declaration is same as C/C++ : ***type var_name;*** and ***int***, ***float*** and ***double*** are available for numerical type data.
All operators rules for **+**, **−**, *****, **/** including "**=**" are same as C/C++. Relational operators are also same **<**, **>**, **<=**, **>=**, **==**, **!=**.

NOTE: Why ***int***, ***float***, ***double*** for numerical data type? Why not only one data type: Because ***int*** is faster than ***float***, ***double***. Also ***int*** require less memory. So separating ***int*** and ***float/double*** makes program fast and efficient.

1.4 Control statement: "if" & "for"

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

- ➢ Here "***condition***" is a ***boolean*** statement, i.e. ***true*** or ***false***.
- ➢ No "***;***" after statement block's "***}***". "***statement block***" and other is exactly same as C/C++.

1.5 The Java Keywords

Fifty keywords are currently defined in the Java language (see Table 1-1). These keywords, combined with the syntax of the operators and separators, form the definition of the Java language. In addition to the keywords, these are ***values*** defined by Java: ***`true`***, ***`false`***, and ***`null`***. These keywords and values cannot be used as names for a ***variable***, ***class***, or ***method***.

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

- The keywords const and goto are reserved but not used.
- You may not use these words for the names of variables, classes, and so on.

1.6 The Java Class Libraries

The Java environment relies on several ***built-in class libraries*** that contain many ***built-in methods*** that provide support for such things as ***I/O***, ***string*** handling, ***networking***, and ***graphics***. The standard classes also provide support for a ***graphical user interface (GUI)***. Thus, Java as a totality is a combination of the Java language itself, plus its ***standard classes***. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, ***part of becoming a Java programmer is learning to use the standard Java classes***.

☞ Java is a ***free-form*** language, meaning that it does not matter where you place statements relative to each other on a line (multiple statements on a line is acceptable).

1.7 Java's Primitive Types

Java contains ***two*** **general categories** of built-in data types: ☑ object-oriented ☑ non-objectoriented

❒ At the core of Java are **`eight`** ***primitive*** (also called ***elemental*** or ***simple***) types of data, which are shown in the Table. The term ***primitive*** is used to indicate that these types are not objects in an object-oriented sense, but rather, normal binary values.

Type group	***Type specifier***	***Meaning***	***Width in bit***	***Range***
Boolean	***boolean***	Represents true/false values	1 bit (true), 0 bit (false)	0 and 1
Integer numbers	***byte***	8-bit integer	8 bit	–128 to 127
	short	Short integer	16	–32,768 to 32,767
	int	Integer	32	–2,147,483,648 to 2,147,483,647
	long	Long integer	64	**–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**
Character	***char***	Character	16	0 to 65,536
Floating numbers	***double***	Double-precision floating point	32	–2,147,483,648 to 2,147,483,647
	float	Single-precision floating point	64	**–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**

☞ Size of Boolean data type "***`boolean`***" is ***`1`*** bit long. ***`1`*** is for "***`true`***" and ***`0`*** is for "***`false`***". However, the actual size of a boolean variable in memory is not precisely defined (undefined?), by java specification.

☞ ***`double`*** is the most commonly used because all of the ***math functions*** in Java's class library use double values. For example, the ***`sqrt()`*** method (which is defined by the ***standard Math class***) returns a ***`double`*** value that is the square root of its ***`double`*** argument.

☞ In Java ***`long`*** is ***64 bit*** and in C/C++ ***`long int`*** is ***32 bit***. Also there is no ***`unsigned`*** (+ve only) in Java.

☞ Characters in Java is actually ***int*** type, also it can be used for integer data types. However, it is generally used for character type data. In Java, ***`char`*** is ***not 8-bit*** like **C/C++**. Instead, Java uses ***`Unicode`***. Unicode defines a character set that can represent all of the ***characters found in all human languages*** and for this it requires ***16 bit***. In Java, ***`char`*** is an ***`unsigned`*** 16-bit type having a range of 0 to 65,536. The standard 8-bit ***ASCII character set is a subset of Unicode*** and also available in Java.

☞ A character variable can be assigned a value by enclosing the character in **single quotes** which makes the character act like a constant (called ***character constants*** Recall 1.2.1 C/C++, Constant, 'A' is a char constant). Eg: **`char`** `ch; ch='b';`

NOTE: ***Math class*** is similar to **`C's`** ***math.h*** header file. As well as all other **standard classes** of Java they are all like **header files**. To access a member of Math class (i.e. a method) we use the "." operator as we did for ***`println()`***. Eg: to access ***`sqrt()`***:

```
Math.sqrt(x*x + y*y)
```

1.8 Literals (also commonly called constants Eg: 100 is a literal.)

By default, **integer literals** are of type ***`int`***. To specify a ***`long`*** literal, append an ***`l`*** or an ***`L`***. For example, ***`12`*** is an ***`int`***, but ***`12L`*** is a ***`long`***. By default, **floating-point literals** are of type ***`double`***. To specify a ***`float`*** literal, append an ***`F`*** or ***`f`***. Eg: ***`10.19F`*** is of type ***`float`***.

☞ Although integer literals create an ***int*** value by default, they can still be assigned to variables of type ***char***, ***byte***, or ***short*** as long as the value being assigned can be *represented* by the **target type**. An **integer literal** can always be assigned to a long variable.

In JDK 7, you can embed one or more ***underscores*** into an ***integer*** or ***floating-point*** literal. Doing so can make it easier to read values consisting of many digits. When the literal is compiled, the underscores are simply discarded. Here is an example: ***`123_45_1234`*** specifies the value ***`123,451,234`***. The use of "_" is particularly useful when encoding things like part numbers, customer IDs, and status codes .

1.9 Character escape sequences or backslash character constants

Most are same as C/C++. Except ***octal*** and ***hexadecimal*** (Recall C/C++ ***2.7***). All presented in the following table.

Escape Sequence	**`\'`**	**`\"`**	**`\\`**	**`\r`**	**`\n`**	**`\f`**	**`\t`**	**`\b`**	**`\ddd`**	**`\uxxxx`**
Description	Single quote	Double quote	Backslash	Carriage return	New line	Form feed	Horizontal tab	Backspace	Octal constant (ddd is an octal)	Hexadecimal constant (xxxx is a hexadecimal)

1.10 Hexadecimal, Octal, and Binary Literals

Java allows you to specify integer literals in ***hexadecimal*** or ***octal*** instead of **decimal**.

- A ***hexadecimal*** literal must begin with **0x** or **0X** (a **zero** followed by an **x** or **X**). Eg: `hex = 0xFF; // 255 in decimal`
  - Java also allows ***hexadecimal*** floating-point literals, but they are seldom used.
- An ***octal*** literal begins with a ***zero***. Eg: `oct = 011; // 9 in decimal`
- In **JDK 7**, it is possible to specify an integer literal by use of ***binary***. To do so, precede the binary number with a **0b** or **0B**. For example, this specifies the value 12 in binary: **0b**1100.

1.11 String literals

In Java character string is always inside double quotes **""**. And character constant is inside single quotes **' '**. So **'k'** and **"k"** are not the same. **'K'** is for ***character constant*** and **"K"** is for ***character string***.

- ✓ Escape sequence in "string" is same as C/C++. Eg: `"A\tB\tC\na\tb\c".`

1.12 Initialization and Dynamic Initialization

- ***Initialization*** is same as C/C++: ***type var_name = value ;*** Eg: **int** count = 10; **char** ch = 'X'; **float** f = 1.2F;
  - ☞ Using a comma-separated list, you can give one or more same type variables an initial value. For example: **int** a, b = 8, c = 19, d; In this case, only ***b*** and ***c*** are initialized.
- ***Dynamic Initialization:*** It is also same for C/C++ local variables. However **global** variables do not support dynamic initialization (initialize a global variable using other variable is not supported, Recall **3.4 Variable initialization** C/C++).
  - ☞ Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared (i.e. ***If the expression variables do not go out of the scope***. Because local variables destroyed after the function call). For example,

```
double radius = 4, height = 5;
double volume = 3.1416 * radius * radius * height; // dynamically initialize volume
```

1.13 Scope and Lifetime of Variables

A ***block*** (block of codes) is begun with "**{**" and ended by "**}**". A block defines a ***scope***. A **new scope is created during creation of new block.**

- ☞ A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- C/C++ defines two general categories of scopes: ***global*** and ***local***. Although supported by Java, these are not the best ways to categorize Java's scopes. In Java:
  - [1] ***Scopes defined by class***. We'll discuss this kind later with classes.
  - [2] ***Scopes defined by method***. For now we focus on
  - ⇨ ***Scope defined by methods*** is mostly similar to C/C++'s ***local variables*** scope. The scope defined by a method **begins with its opening curly brace**. The ***parameters*** of the method are *also included within the method's scope.*
  - ⇨ Variables declared inside a ***scope*** are not visible/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***. Objects of outer scope will be **accessible** to inner scope. However, the reverse is not true. Objects declared within the inner scope ***will not be visible*** outside it. Eg: consider the following program (similar to C/C++):

```
class ScopeDemo {
public static void main(String args[]) { int x;                    // known to all code within main
    x = 10;
    if(x == 10) {  int y = 20;      // y is known only to if block
                   System.out.println("x and y: " + x + " " + y);      // x and y both accessible
                   x = y * 2; }
    y = 100;                       // Won't run, returns an error. y is unknown here
    System.out.println("x is " + x);     }    }       // x is still accessible
```

- Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the ***start*** of a ***method***, it is ***available*** to all of the code within that method. Conversely, if you declare a variable at the ***end*** of a block, it is effectively ***useless***, because no code will have access to it.
- Variables are created when their **scope** is ***entered***, and destroyed (lose its value) when their ***scope*** is ***left*** [Recall C/C++ 3.2]. Thus, the ***lifetime*** of a variable is confined to its scope. Therefore, variables declared within a method will not hold their values between calls to that method. If a variable declaration includes an ***initializer***, that variable will be ***reinitialized*** each time the block in which it is declared is entered. Example:

```
class VarInitDemo {
public static void main(String args[]) {  int x;
        for(x = 0; x < 3; x++) { int y = -1; // y is initialized each time block is entered
                        System.out.println("y is: " + y); // this always prints -1
                        y = 100;
                        System.out.println("y is now: " + y);   }          }  }
```

OUTPUT :
```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

- ❒ ***C/C++ Exception:*** Although **blocks** can be **nested**, no variable declared within an **inner scope** can have the same name as a variable declared by an **enclosing scope**. For example, following tries to declare two separate variables with the same name, will not compile.

```
public static void main(String args[]) { int count;
        for(count = 0; count < 10; count ++) { System.out.println("This is count: " + count);
                                               int count;          // illegal!!!
                                               for(count = 0; count < 2; count++) System.out.println("Gives error!");
                                               }  }
```

In C/C++ there is no restriction on the names that you give **variables** declared in an ***inner scope***. Thus, in C/C++ the declaration of ***count*** within the block of the ***outer for loop*** is completely valid, and such a declaration **hides the outer variable**.

1.14 Operators

Operator	Meaning (Arithmetic)	Operator	Meaning	Operator	Meaning
+	Addition (also unary plus)	*Relational operators*		*Logical operatrs*	
–	Subtraction (also unary minus)	==	Equal to	&	AND
*	Multiplication	!=	Not equal to	|	OR
/	Division	>	Greater than	^	XOR (exclusive OR)
%	Modulus	<	Less than	||	Short-circuit OR
++	Increment	>=	Greater than or equal to	&&	Short-circuit AND
––	Decrement	<=	Less than or equal to	!	NOT

- ✓ All ***arithmetic*** and ***relational*** operators can be applicable to both ***numeric*** and ***char*** type of data.
- ✓ ***relational*** operators are not applicable to **Boolean** type data. i.e. `true > false` has no meaning in ***Java***.
- ✓ ***relational*** operators and ***logical*** operators can act together. For ***logical*** operators the operands must be **Boolean** type .

❒ ***Increment , Decrement and their postfix-prefix:*** Both the **++** and **--** operators can either precede (***prefix***) or follow (***postfix***) the operand. Eg: `x = x + 1;` can be written as ***prefix:*** `++x;` or as ***postfix:*** `x++;` In this case there is no difference whether the increment is applied as a prefix or a postfix. (Recall: C/C++ **2.6**)

☞ However, when an **increment** or **decrement** is used as part of a ***larger expression***, there is an important difference.

prefix: `++x;`	***postfix:*** `x++;`
When an increment or decrement operator precedes its operand, Java will ***perform*** the corresponding ***operation prior*** to *obtaining the operand's value* for use by the rest of the expression. `x = 10; y = ++x;` In this case, **y** will be set to **11**.	If the operator follows its operand, Java will ***obtain the operand's value before*** incrementing or decrementing it. `x = 10; y = x++;` then **y** will be set to **10**. In both cases, ***x*** is still set to ***11***;

❒ ***C/C++ logical operator modification:***
[So what happened to bitwise AND/OR "&' |' ?]

- ♣ "**&&**" is for '***AND***' operation in C/C++, Which is changed to "**&**" to denote "***AND***" operation in ***Java***.
- ♣ "**||**" is for '***OR***' operation in C/C++, Which is changed to "**|**" to denote "***OR***" operation in ***Java***.
- ♣ "**&&**" is now denote " ***Short-circuit AND***" operation in ***Java***.
- ♣ "**||**" is now denote " ***Short-circuit OR***" operation in ***Java***.
- ♣ Newly introduced '**^**' operator for "***XOR***" operation in ***Java***. This is unavailable in C/C++.

❒ ***Explanation for "Short-circuit AND/OR"::***

Truth table. T:true, F:false

p	q	p&q	p|q	p^q	!p
T	**T**	**T**	**T**	**F**	**F**
T	**F**	**F**	**T**	**T**	**F**
F	**T**	**F**	**T**	**T**	**T**
F	**F**	**F**	**F**	**F**	**T**

☞ The only difference between the normal and ***short-circuit versions*** is that the normal operands will always evaluate ***each operand***, but *short-circuit* versions focus only ***first operand*** and will evaluate the ***second operand iff first operand*** is true.

- ☑ Notice the truth table, in an ***AND*** "***&***" operation, if the ***first*** operand is ***false***, the ***outcome*** is ***false*** no matter what value the ***second*** operand has.
- ☑ In an ***OR*** operation, if the ***first*** operand is ***true***, the outcome of the operation is ***true*** no matter what the value of the ***second*** operand.

So, in these two cases ***there is no need to evaluate the second operand***. So time can be saved and more efficient code can be produced if we use "**&&**", "**||**" the "short-circuit versions" of "**AND**", "**OR**" .

☞ "&&" ***and*** "||" ***can be used to solve following kind of situations:*** Since the modulus operation involves a division, the short-circuit form of the AND is used to prevent a divide-by-zero error.

```
n = 10;
d = 2;
if(d != 0 && (n % d) == 0) System.out.println(d + " is a factor of " + n);   // Since d is 2, the second operand "modulus" is evaluated
d = 0;
if(d != 0 && (n % d) == 0) System.out.println(d + " is a factor of " + n);   // Since d is zero, the second operand "modulus" is not evaluated.
if(d != 0 & (n % d) == 0) System.out.println(d + " is a factor of " + n);    // without && operator. This will cause a divide-by-zero error.
```

☞ ***Side effect of "short circuit" forms:*** However sometimes we need both "&", "|" and "&&", "||". In some cases you will want ***both operands*** of an ***AND*** or ***OR*** operation to be evaluated because of the ***side effects produced***. For Example:

```
int i;
i = 0;
/* i incremented even though if statement fails. */
      if(false & (++i < 100) ) System.out.println("won't be displayed");
      System.out.println("if stmt executed:" +i);          // displays 1

/* In this case, i is not incremented because the short-circuit operator skips the increment. */
      if(false && (++i < 100) ) System.out.println("won't be displayed");
      System.out.println("if statement executed: " + i);   // still 1 !!
```

❒ ***SHORTHAND assignment operators*** (Recall **C/C++ 7.8.3**): Java allow some shorthand assignment operators similar to C/C++.

☛ **To create a chain of assignments:** For example, consider this fragment: `int x, y, z;`

```
x = y = z = 100;      // set x, y, and z to 100
```

Here 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**. Using a "***chain of assignment***" is an easy way to ***set a group of variables to a common value***.

☛ ***Shorthand / compound assignment operators:*** `x = x + 10;` can be written as `x += 10;` and `x = x - 100;` can be written as `x -= 100;` In both cases the operator pairs **+=** , **-=** tells the compiler to assign to **x** the value of " ***x plus 10*** " , " ***x minus 100*** ".

This shorthand will work for all the binary operators in Java .
The general form of the shorthand is: ***var op = expression;***
The arithmetic and logical shorthand assignment operators :

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

1.15 Operator Precedence:

Highest														***Lowest***
x++ **x––** (Postfix)	**++x** (prefix) **––x** (prefix) **~** **!** **+** (unary) **–** (unary) (type-cast)	***** **/** **%**	**+** **-**	**>>** **>>>** **<<**	**>** **>=** **<** **<=** **instanceof**	**==** **!=**	**&**	**^**	**|**	**&&**	**||**	**?:**	**->**	**=** **op=**

1.16 Type Conversions and type-cast (Recall C/C++ 3.5, 3.6):

When compatible types are mixed in an assignment, the value of the ***right*** side is automatically converted to the **type** of the ***left*** side. Eg: `int i; float f; i = 10; f = i;` here, the value in ***i*** is converted into a ***float*** and then assigned to ***f***. Not all types are compatible, for example, ***boolean*** and ***int*** are not compatible.

❒ ***Type conversion:*** 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.

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, and both int and byte are integer types, so an automatic conversion from byte to int can be applied.

Example: ***long*** to ***double*** is a widening conversion and is legal. But there is no automatic conversion from ***double*** to ***long***, since this is not a widening conversion.

```
long L, l;     double D, d;
L = 100123285L;  d = 90123285.0;
D = L;         // legal
l = d;         // illegal!!!
```

❒ ***Type-Cast for Incompatible Types:*** When **type-conversion** is illegal (***long*** to ***double***) or incompatible (***Boolean*** to ***int***) we can use the ***type-cast***. A ***cast*** is an ***instruction to the compiler*** to convert one type into another. A ***cast*** has this general form:

(***target-type***) ***expression***

Here, ***target-type*** specifies the ***desired type to convert*** the specified ***expression*** to. Eg: `double x,y; int k =(int)(x/y);`

- ➲ The ***parentheses*** surrounding **x/y** are necessary. Otherwise, the ***cast*** to ***int*** would apply ***only to the x.***
- ➲ The ***cast*** is necessary here because there is ***no automatic conversion*** from ***double*** to ***int***.

☞ ***Data-loss during type-cast:*** When a cast involves a **narrowing conversion**, information might be lost. When casting a ***long*** into a ***short***, information will be lost if the ***long's value*** is ***greater than the range*** of a ***short*** because its **high-order bits** are removed. When a **floating-point value** is cast to an **integer type**, the fractional component will also be lost due to truncation. For example, if the value ***1.23*** is assigned to an ***integer***, the resulting value will simply be ***1***. The ***0.23*** is lost. For Example:

```
byte b;  int i;
i = 100; b = (byte) i;      // No loss of info here. A byte can hold the value 100
i = 257; b = (byte) i;      // Information loss this time. A byte cannot hold the value 257
```

❒ ***Type Conversion in Expressions:*** When different types of data are mixed within an expression, they are all converted to the same type. This is accomplished through the use of Java's type promotion rules.

Recall C/C++ 3.5

[1] First, all ***char***, ***byte***, and ***short*** values are promoted to ***int***.
[2] Then, if one operand is a ***long***, the whole expression is promoted to ***long***.
[3] If one operand is a ***float*** operand, the entire expression is promoted to ***float***.
[4] If any of the operands is ***double***, the result is ***double***.

☞ **type promotions** of a variable ***expire*** after the evaluation of the expression. For example, if the value of a ***byte*** variable is promoted to ***int*** inside an expression, outside the expression, the variable is still a ***byte***. Type promotion only affects the evaluation of an expression.

☞ ***type-cast*** **in expression:** Even though we have **type-promotion** in an expression, but in some cases we still need ***type-cast***. Eg:

```
byte b;   int i;
b = 10;   i = b * b;              // OK, no cast needed
b = 10;   b = (byte) (b * b);     // cast needed!!
```

- ▶ No cast is needed when assigning ***b*b*** to ***i***, because ***b*** is promoted to ***int*** when the expression is evaluated.
- ▶ However, when you try to assign ***b*b*** to ***b***, you do need a ***cast***—back to ***byte***! Keep this in mind if you get *unexpected type-incompatibility error messages* on expressions that would otherwise seem perfectly OK.
- ▶ This same sort of situation also occurs when performing operations on ***chars***. For example, in the following fragment, the ***cast*** back to ***char*** is needed because of the promotion of ***ch1*** and ***ch2*** to ***int*** within the expression:

```
char ch1 = 'a',   ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Without the cast, the result of adding ***ch1*** to ***ch2*** would be ***int***, which can't be assigned to a ***char***.

1.17 Spacing and parenthesis

The two expressions are the same, but the second is easier to read: **x=10/y*(127/x);** **x = 10 / y * (127/x);**

❖ Parentheses increase the precedence of the operations contained within them, just like in algebra. Use of redundant or additional parentheses will not cause errors or slow down the execution of the expression.
❖ Additional spaces makes a good readable code.

1.18 Input Characters from the Keyboard

To read a character from the keyboard, we will use ***System.in.read()***. By default, console input is line buffered.

♠ ***System.in*** is the input object attached to the keyboard. The ***read()*** method waits until the user presses a key and then returns the result. ***The character is returned as an integer***, so it must be ***cast*** into a ***char*** to assign it to a **char variable**. (Not useful for ***int***)

*[Here, the term **buffer** refers to a small portion of memory that is used to hold the characters before they are read by your program. In this case, the **buffer** holds a complete line of text. As a result, you must press **ENTER** before any character that you type will be sent to your program.]*

❒ ***Example 1:*** Here is a program that reads a character from the keyboard:

```
class KbIn {
public static void main(String args[]) throws java.io.IOException{
        char ch;
        System.out.print("Press a key followed by ENTER: ");
        ch = (char) System.in.read();      // get a char. Read a character from the keyboard.
        System.out.println("Your key is: " + ch);      } }
```

☞ Notice that ***main()*** begins like this:

```
public static void main(String args[])
throws java.io.IOException {
```

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***. It is part of Java's ***Exception Handling*** mechanism.

NOTE: Real-world Java programs and applets will be graphical and window based, not console based. So keyboard use isn't popular.

1.19 if-else, Nested if, if-else-if ladder

❒ ***if, if-else :*** Everything is similar to C/C++. In Java "**if(condition)**" is used and in C/C++ "**if(expression)**" is used. In Java "***condition***" must be a "***Boolean expression*** " which returns **true/false** value. (Recall C/C++ 2.3, 2.4)

❒ **Nested *if*** : In a nested if-else, an invisible block appears around the nearest if-else of the same visible block. Eg:

```
if(i == 10) { if(j < 20) a = b;
if(k > 100) c = d;
else a = c;     // this else refers to if(k > 100), Makes an if-else block. Both reacts when "j<20" is true. None will be executed when "k<20" is false
}
else a = d;     // this else refers to if(i == 10)
```

☞ The key point about ***nested ifs*** in Java is that an ***else*** statement always refers to the nearest ***if*** statement that is within the same block as the ***else*** and not **already associated** with an ***else***. (Recall C/C++ 2.10 Last portion)

❒ ***if-else-if ladder:*** Exactly same as C/C++ and "**condition**" must be used instead of "**expression**". Last ***else*** act as ***default condition***.

1.20 Switch statement(similar C/C++ switch . Recall C/C++ 2.17)

In C/C++ we used only ***value*** to control ***switch*** i.e. "**switch(value){ }**" where value must be ***int*** or ***char*** constant. In Java we use "***expression***" i.e "**switch(expression){ }**", generally ***expression*** must be of the types: ***byte***, ***short***, ***int***, ***char*** or an ***enumeration***. In **JDK 7**, *expression* can also be of type ***String***. This means that modern versions of Java can use a **string to control** a ***switch***.

- ❖ Frequently, the ***expression*** controlling a ***switch*** is simply a *variable* rather than a *larger expression.*
- ❖ Each ***value*** specified in the ***case*** statements must be a *unique constant expression* (such as a literal value).

❒ You can have empty cases (***Recall C/C++ 2.17 Note 8***), as shown in this example:

```
switch(i) {   case 1:
              case 2:
              case 3: System.out.println("i is 1, 2 or 3"); break;
              case 4: System.out.println("i is 4"); break; }
```

In this fragment, if ***i*** has the value ***1***, ***2***, or ***3***, the first ***println()*** statement executes. If it is ***4***, the second ***println()*** statement executes. The "***stacking***" of **cases**, as shown in this example, is common when several cases share common code.

❒ ***Nested Switch:*** Similar to C/C++ 2.18 Nested switch.

1.21 for loop (with variations)[Recall C/C++ 2.5 , 2.11]

Similar to C/C++'s ***for loop*** **for(initialization; condition; iteration) {statement;}**

The ***initialization*** is usually an assignment statement that sets the initial value of the loop control variable, which acts as the counter that controls the loop. The ***condition*** is a ***Boolean expression*** that determines whether or not the loop will repeat. The ***iteration*** expression defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be ***separated by semicolons***.

❒ ***Some Variations on the for Loop***

[1] **Multiple loop control variables** can be used in the ***for*** loop. Multiple loop control variables are often convenient and can simplify certain algorithms. Any number of ***initialization*** and ***iteration*** statements are possible, but, more than two or three make the ***for loop*** unwieldy. Consider the following code:

```
for(i=0, j=10; i<j; i++, j--) System.out.println("i and j: " + i + " " + j);
```

▶ Here, ***commas*** separate the ***two initialization*** statements and the ***two iteration*** expressions. When the loop begins, both **i** and **j** are initialized. Each time the loop repeats, **i** is ***incremented*** and **j** is ***decremented***.

[2] The ***condition*** controlling the loop can be **any valid Boolean** expression. It *does not need to involve the* ***loop control variable***. Eg: the loop continues to execute until the user types the letter **S** at the keyboard:

```
for(i = 0;  (char)System.in.read() != 'S';  i++) System.out.println("Pass #" + i);
```

[3] It is possible for any or all of the ***initialization***, ***condition***, or ***iteration*** portions of the for loop to be blank. For example,

```
for(i=0; i<10; ) {  System.out.println("Pass #" + i);
                    i++; }        // increment loop control variable is inside the for loop
```

- ❖ or ***iteration may not present at all***. Eg:

```
for(i = 0;  (char)System.in.read() != 'S';  ) System.out.println("Again");
```

- ❖ The initialization portion can be moved out of the ***for***:

```
int i = 0;                          // move initialization out of loop
for( ; i<10 ; ) {   System.out.println("Pass #" + i); i++; }
```

❖ ***The Infinite Loop:*** To create an ***infinite loop*** use the ***for*** by ***leaving the conditional expression empty***. Eg:

```
for(;;){ }  // intentionally infinite loop
```

This loop will run forever. (It can be terminated by using the ***break*** statement.)

[4] ***Loops with No Body:*** In Java, the body associated with a ***for loop*** (or any other loop) can be **empty**. This is because a null statement is syntactically valid. They are often useful. For example, the following program uses one to sum the numbers 1 through 5: `for(i = 1; i <= 5; sum += i++) ;`

➲ Notice that the summation process is handled entirely within the ***for*** statement, and ***no body is needed***.

➲ Pay special attention to the iteration expression: `sum += i++`. It is equivalent to: `sum = sum + i; i++;`

[5] ***Declaring Loop Control Variables Inside the for Loop:*** Often the variable that controls a ***for*** is needed only for the purposes of the loop and is not used elsewhere. In this case, we declare the variable inside the initialization portion of the ***for***. Eg:

```
int sum = 0, fact = 1;
for(int i = 1; i <= 5; i++) {     sum += i;      // i is known throughout the loop
                                  fact *= i; }
```

➲ In this example, ***i*** is not accessible outside the ***for*** loop. If you need to use the loop control variable ***elsewhere*** in your program, you will ***not be able to declare*** it inside the ***for*** loop.

➲ NOTE: the ***scope*** of that variable ends when the ***for*** statement does. (That is, ***the scope of the variableis limited to the for loop***.) Outside the ***for*** loop, the variable will cease to exist.

[6] ***The Enhanced for Loop:*** Relatively recently, a new form of the ***for*** loop, called the ***enhanced for***, was added to Java. The ***enhanced for*** provides a ***streamlined way to cycle through the contents of a collection of objects***, such as an array. We'll discuss with array.

1.22 while and do-while loop (similar to C/C++)

While loop: `while(condition){}` ***Do-While loop:*** `do{}while(condition);`

1.23 Nested Loops:

One loop can be nested inside of another. Eg: To find **factors** of numbers

```
for(int i=2; i <= 100; i++){ System.out.print("Factors of " + i + ": ");
                             for(int j=2; j<i; j++) if((i%j) == 0) System.out.print(j + " ");
                             System.out.println(); }
```

1.24 "break" and "continue"

Both are similar to C/C++'s "**break**" and "**continue**" (Recall ***C/C++ 2.15, 2.16***). Remember the following points:

❒ The break statement can be used with any of Java's loops, including intentionally infinite loops. Eg: force user to types the letter q:

```
for( ; ; ) { ch = (char) System.in.read(); // get a char
             if(ch == 'q') break; }
```

❒ When used *inside* a **set of nested loops**, the ***break*** statement will break out of only the ***innermost loop***. For example:

```
for(int i=0; i<3; i++){   System.out.println("Outer loop count: " + i);
                          System.out.print(" Inner loop count: ");
                          int t = 0;
                          while(t<100){ if(t == 10) break;  // terminate loop if t is 10
                                        System.out.print(t + " ");
                                        t++;}
                          System.out.println();  }
System.out.println("Loops complete."); }
```

❒ The ***break*** statement in the ***inner loop*** causes the termination of only that loop. The *outer loop is unaffected*. The ***break*** that terminates a ***switch*** statement affects only that ***switch statement*** and not any **enclosing loops**.

1.25 "break" and "continue" with *LABEL* (Replacing "goto")

O ***Labeled break:*** The ***break*** can be employed by itself to provide a "***civilized***" form of the ***goto***. Java does not have ***goto***. There are, however, a few places where the ***goto*** is a useful, eg: the ***goto*** can be helpful when ***exiting from a deeply nested set of loops***. To handle such situations, Java defines an expanded form of ***break***.

☞ By using this form of ***break***, you can, for example, ***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***. It gives you the *benefits* of a ***goto*** without its *problems*.

☞ Further, you can **specify** precisely where execution will ***resume***, because this form of break works with a ***label***.

☞ The ***general form*** of the *labeled break*, `break label;`

☑ Typically, ***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***, but it does not need to be the ***immediately enclosing block***. This means that you can use a ***labeled break*** 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***.

☑ To name a block, put a ***label*** at the **start** of it. The block can be a *stand-alone* block, or a *statement that has a block as its target*. 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***. Doing so *causes execution to resume at the end of the labeled block.*

Example 1: Following program shows three nested blocks:

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

OUTPUT:

```
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.
```

- When ***i*** is ***1***, the first ***if*** succeeds, causing a **break** to the ***end of the block*** of code defined by label ***one***. This causes ***After block one.*** to print.
- When ***i*** is ***2***, the second ***if*** succeeds, causing ***control to be transferred to the end of the block*** labeled by ***two***. This causes the messages ***After block two.*** and ***After block one.*** to be printed, in that order.
- When ***i*** is ***3***, the third ***if*** succeeds, and ***control is transferred to the end of the block*** labeled by ***three***. Now, all three messages are displayed.

Example 2: This time, ***break*** is being used to ***jump*** outside of a series of ***nested for loops***. When the ***break*** statement in the ***inner loop*** is executed, ***program control*** jumps to the ***end of the block*** defined by the ***outer for loop***, which is labeled by done.

```
done: for(int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
             for(int k=0; k<10; k++) { System.out.println(k + " ");
                                 if(k == 5) break done;    // jump to done
                               }
             System.out.println("After k loop");           // won't execute
                               }
            System.out.println("After j loop");           // won't execute
                               }
System.out.println("After i loop");
```

OUTPUT:

```
0
1
2
3
4
5
After i loop
```

Example 3: Precisely ***where you put a label is very important***—especially when working with **loops**. For example:

```
                        // here, put label before for statement.
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); }
               }
```

OUTPUT:

```
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
```

- Both sets of **nested loops** are the same except for one point. In the first set, the **label** precedes the ***outer for loop***. In this case, when the ***break*** executes, it transfers control to the ***end of the entire for block***, skipping the rest of the outer loop's iterations.
- In the second set, the **label** precedes the ***outer for's opening curly brace***. Thus, when ***break stop2*** executes, control is transferred to the ***end of the outer for's block***, **causing the next iteration to occur**.

Example 4: We cannot break to any label that is not defined for an enclosing block. Eg: following is invalid & won't compile:

```
one:   for(int i=0; i<3; i++) { System.out.print("Pass " + i + ": "); }
       for(int j=0; j<100; j++) { if(j == 10) break one; // WRONG
                                  System.out.print(j + " "); }
```

- Since the loop labeled ***one*** does not enclose the ***break***, it is not possible to transfer control to that block.

Labeled continue: As with the ***break*** statement, ***continue*** may specify a label to describe which enclosing loop to continue. Eg:

```
outerloop:
for(int i=1; i < 10; i++) { System.out.print("\nOuter loop pass " + i + ", Inner loop: ");
                            for(int j = 1; j < 10; j++) { if(j == 5) continue outerloop;  // continue outer loop
                                                          System.out.print(j); }
                          }
```

OUTPUT

```
Outer loop pass 1, Inner loop: 1234
Outer loop pass 2, Inner loop: 1234
Outer loop pass 3, Inner loop: 1234
        ... ... ...
Outer loop pass 9, Inner loop: 1234
```

- As the output shows, when the ***continue*** executes, control passes to the **outer loop**, skipping the remainder of the **inner loop**.

NOTE

[1] ***Choosing appropriate loop:*** Use a ***for loop*** when performing a **known number of iterations**. Use the ***do-while*** when you need a loop that will always perform **at least one iteration**. The ***while*** is best used when the loop will repeat an **unknown number of times**.

[2] **Using *if-else-if* ladder:** Use an ***if-else-if*** ladder when the conditions controlling the selection process do not rely upon a single value.

```
if(x < 10)          // ...
else if(y != 0)     // ...
else if(!done)      // ...
```

This sequence cannot be recoded into a ***switch*** because all three conditions involve different variables—and differing types.

- Also, you will need to use an ***if-else-if*** ladder when testing ***floating-point*** values or other objects that are not of types valid for use in a ***switch expression***.