

Chapter 1 (Complete Reference By Schildth)

Introduction to Java

Java History

- Java is a general purpose, object-oriented programming language.
- Developed by Sun Microsystems of USA in 1991. (Currently owned by Oracle)
- Originally called **Oak** by James Gosling (Father of Java).
- Designed for development of software for consumer electronics devices like TV, VCR, toasters, and other such electronic machines.
- **Goal:** make the language simple, portable and highly reliable.
- Most Striking feature: *Platform – neutral language*.
- Java is the first programming language that is not tied to any particular hardware or OS.
- Programs developed in java can be executed anywhere on any system.
- Java – a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs.
- Nothing like this happened to the software industry before.
- Java is just a name and not an acronym.

Java Milestones :

- 1990→ Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
- 1991→ After exploring the possibility of most Object Oriented Programming Language C++, the team announced a new language named "Oak".
- 1992→ The team, known as a Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch sensitive screen.
- 1993→ The World Wide Web(WWW) appeared on the internet and transformed the text-based Internet into a Graphical-rich environment. The green Project team came up with the idea of developing Web Applets(tiny programs) using the new language that could run on all types of computers connected to Internet.
- 1994→ The team developed a web browser called "Hot Java" to locate and run applet programs on Internet. Hot Java demonstrated the power of the new language, thus making it instantly popular among the Internet users.
- 1995→ Oak was named "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announce to their support to Java.

1996→ Java established itself not only a leader for Internet Programming but also as a general-purpose, object oriented programming language. Java found its home.

The most striking feature of the language is that it is a platform-neutral language. Java is a first programming language that is not tied to any particular hardware or operating system.

Features of Java :

- **Compiled and Interpreted.**
- **Platform-Independent and Portable**
- **Object-Oriented**
- **Robust and Secure**
- **Distributed**
- **Familiar, Simple and Small**
- **Multithreaded and Interactive**
- **High Performance**
- **Dynamic and Extensible**

Java Buzzwords

There are 11 salient features of Java which makes java so popular. They are

1. Simple
2. Secure
3. Portable
4. Object – oriented
5. Robust
6. Multithreaded
7. Architecture – neutral
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

Simple

- Java was designed to be easy for professional programmer to learn and use effectively.
- It's simple and easy to learn if you already know the basic concepts of Object

Oriented Programming.

- C++ programmer can move to JAVA with very little effort to learn.
- Because Java inherits the C/C++ syntax and many of the Object Oriented features of C++, most programmers have little trouble in learning java.

Secure

- Java is designed for use on internet.
- It enables construction of virus-free, error free systems.
- The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Portable

- Java programs can be easily moved from one computer system to another, anywhere and anytime.
- Changes and upgrades in OS, processors and system resources will not force any changes in Java programs.
- This is the reason why java has become a popular language for programming on internet which interconnects different kinds of systems worldwide.

Object Oriented

- Java is true object oriented language.
- Almost —Everything is an Object|| paradigm.
- All program code and data reside within objects and classes.
- The object model in Java is simple and easy to extend.
- Java comes with an extensive set of classes, arranged in packages that can be used in our programs through inheritance.

Robust

- It provides many features that make the program execute reliably in variety of environments.
- Java is a strictly typed language. It checks code both at compile time and runtime.
- Java takes care of all memory management problems with garbage-collection.
- Java, with the help of exception handling captures all types of serious errors and eliminates any risk of crashing the system.

Multithreaded

- Multithreaded Programs handled multiple tasks simultaneously, which was helpful in creating interactive, networked programs.
- Java run-time system comes with tools that support multiprocess synchronization used to construct smoothly interactive systems.

Architecture Neutral

- Java language and Java Virtual Machine helped in achieving the goal of -write once; run anywhere, any time, forever. |
- Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.

Compiled and Interpreted

- Usually a computer language is either compiled or Interpreted.
- Java combines both this approach and makes it a two-stage system.
- **Compiled** : Java enables creation of a cross platform programs by compiling into an intermediate representation called Java Bytecode.
- **Interpreted** : Bytecode is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

High Performance

- Java performance is high because of the use of bytecode.
- The bytecode was used, so that it was easily translated into native machine code.

Distributed

- Java is designed for distributed environment of the Internet.
- Its used for creating applications on networks.
- Java applications can access remote objects on Internet as easily as they can do in local system.
- Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

Dynamic

- Java is capable of linking in new class libraries, methods, and objects.

- It can also link native methods (the functions written in other languages such as C and C++).

Java's Magic: The ByteCode

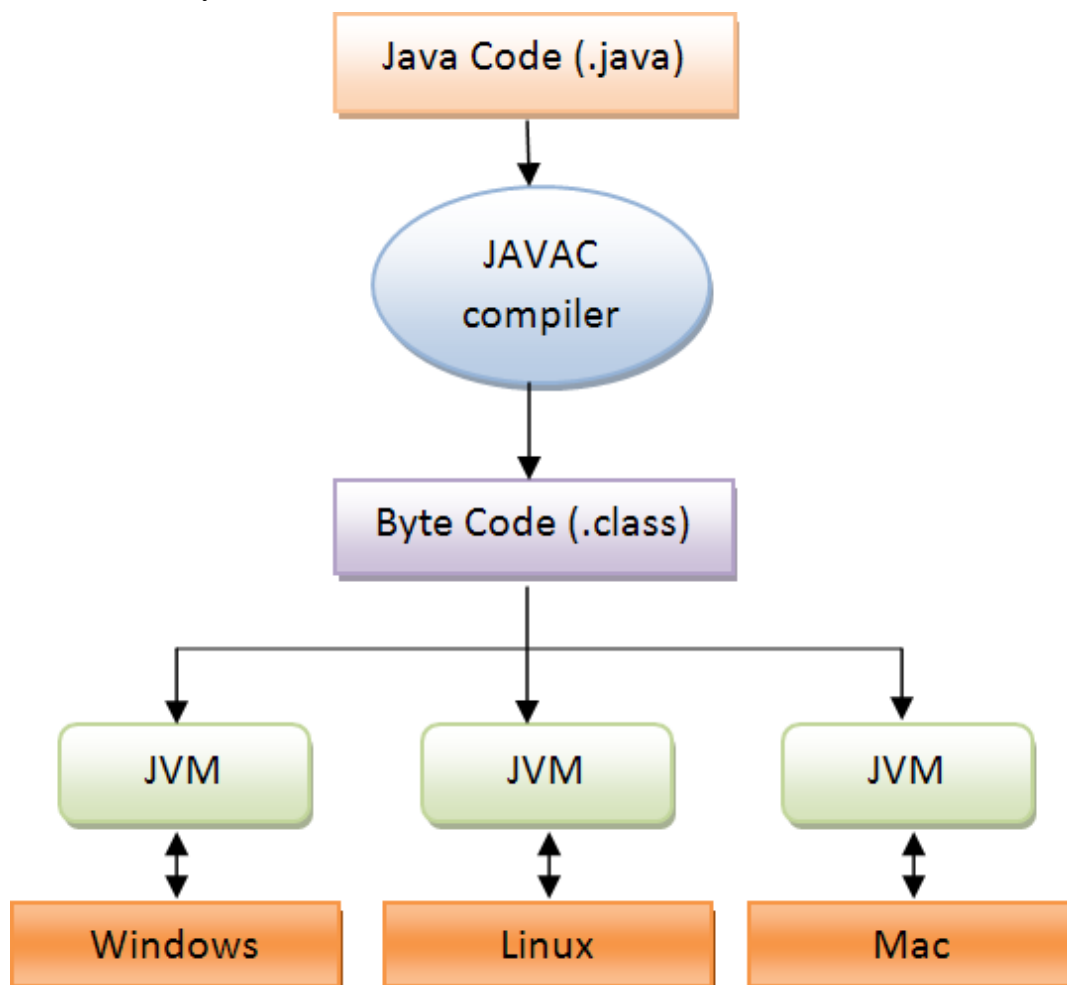
The key that allows Java to solve both the security and the portability problems is : Bytecode

“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)”.

The original JVM was designed as an interpreter for bytecode. This may come as a bit of a surprise. Many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web – based programs.

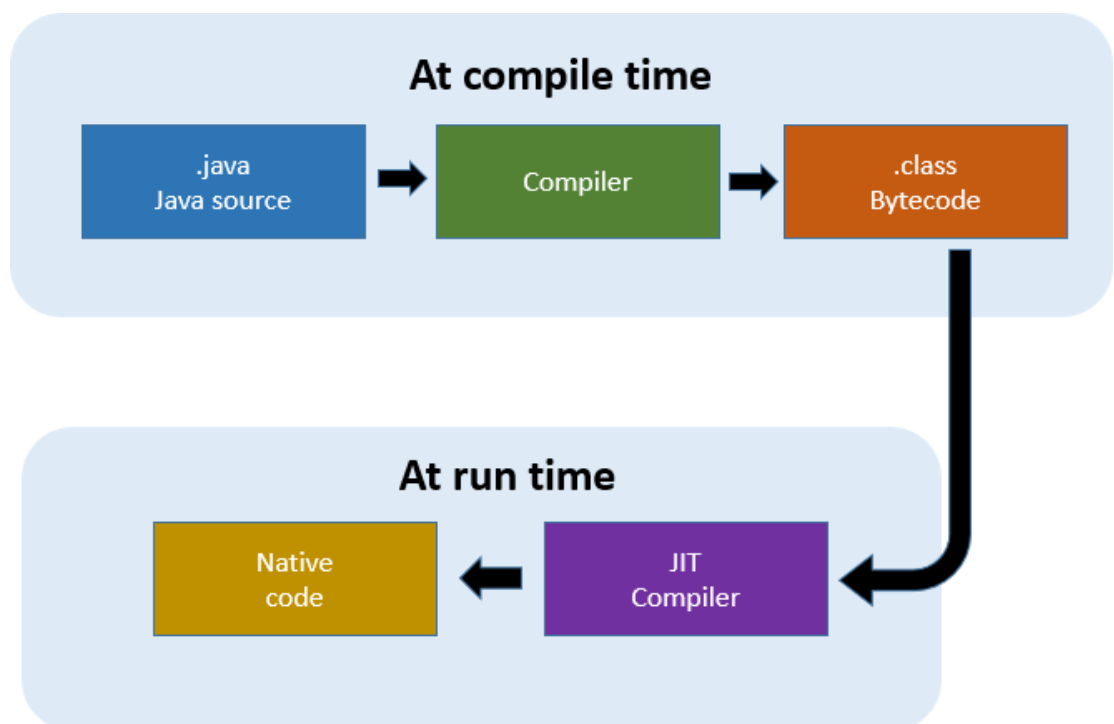
Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.

Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.



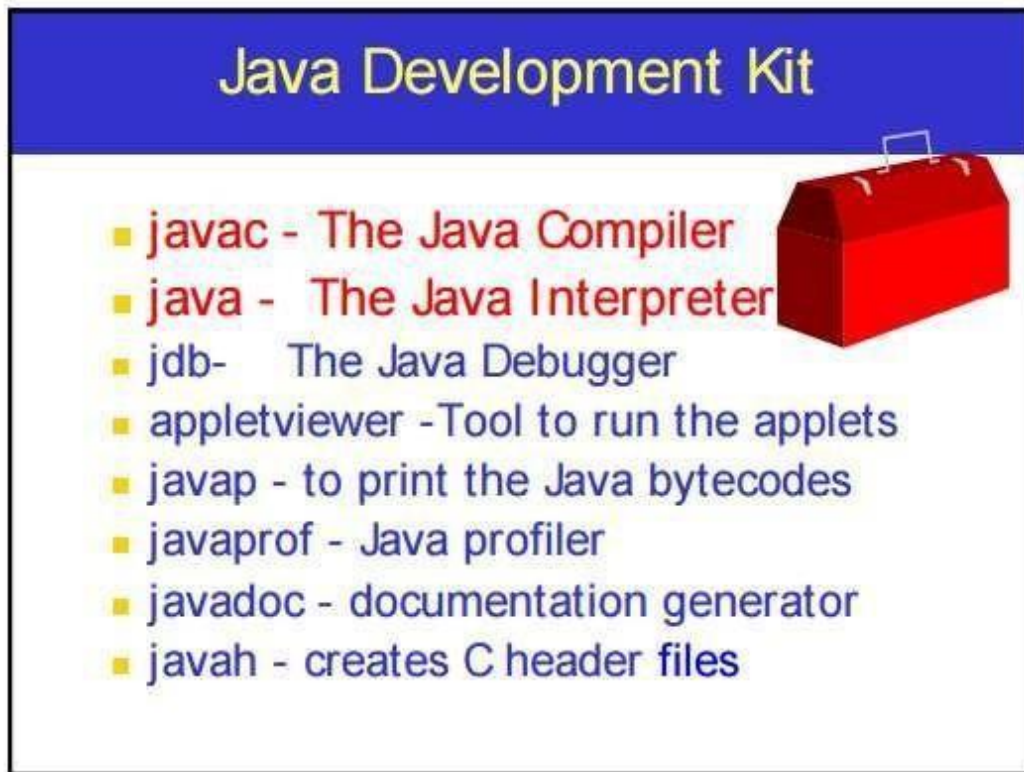
Object Oriented Concepts

- If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet.
- This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.
- In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it 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.
- Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance.
- For this reason, Sun began supplying its Hot Spot technology not long after Java's initial release.
- Hot Spot provides a 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.
- It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time.
- Instead, a JIT compiler compiles code as it is needed, during execution.
- The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode.



Java DevelopmentKit(JDK)

- To develop Java applications and applets as well as run them, the JDK is needed.
- To run Java applications and applets, simply download the JRE.



Chapter 2 (Complete Reference By Schildth)

An Overview of Java

Object-Oriented Programming

These are the two paradigms that govern how a program is constructed.

- The first way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps. The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success.
- To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*.

Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Abstract classes and Abstract methods:

- An abstract class is a class that is declared with abstract keyword.
- An abstract method is a method that is declared without an implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
- An abstract class can have parameterized constructors and default constructor is always present in an abstract class.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

The super keyword

The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.

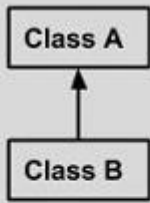
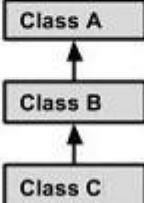
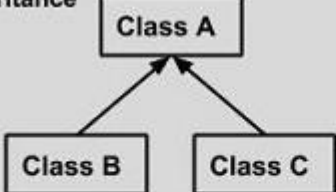
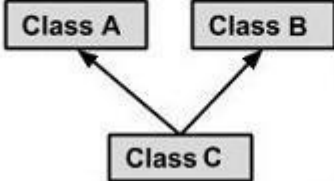
- It is used to differentiate the members of superclass from the members of subclass, if they have same names.
- It is used to invoke the superclass constructor from subclass.

Object Oriented Concepts

Types of Inheritance

On the basis of class, there can be three types of inheritance in java:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance and
- Multiple Inheritance

Single Inheritance  <pre>graph BT; B[Class B] --> A[Class A]</pre>	<pre>public class A { } public class B extends A { }</pre>
Multi Level Inheritance  <pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A]</pre>	<pre>public class A {} public class B extends A {.....} public class C extends B {.....}</pre>
Hierarchical Inheritance  <pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A</pre>	<pre>public class A {} public class B extends A {.....} public class C extends A {.....}</pre>
Multiple Inheritance  <pre>graph BT; C[Class C] --> A[Class A]; C --> B[Class B]</pre>	<pre>public class A {} public class B {.....} public class C extends A,B { } // Java does not support multiple Inheritance</pre>

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Let us look at an example.

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

A First Simple Java Program

Example.java

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

- Java source file may contain more than one class definitions.
- Name of the source file and the name of the class which holds main() must be same.
- Java source files are saved with **.java** extension.
- In Java all code must reside inside a class.
- Java is case-sensitive. (Name of file should be same case as that of class name)

Compiling a Java Program

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

```
C:\> javac Example.java
```

- The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program.
- The Java bytecode is the intermediate representation of your program that contains instructions the JVM will execute.
- Thus, the output of **javac** is not code that can be directly executed.
- When Java source code is compiled, each individual class has its own **.class** file.

Executing a Java Program

- To actually run the program, you must use the Java application launcher, called **java**.
- **java** is the interpreter of Java Programming Language
- To do so, pass the class name **Example** as a command-line argument.

```
C:\> java Example
```

- When the program is run, the following output is displayed.

This is a simple Java program.

Explanation of Program

- The program begins with

```
/*  
    This is a simple Java program.  
    Call this file "Example.java".  
*/
```

- This is a ***Comment***.
- Comments help to increase the readability of the program.
- Comments are ignored by the compiler.
- Comments include name of the program, title, author, company, etc.

Java supports **3 styles of comments**:

- Multiline Comment
- Single-line Comment
- Documentation Comment

1. Multiline Comment:

begins with `/*` and ends with `*/`. Can include multiple lines.

Example : `/* This is a simple Java program.`

`Call this file "Example.java"*/`

2. Single-line Comment:

which starts with `//` and ends at the end of the line.

Example: `// Your program begins with a call to main().`

3. Documentation Comment:

begins with `/**` and ends with `*/`. Includes `@author`, etc.

Example: `/** documentation */`

The next line in the code is:

```
class Example

{

//class body

}
```

The keyword **class** tells that new class is being defined. **Example** is the name of the class which should be valid identifier. { -indicates the beginning of class. } -indicates the end of class. Class in Java do not end with semi-colon.

Public static void main (String args[])

The **public** key word is an access specifier, which allows the programmer to control the visibility of class members. When a class member is public, then that member can be accessed outside the class. main() is public, since it must be called by code outside of its class when the program is started.

- All Java applications begin execution by calling **main()**.
- The keyword **static** allows **main()** to be called **without** having to instantiate a particular instance of the class.
- This is necessary since **main()** is called by the Java Virtual Machine before any objects are made.
- The keyword **void** simply tells the compiler that **main()** **does not return a value**.

Java program may have more than one class. Java compiler will compile classes that do not contain main() method. But, java has no way to run these classes. Therefore, java would report an error because it would be unable to find main() method. In main(), there is only one parameter, String args[]. String arg[] declares a parameter named args, which is an array of instances of the class String. Objects of type String to represent character strings. Args receive any command line arguments present when the program is executed.

Object Oriented Concepts

A complex program will have dozens of classes, only one of which will need to have a **main()** **method to get** things started. The next line of code is shown here. Notice that it occurs inside **main()**.

```
System.out.println ("This is a simple Java program.");
```

This line out puts the string –This is a simple Java program.”, followed by a new line on the screen.

➤ **println()** –displays a string and the cursor comes to the next line.

➤ **print()** -displays a string and the cursor remains in the same line.

➤ The line begins with **System.out**. Here, **System** is a class, **out** is a static object and **println()** is a method.

➤ **System:** It is the name of standard class that contains objects that encapsulate the standard **I/O** devices of your system.

➤ It is contained in the **package** `java.lang`.

➤ Since `java.lang` package is imported in every java program by default, therefore **java.lang** package is the only package in Java API which does not require an import declaration.

➤ **out:** The object `out` represents output stream (i.e. Command window) and is the static data member of the class **System**.

So note here **System.out** (System -Class & out-static object i.e. why its simply referred by class name and we need not to create any object).

➤ **println:** The `println()` is **method** of `out` object that takes the text string as an argument and displays it to the standard output i.e. on monitor screen.

A Second Short Program

- *variable is a named memory location that may be assigned a value by your program.*
- The value of a variable may be changed during the execution of the program.

```
/*
Here is another short example.
Call this file "Example2.java".
*/
class Example2
{
public static void main(String args[])
{
    Int num;           // this declares a variable called num
    num = 100;         // this assigns num the value 100
    System.out.println("This is num: " + num);
    num = num * 2;
    System.out.print("The value of num * 2 is ");
    System.out.println(num);
}
}
```

Output

```
This is num: 100
The value of num * 2 is 200
```

```
int num; // this declares a variable called num
```

- This line declares an integer variable called **num**. **Java (like most other languages) requires** that variables be declared before they are used.

Following is the general form of a variable declaration:

type var-name;

- Here, *type specifies the type of variable being declared, and var-name is the name of the variable.*
- If you want to declare more than one variable of the specified type, you may use a comma separated list of variable names. Java defines several data types, including integer, character, and floating-point. The keyword **int** specifies an integer type.
- In the program, the line

```
num = 100;           // this assigns num the value 100
```

- Assigns to **num** the value **100**. In Java, the assignment operator is a single equal sign. The next line of code outputs the value of **num** preceded by the string "This is num:".

```
System.out.println("This is num: " + num);
```

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output.

- This approach can be generalized. Using the + operator, you can join together as many items as you want within a single `println()` statement.
- First, the built-in method `print()` is used to display the string -The value of `num * 2` is `l`. This string is *not followed by a newline*.

Two Control Statements

The if Statement:

The Java if statement works much like the IF statement in any other language. Further, it is syntactically identical to the if statements in C, C++, and C#.

Its simplest form is shown here:

`if(condition) statement;`

Here, condition is a Boolean expression. If condition is true, then the statement is executed.

- *If condition is false, then the statement is bypassed. Here is an example:*

If (num < 100)

```
System.out.println("num is less than 100");
```

- *In this case, if num contains a value that is less than 100, the conditional expression is true, and `println()` will execute. If num contains a value greater than or equal to 100, then the `println()` method is bypassed.*

```
/* Demonstrate the if.  
Call this file "IfSample.java". */
```

```
class IfSample{  
    public static void main(String args[])  
    {  
        int x, y;  
        x = 10;  
        y = 20;  
        if(x < y) System.out.println("x is less than y");  
        x = x * 2;
```

```
        if(x == y) System.out.println("x now equal to y");
        x = x * 2;
        if(x > y) System.out.println("x now greater than y");
        if(x == y) System.out.println("you won't see this");
    }
}
```

Output:

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

The for Loop:

Loop statements are an important part of nearly any programming language. Java is no exception.

- Perhaps the most versatile is the for loop. The simplest form of the for loop is shown here:

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

- In its most common form, the *initialization portion of the loop sets a loop control variable* to an initial value. The *condition is a Boolean expression that tests the loop control variable*. If the outcome of that test is true, the for loop continues to iterate.
- If it is false, the loop terminates. The *iteration expression determines how the loop control variable is changed* each time the loop iterates.

```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
class ForTest{
public static void main(String args[]) {
    int x;
    for(x = 0; x<10; x = x+1)
        System.out.println("This is x: " + x);
    }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

In this example, `x` is the loop control variable. It is initialized to zero in the initialization portion of the `for`.

- At the start of each iteration (including the first one), the conditional test `x < 10` is performed. If the outcome of this test is true, the `println()` statement is executed, and then the iteration portion of the loop is executed. This process continues until the conditional test is false.
- As a point of interest, in professionally written Java programs you will almost never see the iteration portion of the loop written as shown in the preceding program. That is, you will seldom see statements like this:

```
x = x + 1;
```

- The reason is that Java includes a special increment operator which performs this operation more efficiently. The increment operator is `++`. The increment operator increases its operand by one. By use of the increment operator, the preceding statement can be written like this:

```
x++;
```

- Thus, the `for` in the preceding program will usually be written like this:
`for(x = 0; x < 10; x++)`

Using Blocks of Code

- Java allows two or more statements to be grouped into *blocks of code*, also called *code blocks*. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.


```
if(x < y) { // begin a block
    x = y;
    y = 0;
} // end of block
```

- Here, if **x is less than y, then both statements inside the block will be executed. Thus, the two** statements inside the block form a logical unit, and one statement cannot execute without the other also executing.
- The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

```
/*
Demonstrate a block of code. Call this file "BlockTest.java"
*/
class BlockTest
{
    public static void main(String args[ ])
    {
        int x, y;
        y = 20;
        // the target of this loop is a block
        for(x = 0; x < 10; x++)
        {
            System.out.println("This is x: " + x);
            System.out.println("This is y: " + y);
            y = y - 2;
        }
    }
}
```

The output generated by this program is shown here:

```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
```

```
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords (These are called as Lexical Issues).

Whitespace: Java is a free-form language. This means that you do not need to follow any special indentation rules.

For instance, the **Example program could have been written all on one line or in any** other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator.

In Java, whitespace is a space, tab, or newline.

Identifiers: Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.

They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE is a different identifier than Value.**

Literals: A constant value in Java is created by using a *literal representation of it*.

100	98.6	'X'	"This is a test"
-----	------	-----	------------------

Comments: As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*.

Separators: In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon.

Object Oriented Concepts

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

The Java Keywords

There are 50 keywords currently defined in the Java language

These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.

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

TABLE 2-1 Java Keywords

Chapter 3 (Complete Reference By Schildth)

Data Types, Variables, and Arrays

Java Is a Strongly Typed Language: It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact.

- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types

Java defines eight *primitive types of data*:

byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as *simple types*

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

1. Integers:

- Java defines four integer types: **byte**, **short**, **int**, and **long**.
- All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.
- Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary.
- Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the sign of an integer value.
- The *width of an integer type should not be thought of as the amount of storage it consumes*, but rather as the *behavior it defines for variables and expressions of that type*.

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

byte: The smallest integer type is byte. This is a signed 8-bit type that has a range from −128 to 127.

Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the byte keyword.

Ex: byte b, c;

short: short is a signed 16-bit type. It has a range from −32,768 to 32,767. It is probably the least-used Java type. Here are some examples of short variable declarations:

short s;

short t;

int: The most commonly used integer type is int. It is a signed 32-bit type that has a range from −2,147,483,648 to 2,147,483,647.

Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed..... this may not be the case. The reason is that when byte and short values are used in an expression they are *promoted to int when the expression is evaluated*.

Object Oriented Concepts

Therefore, `int` is often the best choice when an integer is needed.

long : `long` is a signed 64-bit type and is useful for those occasions where an `int` type is not large enough to hold the desired value. The range of a `long` is quite large. This makes it useful when big, whole numbers are needed.

```
// Compute distance light travels using long variables.
class Light {
    public static void main(String args[])
    {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // approximate speed of light in miles per second
        lightspeed = 186000;
        days = 1000;                // specify number of days here
        seconds = days * 24 * 60 * 60;    // convert to seconds
        distance = lightspeed * seconds;    // compute distance
        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

Floating-Point Types

Floating-point numbers, also known as *real numbers*, are used when evaluating expressions that require fractional precision.

For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators.

There are two kinds of floating-point types, `float` and `double`

Name	Width in Bits	Approximate Range
<code>double</code>	64	4.9e–324 to 1.8e+308
<code>float</code>	32	1.4e–045 to 3.4e+038

float: The type `float` specifies a *single-precision value that uses 32 bits of storage*.

Object Oriented Concepts

Single precision is faster on some process or sand takes half as much space as double precision, but will become imprecise when the values are either very large or very small.

Here are some examples of **float variable declarations**:

```
float hightemp, lowtemp;
```

double

- Double precision, as denoted by the **double keyword**, uses **64 bits to store a value**.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. When you need to maintain accuracy then double is used.
- Here is a short program that uses **double variables to compute the area of a circle**:

```
// Compute the area of a circle.
class Area
{
    public static void main(String args[ ])
    {
        double pi, r, a;
        r = 10.8;           // radius of circle
        pi = 3.1416;        // pi, approximately
        a = pi * r * r;      // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

Characters

- In Java, the data type used to store characters is **char**. **However, C/C++ programmers beware: char in Java is not the same as char in C or C++.**
- **In C/C++, char is 8 bits wide. This is *not the* case in Java.**
- Instead, Java uses Unicode to represent characters. *Unicode defines a fully*
- *international character set that can represent all of the characters found in all human languages.*
- It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- For this purpose, it requires 16 bits. Thus, in Java **char is a 16-bit type**.
- There are no negative **chars**.

- The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.
- Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters

Example 1

```
// Demonstrate char data type.
class CharDemo
{
    public static void main(String args[ ])
    {
        char ch1, ch2;
        ch1 = 88; // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

output:

ch1 and ch2: X Y

- Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X.
- Although char is designed to hold Unicode characters, it can also be thought of as an integer type on which you can perform arithmetic operations.

Example 2

```
// char variables behave like integers.
class CharDemo2
{
    public static void main(String args[ ])
    {
        char ch1;
        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

output:

ch1 contains X
ch1 is now Y

Object Oriented Concepts

In the program, `ch1` is first given the value `X`. Next, `ch1` is incremented. This results in `ch1` containing `Y`, the next character in the ASCII (and Unicode) sequence.

Booleans

- Java has a primitive type, called `Boolean`, for logical values. It can have only one of two possible values, `true` or `false`.
- This is the type returned by all relational operators, as in the case of `a < b`.
- `Boolean` is also the type required by the conditional expressions that govern the control statements such as `if` and `for`.

```
// Demonstrate Boolean values.
class BoolTest
{
    public static void main(String args[ ])
    {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if (b)
            System.out.println("This is executed.");
        b = false;
        if (b)
            System.out.println("This is not executed.");
        // outcome of a relational operator is a Boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

There are three interesting things to notice about this program.

- First, as you can see, when a `Boolean` value is output by `println()`, `-true` or `-false` is displayed.
- Second, the value of a `Boolean` variable is sufficient, by itself, to control the `if` statement. There is no need to write an `if` statement like this:
`if(b == true) ...`
- Third, the outcome of a relational operator, such as `<`, is a `Boolean` value.

This is why the expression `10 > 9` displays the value `-true`.

Literals:

- 1) Integer Literals
- 2) Floating point Literals
- 3) Boolean Literals
- 4) Character Literals
- 5) String Literals

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable : In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

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

The *type* is one of Java's atomic types, or the name of a class or interface.

- The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value.
- Here are several examples of variable declarations of various types. Note that some include an initialization.

```
inta, b, c; // declares three ints, a, b, and c.
intd = 3, e, f = 5; // declares three more ints, initializing d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';      // the variable x has the value 'x'.
```

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
class DynInit
{
    public static void main(String args[ ])
    {
        double a = 3.0, b = 4.0;
        // c is dynamically initialized
    }
}
```

```
        double c = Math.sqrt(a * a + b * b);  
        System.out.println("Hypotenuse is " + c);  
    }  
}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse.

The program uses another of Java's built-in methods, `sqrt()`, which is a member of the `Math` class, to compute the square root of its argument.

The Scope and Lifetime of Variables

A block defines a *scope*. A block is begun with an opening curly brace and ended by a closing curly brace. –Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.¶

Many other computer languages define two general categories of scopes:

global and local

But java has two major scopes

- 1) Scope Defined by the class
- 2) Scope Defined by the methods

However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.

Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

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

```
// Demonstrate block scope.
class Scope
{
    public static void main(String args[])
    {
        int x;                // known to all code within main
        x = 10;
        if(x == 10)
        {
            // start new scope
            int y = 20;        // known only to this block x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100;            // Error! y not known here
        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

As the comments indicate, the variable x is declared at the start of main()s scope and is accessible to all subsequent code within main().

Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block.

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.

```
// This fragment is wrong!
count = 100; // oops! Cannot use count before it is declared!
int count;
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.


```
// Demonstrate lifetime of a variable.
class LifeTime
{
    public static void main(String args[ ])
    {
        intx;
        for(x = 0; x < 3; x++)
        {
            inty = -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
```

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

```
// This program will not compile
class ScopeErr
{
    public static void main(String args[])
    {
        intbar = 1;
        {
            intbar = 2;          // creates a new scope
                                // Compile-time error –bar already defined!
        }
    }
}
```

Type Conversion and Casting

If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable.

However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte.

Object Oriented Concepts

Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an *explicit conversion* between incompatible types.

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

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

When these two conditions are met, a *widening conversion takes place*. For example, the `int` type is always large enough to hold all valid `byte` values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char or boolean**. Also, `char` and `Boolean` are not compatible with each other.

Casting Incompatible Types : what if you want to assign an `int` value to a `byte` variable? This conversion will not be performed automatically, because a `byte` is smaller than an `int`.

CASTING WILL CONVERT LARGER DATA TYPE INTO SMALLER DATA TYPE.

This kind of conversion is sometimes called a *narrowing conversion*.

- To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.
It has this general form: *(target-type) value*

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

For example, the following fragment casts an **int to a byte**.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation**.

- integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example,

If the value 1.23 is assigned to an integer, the resulting value will simply be 1.

The 0.23 will have been truncated.

Example :

```
// Demonstrate casts.
class Conversion
{
    public static void main(String args[ ])
    {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nConversion of int to byte.");

        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");

        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");

        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

Output

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.

Automatic Type Promotion in Expressions

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

```
byte a = 40;
byte b = 50;
```

```
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term $a * b$ easily exceeds the range of either of its byte operands.

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

As useful as the automatic promotions are, they can cause confusing compile-time errors.

```
byte b = 50;
b = b * 2;           // Error! Cannot assign an int to a byte!
```

The code is attempting to store $50 * 2$, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int.

Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

- First, all byte, short, and char values are promoted to int.
- Then, if one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

```
class Promote
{
    public static void main(String args[ ])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
```

```
double d = .1234;
double result = (f * b) + (i/ c) -(d * s);

System.out.println((f * b) + " + " + (i/ c) + " -" + (d * s));
System.out.println("result = " + result);
    }
}
```

In the first sub expression, $f*b$, b is promoted to a float and the result of the sub expression is float.

Next, in the sub expression i/c , c is promoted to int, and the result is of type int.

Then, in $d*s$, the value of s is promoted to double, and the type of the sub expression is double.

Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.

One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-name[ ];
```

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.

```
int month_days[ ];
```

Although this declaration establishes the fact that **month_days** is an array variable, **no** array actually exists. In fact, the value of **month_days** is set to null, which represents an array with no value.

To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:

```
array-var= new type[size];
```

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array.

That is, to use `new` to allocate an array, you must specify the type and number of elements to allocate.

The elements in the array allocated by `new` will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to `month_days`.

```
month_days= new int[12];
```

After this statement executes, `month_days` will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Obtaining an array is a two-step process.

- First, you must declare a variable of the desired array type.
- Second, you must allocate the memory that will hold the array, using **`new`**, and assign it to the array variable.

Thus, in Java all arrays are dynamically allocated.

- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.
- All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **`month_days`**.

```
month_days[1] = 28;
```

The next line displays the value stored at index 3.

```
System.out.println(month_days[3]);
```

Example :

```
// Demonstrate a one-dimensional array.
class Array
{
    public static void main(String args[ ])
    {
        int month_days[ ];
        month_days= new int[12];
        month_days[0] = 31;           // For January
        month_days[1] = 28;           // For February
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
```

```
        month_days[10] = 30;
        month_days[11] = 31;
        System.out.println("April has " + month_days[3] + " days.");
    }
}
```

- When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3] or 30**.
- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int month_days[] = new int[12];
```

- This is the way that you will normally see it done in professionally written Java programs
- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces.
- The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use **new**.

```
// An improved version of the previous program.
class AutoArray
{
    public static void main(String args[ ])
    {
        int month_days[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        System.out.println("April has " + month_days[3] + " days.");
    }
}
```

If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Ex: if you try to access month_days[13] You will get an error

Multidimensional Arrays

In Java, *multi-dimensional arrays are actually arrays of arrays*. However, as you will see, there are a couple of subtle differences. To declare a multi-dimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a two dimensional array variable called **two D**.

```
int twoD[ ][ ] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. **Internally this matrix is implemented as an array of array so *int***.

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

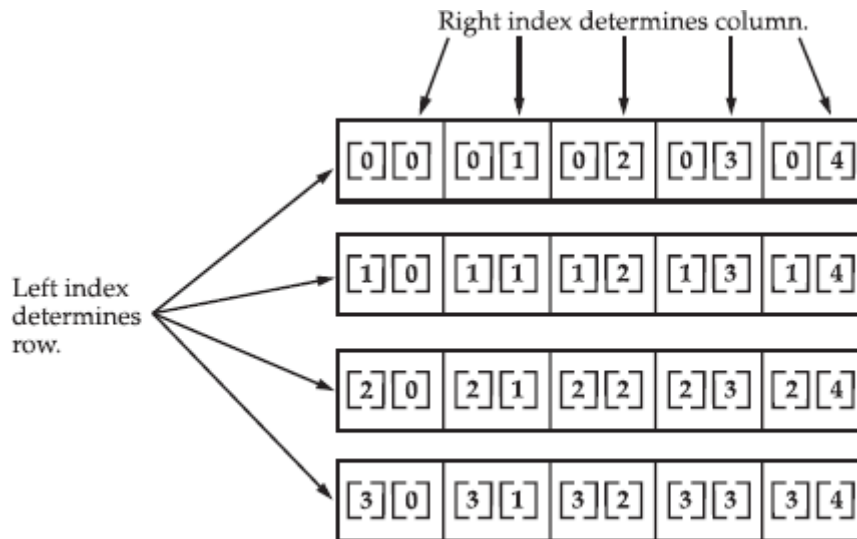
Example :

```
// Demonstrate a two-dimensional array.
class TwoDArray
{
    public static void main(String args[ ])
    {
        inttwoD[ ][ ]= new int[4][5];
        inti, j, k = 0;
        for(i=0; i<4; i++)
        for(j=0; j<5; j++)
        {
            twoD[i][j] = k;
            k++;
        }
        for(i=0; i<4; i++)
        {
            for(j=0; j<5; j++)
            System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

Output

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately.



Given: `int twoD [] [] = new int [4] [5];`

For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD[ ][ ] = new int[4][ ];
```

```
twoD[0] = new int[5];
```

```
twoD[1] = new int[5];
```

```
twoD[2] = new int[5];
```

```
twoD[3] = new int[5];
```

- While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others.
- For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.
- As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.

```
class TwoDAgain
```

```
{
```

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

```
    {
```

```
        inttwoD[ ][ ] = new int[4][ ];
```

```
        twoD[0] = new int[1];
```

```
        twoD[1] = new int[2];
```

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

```
        twoD[3] = new int[4];
```

```
    inti, j, k = 0;
    for(i=0; i<4; i++)
    for(j=0; j<i+1; j++)
    {
        twoD[i][j] = k;
        k++;
    }
    for(i=0; i<4; i++)
    {
        for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
        System.out.println( );
    }
}
```

output:

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

- The array created by this program looks like this:
- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initialize within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.

```
// Initialize a two-dimensional array.
class Matrix
{
    public static void main(String args[])
    {
        double m[ ][ ] = {
            { 0*0, 1*0, 2*0, 3*0 },
            { 0*1, 1*1, 2*1, 3*1 },
            { 0*2, 1*2, 2*2, 3*2 },
            { 0*3, 1*3, 2*3, 3*3 }
        };

        inti, j;
        for(i=0; i<4; i++)
        {
            for(j=0; j<4; j++)
            System.out.print(m[i][j] + " ");
            System.out.println();
        }
    }
}
```

```
    }  
  }  
}
```

OUTPUT

```
0.0 0.0 0.0 0.0  
0.0 1.0 2.0 3.0  
0.0 2.0 4.0 6.0  
0.0 3.0 6.0 9.0
```

Let's look at one more example that uses a multidimensional array. The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes.

```
// Demonstrate a three-dimensional array.  
class ThreeDMatrix  
{  
    public static void main(String args[ ])   
    {  
        int threeD[ ][ ][ ] = new int[3][4][5];  
        int i, j, k;  
        for(i=0; i<3; i++)  
            for(j=0; j<4; j++)  
                for(k=0; k<5; k++)  
                    threeD[i][j][k] = i* j * k;  
        for(i=0; i<3; i++) {  
            for(j=0; j<4; j++) {  
                for(k=0; k<5; k++){  
                    System.out.print(threeD[i][j][k] + " ");  
                    System.out.println();  
                }  
                System.out.println();  
            }  
        }  
    }  
}
```

OUTPUT

```
0 0 0 0 0  
0 0 0 0 0  
0 0 0 0 0  
0 0 0 0 0  
0 0 0 0 0  
0 1 2 3 4  
0 2 4 6 8  
0 3 6 9 12  
0 0 0 0 0  
0 2 4 6 8  
0 4 8 12 16  
0 6 12 18 24
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

```
intal[ ] = new int[3];
```

```
int[ ] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[ ][ ] = new char[3][4];
```

```
char[ ][ ] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[ ] nums, nums2, nums3; // create three arrays
```

creates three array variables of type **int**. **It is the same as writing**

```
int nums[ ], nums2[ ], nums3[ ]; // create three arrays
```

Operators

Arithmetic Operators

Operator	Result
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

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

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types.

```
// Demonstrate the basic arithmetic operators.
class BasicMath
{
    public static void main(String args[])
    {
        // arithmetic using integers
        System.out.println("Integer Arithmetic");
        inta = 1 + 1;
        intb = a * 3;
        intc = b / 4;
        intd = c -a;
        inte = -d;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);

        // arithmetic using doubles
        System.out.println("\nFloatingPoint Arithmetic");
        double da= 1 + 1;
        double db = da* 3;
        double dc = db / 4;
        double dd= dc -a;
        double de = -dd;

        System.out.println("da= " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
        System.out.println("dd= " + dd);
        System.out.println("de = " + de);
    }
}
```

The Modulus Operator: The modulus operator, %, returns the remainder of a division operation.

```
// Demonstrate the % operator.
class Modulus
{
    public static void main(String args[ ])
    {
```

```
        intx = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod10 = " + y % 10);
    }
}
```

OUTPUT

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators: Java provides special operators that can be used to combine an arithmetic operation with an assignment.

```
a = a + 4;
```

In Java, you can rewrite this statement as shown here:

```
a += 4;
```

// Demonstrate the % operator.

```
class Modulus
```

```
{
    public static void main(String args[ ])
    {
        intx = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod10 = " + y % 10);
    }
}
```

OUTPUT

```
x mod 10 = 2
y mod 10 = 2.25
```

This version uses the += *compound assignment operator*. *Both statements perform the same action: they increase the value of a by 4.*

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

The compound assignment operators provide two benefits.

First, they save you a bit of typing, because they are –shorthand for their equivalent long forms.

Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

```
// Demonstrate several assignment operators.
class OpEquals
{
    public static void main(String args[ ]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Increment and Decrement

The ++ and the —are Java's increment and decrement operators.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement: `x = x - 1;`

is equivalent to `x--;`

These operators are unique in that they can appear both in *postfix form*, where they follow the operand as just shown, and *prefix form*

```
x = 42; y = ++x;
```

In this case, **y is set to 43** because the increment occurs *before x is assigned* to y. Thus, the line **y = ++x; is the equivalent of these two statements:**

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of **x is obtained before the increment operator is executed**, so the value of y is **42**.

Of course, in both cases **x is set to 43**.

Here, the line y = x++; is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

// Demonstrate ++.

```
class IncDec{
    public static void main(String args[ ]) {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b;
        d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

Output

```
a = 2
b = 3
c = 4
d = 1
```


The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

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

All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte value for 42 in binary is 00101010**.

All of the integer types (except **char**) are **signed integers**. This means that they can represent negative values as well as positive ones.

Java uses an encoding known as *two's complement*, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.

For example, **−42** is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or **−42**.

The Bitwise Logical Operators

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

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT Also called the *bitwise complement*, the *unary NOT operator*, \sim , *inverts all of the bits of its operand*.

For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

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

$$\begin{array}{r} 00101010 \quad 42 \\ \& \quad 00001111 \quad 15 \\ \hline 00001010 \quad 10 \end{array}$$

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

$$\begin{array}{r} 00101010 \quad 42 \\ | \quad 00001111 \quad 15 \\ \hline 00101111 \quad 47 \end{array}$$

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

$$\begin{array}{r} 00101010 \quad 42 \\ \wedge \quad 00001111 \quad 15 \\ \hline 00100101 \quad 37 \end{array}$$

The Left Shift (\ll)

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

$$value \ll num$$

Here, *num* specifies the number of positions to left-shift the value in *value*. That is, the \ll moves all of the bits in the specified value to the left by the number of bit positions specified by *num*. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

Java's automatic type promotions produce unexpected results when you are shifting **byte and short values**. As you know, **byte and short values are promoted to int when an expression is evaluated**. Furthermore, the result of such an expression is also an **int**.

Right Shift(>>)

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

value >> num

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

The following code fragment shifts the value 32 to the right by two positions, resulting in **a being set to 8**:

```
inta = 32;
a = a >> 2; // a now contains 8
inta = 35;
a = a >> 2; // a still contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011          35
>> 2
00001000          8
```

Each time you shift a value to the right, it divides that value by two—and discards any remainder.

The Unsigned Right Shift

As you have just seen, the **>> operator automatically fills the high-order bit with its previous contents** each time a shift occurs.

This preserves the sign of the value. However, sometimes this is undesirable.

For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place.

In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift*.

To accomplish this, you will use Java's unsigned, shift-right operator, `>>>`, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the `>>>`. **Here, a is set to -1, which sets all 32 bits to 1 in binary.**

This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a to 255**.

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

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

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

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

Remember, smaller values are automatically promoted to **int** expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8- or 16-bit value.

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combine the assignment with the bitwise operation.

For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:

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

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a OR b**, are equivalent:

```
a = a | b;
a |= b;
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other.

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

The outcome of these operations is a **booleanvalue**.

```
int a = 4;
```

```
int b = 1;
```

```
boolean c = a < b;
```

In this case, the result of **a<b (which is false)** is stored in **c**.

Boolean Logical Operators

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

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

The logical Boolean operators, **&**, **|**, and **^**, operate on **Boolean values in the same way** that they operate on the bits of an integer.

!true == false and !false == true.

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

Short-Circuit Logical Operators (V.V.IMP)

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit logical operators*.

As you can see from the preceding table, the OR operator results in **true when A is true, no matter what B is**. Similarly, the AND operator results in **false when A is false, no matter what B is**.

If you use the **||** and **&&** forms, rather than the **|** and **&** forms of these operators, Java will **not bother to evaluate the right-hand operand** when the outcome of the expression can be determined by the left operand alone.

This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom!= 0 && num / denom> 10)
```

Since the short-circuit form of AND (**&&**) is used, **there is no risk of causing a run-time exception when *denomis zero***.

If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denomis zero**.

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

For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

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

The Assignment Operator

The *assignment operator is the single equal sign, =*. **The assignment operator works in Java** much as it does in any other computer language.

It has this general form: `var= expression;`

Here, the type of *var* must be compatible with the type of *expression*.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you 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
```

This fragment sets the variables *x*, *y*, and *z* to 100 using a single statement. This works because the `=` is an operator that yields the value of the right-hand expression.

Thus, the value of `z = 100` is 100, which is then assigned to *y*, which in turn is assigned to *x*. Using a *-chain of assignment* is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary (three-way) operator that can replace certain types of if-then-else statements*.

This operator is the `?`. It can seem somewhat confusing at first, but the `?` can be used very effectively once mastered. The `?` has this general form:

expression1 ? expression2 : expression3

Here, *expression1* can be any expression that evaluates to a boolean value. If *expression1* is true, then *expression2* is evaluated; otherwise, *expression3* is evaluated.

The result of the `?` operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be void.

Here is an example of the way that the `?` is employed:

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

Object Oriented Concepts

When Java evaluates this assignment expression, it first looks at the expression to the *left of the question mark* ? If denom equals zero, then the expression *between the question mark and the colon* is evaluated and used as the value of the entire ? expression.

If denom does not equal zero, then the expression *after the colon* is evaluated and used for the *value of the entire* ? expression.

The result produced by the ? operator is then assigned to ratio.

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

`a >> b + 3`

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

`a >> (b + 3)`

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

`(a >> b) + 3`

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression.

For anyone reading your code, a complicated expression can be difficult to understand.

`a | 4 + c >> b & 7`

`(a | (((4 + c) >> b) & 7))`

Example

```
class BoolLogic
{
    public static void main(String args[ ])
    {
        Boolean a = true;
        Boolean b = false;
        Boolean c = a | b;
        Boolean d = a & b;
        Boolean e = a ^ b;
        Boolean f = (!a & b) | (a & !b);
        Boolean g = !a;
```



```
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" a|b= " + c);
        System.out.println(" a&b= " + d);
        System.out.println(" a^b= " + e);
        System.out.println(" !a&b|a&!b = " + f);
        System.out.println(" !a = " + g);
    }
}
```

OUTPUT

```
a = true
b = false
a|b= true
a&b= false
a^b= true
a&b|a&!b = true
!a = false
```

Control Statements

Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.*

Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.

Java's Selection Statements:

- if
- Nested if
- The if-else-if Ladder
- switch
- Nested switch Statements

Java supports two selection statements: **if and switch**.

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

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

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean value**. The *else* clause is optional.

The **if** works like this: **If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.**

```
inta, b;
// ...
if(a < b) a = 0;
else b = 0;
```

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

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single Boolean variable, as shown in this code fragment:

```
booleandataAvailable;
// ...
if (dataAvailable)
    ProcessData();
else
    waitForMoreData();
```

If you want to include more statements, you'll need to create a block, as in this fragment:

```
intbytesAvailable;
// ...
if (bytesAvailable > 0)
{
    ProcessData();
    bytesAvailable -= n;
}
else
    waitForMoreData();
```

Here, both statements within the **if block** will execute if **bytesAvailable** is greater than zero.

Nested if

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

When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

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

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else).

Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

The if-else-if Ladder

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

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

The if statements are executed from the top down.

As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is by passed. If none of the conditions is true, then the final else statement will be executed.

The final else acts as a default condition; that is, if all other conditional tests fail, then the Last else statement is performed.

If there is no final else and all other conditions are false, then no action will take place.

```
// Demonstrate if-else-if statements.
class IfElse
{
    public static void main(String args[ ])
    {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Output

April is in the Spring.

Switch

The switch statement is Java's multiway branch statement.

- It provides an easy way to dispatch execution to different parts of code based on the value of an expression.
- It provides a better alternative than a large series of if-else-if statements.
- Here is the general form of a switch statement:

```
switch (expression) {
    case value1:
        // statement sequence
        break;
```

```
case value2:
    // statement sequence
    break;
...
case valueN:
    // statement sequence
    break;
default:
    // default statement sequence
}
```

- ❑ The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression.
- ❑ Each case value must be a unique literal (that is, it must be a constant, not a variable).
- ❑ Duplicate case values are not allowed.
- ❑ The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed.
- ❑ If none of the constants matches the value of the expression, then the default statement is executed.
- ❑ However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

Nested switch Statements

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

```
switch(count)
{
```

case 1:

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

case 2: // ...

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

In summary, there are three important features of the **switch** statement to note:

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

Iteration Statements

- Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*.
- a loop repeatedly executes the same set of instructions until a termination condition is met.

while

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

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

- The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true.

When *condition* becomes false, control passes to the next line of code immediately following the loop.

```
class While
{
    public static void main(String args[])
    {
        int n =
        5;
        while(
        n > 0)
        {
            System.out.println("tick "
            + n); n--;
        }
    }
}
```

When you run this program, it will -tick five times:
tick 5
tick 4
tick 3
tick 2
tick 1

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

For example, in the following fragment, the call to **println()** is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null Statement* is syntactically valid in Java.

do-while

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

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

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
class DoWhile
{
    public static void main(String args[])
    {
        int n = 5;
    do {
        System.out.println("tick " + n); n--;
    } while(n > 0);
    }
}

class Menu
{
    public static void main(String args[]) throws java.io.IOException
    {
        char choice;
        do
        {
            System.out.println("Help on:"); System.out.println(" 1. good");
            System.out.println(" 2. better");
            System.out.println(" 3. best"); System.out.println("-4.
            Excellent!); System.out.println("Choose one:"); choice =
            (char) System.in.read();
        } while( choice < '1' || choice > '4');
        System.out.println("\n");
        switch(choice)
        {
            case '1':    System.out.println("Good");
                        break;
            case '2':    System.out.println("better");
                        break;
            case '3':    System.out.println("best");
                        break;
            case '4':    System.out.println("Excellent");
                        break;
        }
    }
}
```


For

- there are two forms of the **for** loop.
- The first is the traditional form that has been in use since the original version of Java.
- The second is the new `for-each` form.
- general form of the traditional **for** statement:

```
    for(initialization; condition; iteration) {  
        // body  
    }  
class ForTick  
{  
    public static void main(String args[])  
    {  
        int n;  
        for(n=10; n>0; n--)  
            System.out.println("tick " +  
                                n);  
    }  
}
```

Declaring Loop Control Variables Inside the for Loop

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

The For-Each Version of the for Loop

- The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- The for-each style of **for** is also referred to as the *enhanced for* loop.
- The general form for-each version of the **for** is shown
here: `for(type itr-var : collection) statement-block`
- Here, *type* specifies the type
- *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by *collection*.
- There are various types of collections that can be used with the **for**, but the only type used here is the array..

```
class ForEach  
{  
    public static void main(String args[])  
    {  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        int sum = 0;
```

```
for(int x : nums)
{
    System.out.println("Value is: " + x); sum += x;
}
System.out.println("Summation: " + sum);
}
```

- the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

Jump Statements

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

Using break

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

Using break to Exit a Loop

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

```
class BreakLoop
{
    public static void main(String args[])
    {
        for(int i=0; i<100; i++)
        {
            if(i == 10) break;
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
```

i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.

Using **break** as a Form of Goto

- the **break** statement can also be employed by itself to provide a –civilized form of the goto statement.
- Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner.
- The general form of the labeled **break** statement is shown here: `break label;`
- Most often, *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.

return

- The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**.