

Unit-II

Java Basics: History of Java:

- In 1990, Sun Micro Systems Inc. (US) conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called Stealth Project but later its name was changed to Green Project.
- In January 1991, Project Manager James Gosling and his team members Patrick Naughton, Mike Sheridan, Chris Wrath, and Ed Frank met to discuss about this project.
- Gosling thought C and C++ would be used to develop the project. But the problem he faced with them is that they were system dependent languages. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target and could not be used on various processors, which the electronic devices might use.
- James Gosling with his team started developing a new language, which was completely system independent. This language was initially called OAK. Since this name was registered by some other company, later it was changed to Java.
- James Gosling and his team members were consuming a lot of coffee while developing this language. Good quality of coffee was supplied from a place called "Java Island". Hence they fixed the name of the language as Java. The symbol for Java language is cup and saucer.
- Sun formally announced Java at Sun World conference in 1995. On January 23rd 1996, JDK1.0 version was released. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.
- the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences.
- While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java.
- One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

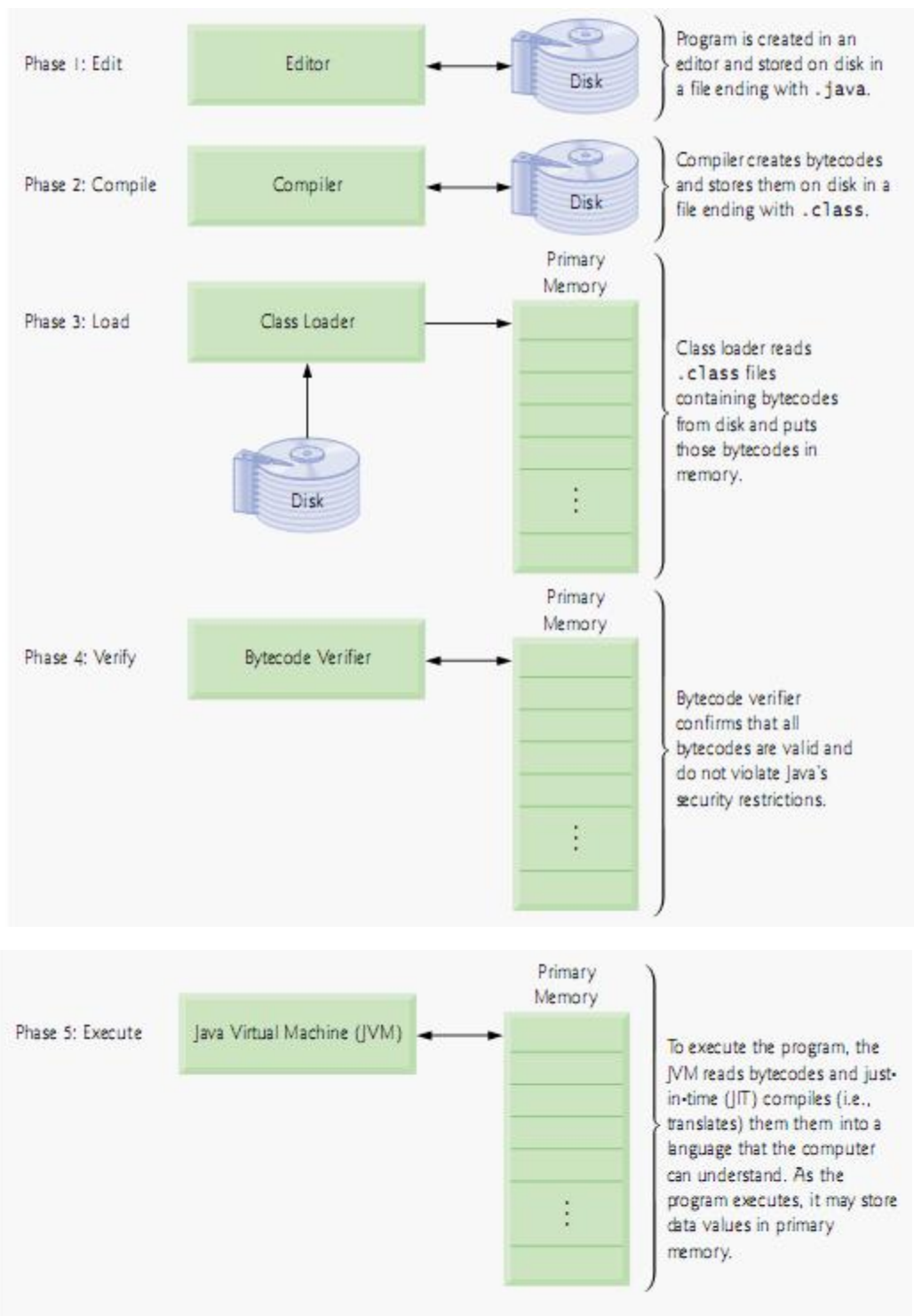


Fig. 1.1 | Typical Java development environment.

Table 2-1 Java Jargon

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on cell phones and other small devices
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Sun's term for a bug fix release
NetBeans	—	Sun's integrated development environment

Table 2-2 Java Directory Tree

Directory Structure	Description
<i>jdk</i>	(The name may be different, for example, <i>jdk5.0</i>)
— <i>bin</i>	The compiler and tools
— <i>demo</i>	Look here for demos
— <i>docs</i>	Library documentation in HTML format (after expansion of <i>j2sdkversion-doc.zip</i>)
— <i>include</i>	Files for compiling native methods (see Volume II)
— <i>jre</i>	Java runtime environment files
— <i>lib</i>	Library files
— <i>src</i>	The library source (after expanding <i>src.zip</i>)

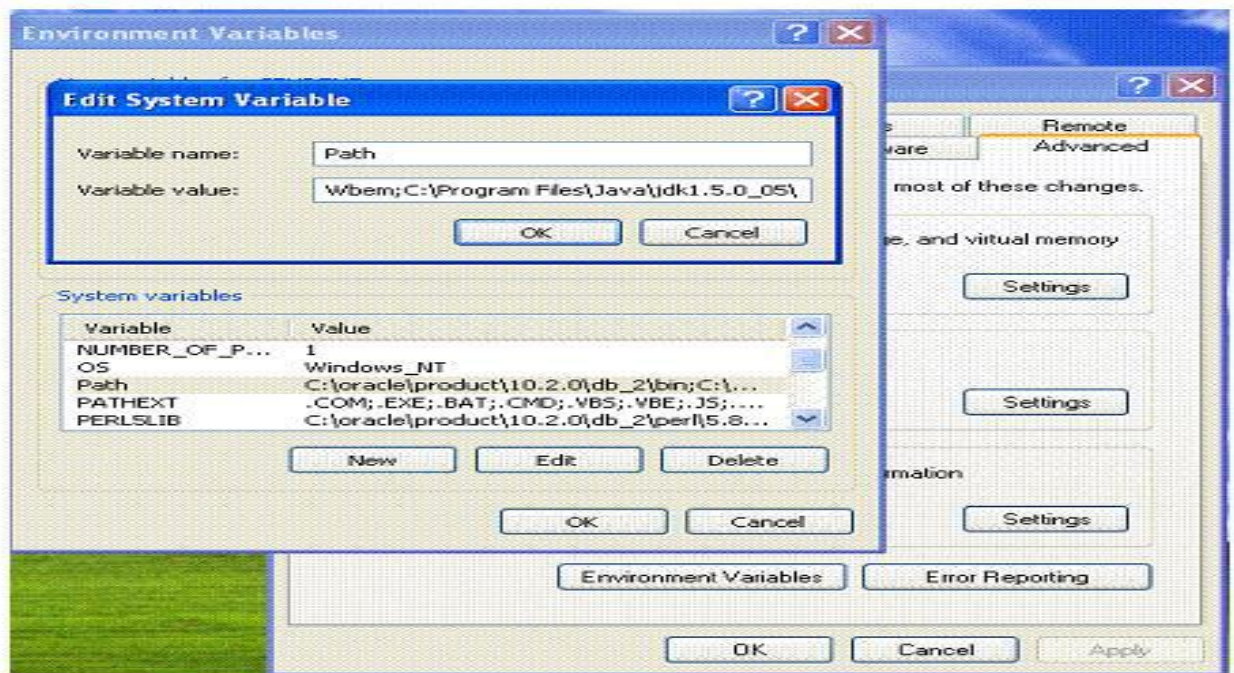
Obtaining the Java Environment:

- Install JDK after downloading, by default JDK will be installed in

C:\Program Files\Java\jdk1.5.0_05 (Here *jdk1.5.0_05* is JDK's version)

Setting up Java Environment: After installing the JDK, we need to set at least one environment variable in order to be able to compile and run Java programs. A PATH environment variable enables the operating system to find the JDK executables when our working directory is not the JDK's binary directory.

- Setting environment variables from a command prompt: If we set the variables from a command prompt, they will only hold for that session. To set the PATH from a command prompt: `set PATH=C:\Program Files\Java\jdk1.5.0_05\bin;.%PATH%`
- Setting environment variables as system variables: If we set the variables as system variables they will hold continuously.
 - o Right-click on My Computer
 - o Choose Properties
 - o Select the Advanced tab
 - o Click the Environment Variables button at the bottom
 - o In system variables tab, select path (system variable) and click on edit button
 - o A window with variable name- path and its value will be displayed.
 - o Don't disturb the default path value that is appearing and just append (add) to that path at the end:;`C:\ProgramFiles\Java\jdk1.5.0_05\bin;.`
 - o Finally press OK button.
- Repeat the process and type at:
 - Variable name class path
 - Variable value:`C:\ProgramFiles\Java\jdk1.5.0_05\lib\tool.jar;.;`
- Finally press OK button.



Simple java program:

Comments: Comments are description about the aim and features of the program.
Comments

increase readability of a program. Three types of comments are there in Java:

- Single line comments: These comments start with //

e.g.: // this is comment line

- Multi line comments: These comments start with /* and end with */

e.g.: /* this is comment line*/

- Java documentation comments: These comments start with /** and end with */

These comments are useful to create a HTML file called API (application programming Interface) document. This file contains description of all the features of software.

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements.

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
```

```
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, [Control Flow Statements](#)

Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, [BlockDemo](#), illustrates the use of blocks:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

Structure of the Java Program:

As all other programming languages, Java also has a structure.

- The first line of the C/C++ program contains include statement. For example, <stdio.h> is the header file that contains functions, like printf (), scanf () etc. So if we want to use any of these functions, we should include this header file in C/ C++ program.
- Similarly in Java first we need to import the required packages. By default java.lang.* is imported. Java has several such packages in its library.
- A package is a kind of directory that contains a group of related classes and interfaces. A class or interface contains methods.
- Since Java is purely an Object Oriented Programming language, we cannot write a Java program without having at least one class or object.
- So, it is mandatory to write a class in Java program. We should use class keyword for this purpose and then write class name.
- In C/C++, program starts executing from main method similarly in Java, program starts executing from main method. The return type of main method is void because program starts executing from main method and it returns nothing.

Sample Program:

```
//A Simple Java Program
import java.lang.System;
import java.lang.String;
class Sample
{
    public static void main(String args[])
    {
        System.out.print ("Hello world");
    }
}
```

- Since Java is purely an Object Oriented Programming language, without creating an object to a class it is not possible to access methods and members of a class.
- But main method is also a method inside a class, since program execution starts from main method we need to call main method without creating an object.
- Static methods are the methods, which can be called and executed without creating objects.
- Since we want to call main () method without using an object, we should declare main () method as static. JVM calls main () method using its Classname.main () at the time of running the program.
- JVM is a program written by Java Soft people (Java development team) and main () is the method written by us. Since, main () method should be available to the JVM, it should be declared as public. If we don't declare main () method as public, then it doesn't make itself available to JVM and JVM cannot execute it.
- JVM always looks for main () method with String type array as parameter otherwise JVM cannot recognize the main () method, so we must provide String type array as parameter to main () method.
- A class code starts with a {and ends with a }.
- A class or an object contains variables and methods (functions). We can create any number of variables and methods inside the class.
- This is our first program, so we had written only one method called main ().
- Our aim of writing this program is just to display a string "Hello world".
- In Java, print () method is used to display something on the monitor.
- A method should be called by using objectname.methodname (). So, to call print () method, create an object to PrintStream class then call objectname.print () method.
- An alternative is given to create an object to PrintStream Class i.e. System.out.
- Here, System is the class name and out is a static variable in System class. out is called a field in System class.
- When we call this field a PrintStream class object will be created internally. So, we can call print() method as: System.out.print ("Hello world");

- println () is also a method belonging to PrintStream class. It throws the cursor to the next line after displaying the result.
- In the above Sample program System and String are the classes present in java.lang package.

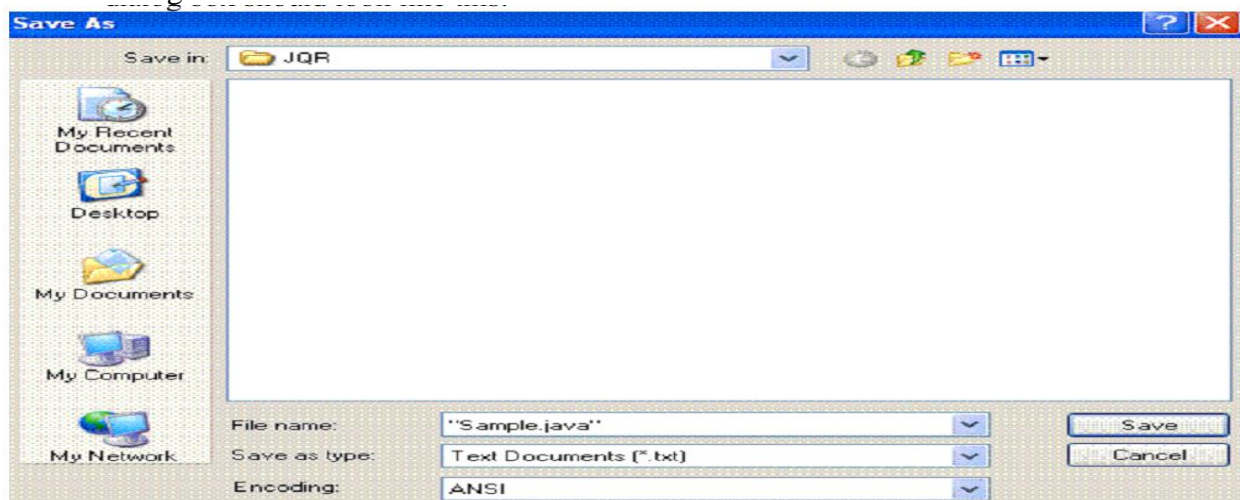
Escape Sequence: Java supports all escape sequence which is supported by C/ C++. A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Creating a Source File:

- Type the program in a text editor (i.e. Notepad, WordPad, Microsoft Word or Edit Plus). We can launch the Notepad editor from the Start menu by selecting Programs > Accessories > Notepad. In a new document, type the above code (i.e. Sample Program).
- Save the program with filename same as Class_name (i.e. Sample.java) in which main method is written. To do this in Notepad, first choose the File > Save menu item. Then, in the Save dialog box:
- Using the Save in combo box, specify the folder (directory) where you'll save your file. In this example, the directory is JQR on the D drive.
- In the File name text field, type "Sample.java", including the quotation marks. Then the dialog box should look like this:



- Now click Save, and exit Notepad.

Compiling the Source File into a .class File:

- To Compile the Sample.java program go to DOS prompt. We can do this from the Start menu by choosing Run... and then entering cmd. The window should look similar to the following figure.



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\USER>
```

- The prompt shows current directory. To compile Sample.java source file, change current directory to the directory where Sample.java file is located. For example, if source directory is JQR on the D drive, type the following commands at the prompt and press Enter:



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\USER>d:
D:\>cd JQR
D:\JQR>_
```

Now the prompt should change to D:\JQR>

- At the prompt, type the following command and press Enter.
javac Sample.java



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Sample.java
D:\JQR>
```

- The compiler generates byte code and Sample.class will be created.

Executing the Program (Sample.class):

- To run the program, enter java followed by the class name created at the time of compilation at the command prompt in the same directory as:
java Sample



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>java Sample
Hello world
D:\JQR>
```

- The program interpreted and the output is displayed.

The Java Virtual Machine: Java Virtual Machine (JVM) is the heart of entire Java program execution process. First of all, the .java program is converted into a .class file consisting of byte code instructions by the java compiler at the time of compilation. Remember, this java compiler is outside the JVM. This .class file is given to the JVM. Following figure shows the architecture of Java Virtual Machine.

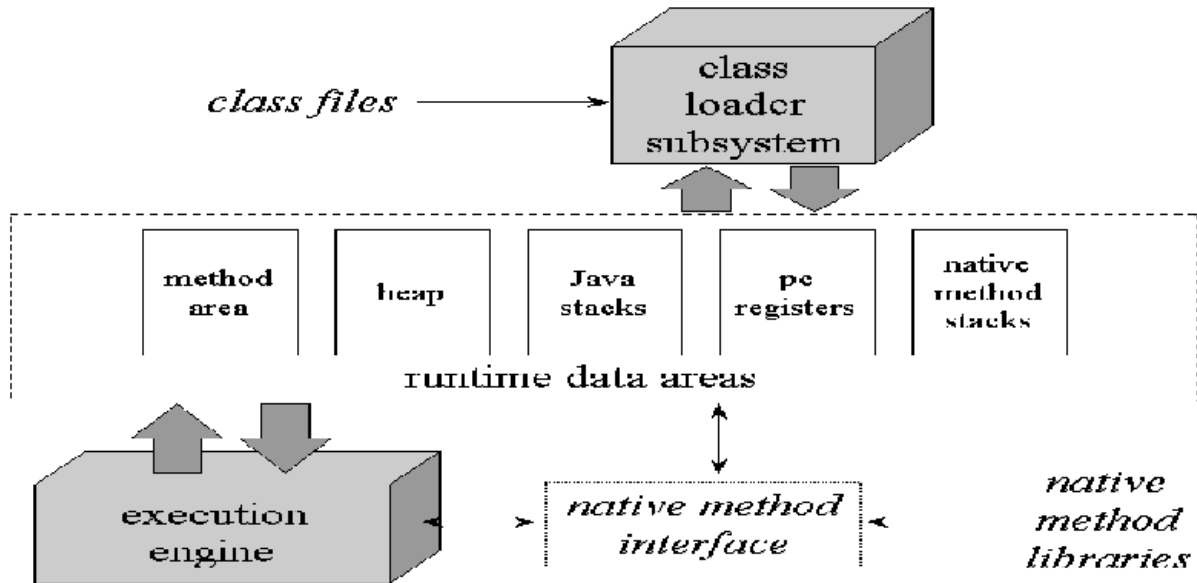


Figure: The internal architecture of the Java virtual machine.

In JVM, there is a module (or program) called class loader sub system, which performs the following instructions:

- First of all, it loads the .class file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program. This memory is divided into 5 parts, called run time data areas, which contain the data and results while running the program. These areas are as follows:
- **Method area:** Method area is the memory block, which stores the class code, code of the variables and code of the methods in the Java program. (Method means functions written in a class).
- **Heap:** This is the area where objects are created. Whenever JVM loads a class, method and heap areas are immediately created in it.
- **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java Stacks. So, Java Stacks are memory area where Java methods are executed. While executing methods, a separate frame will be created in the Java Stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.
- **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instruction of the methods.
- **Native Method Stacks:** Java methods are executed on Java Stacks. Similarly, native methods (for example C/C++ functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called **Native method interface**.

Execution Engine contains interpreter and JIT compiler which translates the byte code instructions into machine language which are executed by the microprocessor. Hot spot (loops/iterations) is the area in .class file i.e. executed by JIT compiler. JVM will identify the Hot spots in the .class files and it will give it to JIT compiler where the normal instructions and statements of Java program are executed by the Java interpreter.

JAVA BUZZWORDS:

Features of Java (Java buzz words):

- Simple: Learning and practicing java is easy because of resemblance with c and C++.
- Object Oriented Programming Language: Unlike C++, Java is purely OOP.
- Distributed: Java is designed for use on network; it has an extensive library which works in agreement with TCP/IP.
- Secure: Java is designed for use on Internet. Java enables the construction of virus-free, tamper free systems.
- Robust (Strong/ Powerful): Java programs will not crash because of its exception handling and its memory management features.
- Interpreted: Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter. .class file will have byte code instructions and JVM which contains an interpreter will execute the byte code.
- Portable: Java does not have implementation dependent aspects and it yields or gives same result on any machine.
- Architectural Neutral Language: Java byte code is not machine dependent, it can run on any machine with any processor and with any OS.
- High Performance: Along with interpreter there will be JIT (Just In Time) compiler which enhances the speed of execution.
- Multithreaded: Executing different parts of program simultaneously is called multithreading. This is an essential feature to design server side programs.
- Dynamic: We can develop programs in Java which dynamically change on Internet (e.g.: Applets).

Naming Conventions: Naming conventions specify the rules to be followed by a Java programmer while writing the names of packages, classes, methods etc.

- Package names are written in small letters.

e.g.: java.io, java.lang, java.awt etc

- Each word of class name and interface name starts with a capital

e.g.: Sample, AddTwoNumbers

- Method names start with small letters then each word start with a capital

e.g.: sum (), sumTwoNumbers (), minValue ()

- Variable names also follow the same above method rule

e.g.: sum, count, totalCount

- Constants should be written using all capital letters

e.g.: PI, COUNT

- Keywords are reserved words and are written in small letters.

e.g.: int, short, float, public, void

DATA TYPES:

As with all modern programming languages, Java supports several types of data. You may use these types to declare variables and to create arrays. As you will see, Java's approach to these items is clean, efficient, and cohesive.

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. Let's see what this means. 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, and both terms will be used in this book. These can be put in four groups:

- **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.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types

found in most other non-object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range.

For example, an int is always 32 bits,

regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

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. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

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. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

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

Let's look at each type of integer.

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.

For example, the following declares two byte variables called b and c:

```
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 $2,147,483,647$. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. 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. 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. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

```
// 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.");
}
}

```

This program generates the following **output**:

In 1000 days light will travel about 16070400000000 miles.

Clearly, the result could not have been held in an int variable.

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, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

Float:

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

For example, float can be useful when representing dollars and cents.

Here are **some example** 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.

All transcendental math functions, such as `sin()`, `cos()`, and `sqrt()`, return double values. When you need to maintain accuracy many iterative calculations, or are manipulating large-valued numbers, `double` is the best choice.

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`.

Note: `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**. The range of a `char` is 0 to 65,536. 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. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits.

But such is the price that must be paid for global portability.

NOTE: More information about Unicode can be found at <http://www.unicode.org>.

Here is a program that demonstrates char variables:

```
// 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);

}

}
```

This program displays the following

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. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the “old tricks” that you may have used with characters in other languages will work in Java, too.

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. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

```
// 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);  
}  
}
```

The **output** generated by this program is shown here:

ch1 contains X

ch1 is now Y

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.

Here is a program that demonstrates the boolean type:

```
// 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 statementChapter 3: Data Types, Variables, and Arrays 39  
        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));  
    }  
}
```

```
}
```

The **output** generated by this program is shown here:

```
b is false
```

```
b is true
```

```
This is executed.
```

```
10 > 9 is true
```

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.” Further, the extra set of parentheses around `10>9` is necessary because the `+` operator has a higher precedence than the `>`.

Escape Sequence: Java supports all escape sequence which is supported by C/ C++. A character preceded by a backslash (`\`) is an escape sequence and has special meaning to the compiler. When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Escape Sequence	Description
<code>\t</code>	Insert a tab in the text at this point.
<code>\b</code>	Insert a backspace in the text at this point.
<code>\n</code>	Insert a newline in the text at this point.
<code>\r</code>	Insert a carriage return in the text at this point.
<code>\f</code>	Insert a form feed in the text at this point.
<code>\'</code>	Insert a single quote character in the text at this point.
<code>\"</code>	Insert a double quote character in the text at this point.
<code>\\</code>	Insert a backslash character in the text at this point.

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. These elements are examined next.

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. (Class and interface types are discussed later in Part I of this book.) The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

```
int a, b, c;           // declares three ints, a, b, and c.
```

```
int d = 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'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

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.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// 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 (using the

Pythagorean theorem). 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 key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the `main()` method. However, Java allows variables to be declared within any block.

A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. 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**.

However, these traditional scopes do not fit well with Java's strict, object-oriented model.

While it is possible to create what amounts to being a global scope, it is by far the not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. 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. Because of the differences, a discussion of class scope (and variables declared within it) is deferred, when classes are described.

For now, we will only examine the scopes defined by or within a method.

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. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// 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 the `if` block, `x` can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

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.

For example, this fragment is invalid because `count` cannot be used prior to its declaration:

```

// 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.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program.

```
// Demonstrate lifetime of a variable.

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

The output generated by this program is shown here:

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

As you can see, *y* is reinitialized to -1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

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

```
// This program will not compile

class ScopeErr {
public static void main(String args[]) {
```

```
int bar = 1;
{
    // creates a new scope
int bar = 2; // Compile-time error – bar already defined!
}
}
}
```

Type Conversion and Casting:

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.

For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs **an explicit conversion** between incompatible types.

Let's look at both automatic type conversions and casting.

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.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called

a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

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

(target-type) value;

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

```
int a;
```

```
byte b;
```

```
// ...
```

```
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some **type conversions that require casts**:

```
// 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);
}
}
```

This program generates the following **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 67Let's look at each conversion. 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.

When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

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

examine the following expression:

```
byte a = 40;
```

```
byte b = 50;
```

```
byte c = 100;
```

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

The result of the intermediate term $a*b$ easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression.

This means that the subexpression $a*b$ 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.

For example, this seemingly correct code causes a problem:

```
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. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

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

which yields the correct value of 100.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions.

They are as follows:

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

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
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);
}
}

```

Let's look closely at the type promotions that occur in this line from the program:

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

In the first subexpression, $f * b$, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression 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 subexpression is double.

Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Conclusion of Type Conversion:

- Size Direction of Data Type
 - Widening Type Conversion (Casting down)
 - Smaller Data Type → Larger Data Type
 - Narrowing Type Conversion (Casting up)
 - Larger Data Type → Smaller Data Type
- Conversion done in two ways
 - Implicit type conversion
 - Carried out by compiler automatically
 - Explicit type conversion
 - Carried out by programmer using casting
- Widening Type Conversion
 - Implicit conversion by compiler automatically

```
byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double
```

- Narrowing Type Conversion
 - Programmer should describe the conversion explicitly

```
byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float
```

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double
- General form: (targetType) value
- Examples:
 - 1) integer value will be reduced module bytes range:

```
int i;
```

```
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
```

```
int i = (int) f;
```

ACCESS CONTROL:

Type of modifier	Keyword	Description
Access modifier (use only one)	public	Data field is available everywhere (when the class is also declared <i>public</i>).
	private	Data field is available only within the class.
	protected	Data field is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Data field is available within the class and within the package.
Use modifiers (all can be used at once)	static	Indicates that only one such data field is available for all instances of this class. Without this modifier, each instance has its own copy of a data field.
	final	The value provided for the data field cannot be modified (a constant).
	transient	The data field is not part of the persistent state of the object.
	volatile	The value provided for the data field can be accessed by multiple threads of control. Java ensures that the freshest copy of the data field is always used.

Access Specifiers: An access specifier is a key word that represents how to access a member of a class. There are mainly four access specifiers in java.

- o private: private members of a class are not available outside the class.
- o public: public members of a class are available anywhere outside the class.
- o protected: protected members are available outside the class.
- o default: if no access specifier is used then default specifier is used by java compiler.

Default members are available outside the class.

The access modifiers supported by Java are static, final, abstract, synchronized, native, volatile, transient and strictfp.

Can u declare a class as 'private'?

No, if we declare a class as private, then it is not available to java compiler and hence a compile time error occurs. But, inner classes can be declared as private.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;

private double j;

private int myMethod(int a, char b) { // ...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
```

```
public and private.
```

```
*/
```

```
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access
    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() { // get c's value
        return c;
    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();
```

```

// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;

// This is not OK and will cause an error
// ob.c = 100; // Error!

// You must access c through its methods
ob.setc(100); // OK

System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}

```

As you can see, inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods setc() and getc(). If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the Stack class shown at the end.

```
// This class defines an integer stack that can hold 10 values.
```

```
class Stack {
```

```
/* Now, both stck and tos are private. This means that they cannot be accidentally or maliciously
altered in a way that would be harmful to the stack.*/
```

```
private int stck[] = new int[10];
```

```
private int tos;
```

```
// Initialize top-of-stack
```

```
Stack() {
```

```
tos = -1;
```



```

}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}

```

As you can see, now both `stck`, which holds the stack, and `tos`, which is the index of the top of the stack, are specified as `private`. This means that they cannot be accessed or altered except through `push()` and `pop()`. Making `tos` `private`, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the `stck` array.

The following program demonstrates the improved `Stack` class. Try removing the commented-out lines to prove to yourself that the `stck` and `tos` members are, indeed, inaccessible.

```

class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

```

```

// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);

// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());

// these statements are not legal

// mystack1.tos = -2;

// mystack2.stck[3] = 100;

}

}

```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword `static`. When a member is declared `static`, it can be accessed before any objects of its class are created, and without reference to any object.

You can declare both methods and variables to be `static`. The most common example of a `static` member is `main()`. **`main()` is declared as `static` because it must be called before any objects exist.**

Instance variables declared as `static` are, essentially, global variables. When objects of its class are declared, no copy of a `static` variable is made. Instead, all instances of the class share the same `static` variable.

Methods declared as static have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way. (The keyword super relates to inheritance.).

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
```

```
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then

main() is called, which calls meth(), passing 42 to x. The three println() statements refer to the two static variables a and b, as well as to the local variable x.

Here is the **output** of the program:

Static block initialized.

x = 42

a = 3

b = 12

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by **the dot operator**.

For example, if you wish to call a static method from outside its class, you can do so using the following general form:

```
classname.method( )
```

Here, classname is the name of the class in which the static method is declared. As you can see, this format is similar to that used to call non-static methods through object-reference variables.

A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside main(), the static method callme() and the static variable b are accessed through their class name StaticDemo.

```
class StaticDemo {  
  
    static int a = 42;  
  
    static int b = 99;  
  
    static void callme() {  
  
        System.out.println("a = " + a);  
  
    }  
  
}  
  
class StaticByName {  
  
    public static void main(String args[]) {  
  
        StaticDemo.callme();  
  
        System.out.println("b = " + StaticDemo.b);  
  
    }  
  
}
```

```
}  
}
```

Here is the output of this program:

```
a = 42
```

```
b = 99
```

Introducing final

A variable can be declared as final. Doing so prevents its contents from being modified.

This means that you must initialize a final variable when it is declared. For example:

```
final int FILE_NEW = 1;
```

```
final int FILE_OPEN = 2;
```

```
final int FILE_SAVE = 3;
```

```
final int FILE_SAVEAS = 4;
```

```
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final variables.

Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

native Modifier:

"**native**" keyword applies to methods. The code of a native method is written in some other language like C or C++. At execution time, this native code is executed separately and put with the Java output and is given.

To do all this, Java includes Java Native Interface (JNI) which is inbuilt into JDK. "native" permits the Java developer to write the machine-dependent code in other language (like C/C++) and make use in Java. Alternatively, if a C or C++ function exists with a very complex code, it is not required to convert it into Java code and instead can be used directly in a Java program.

Observe some methods of Java API.

```
public static native void sleep(long) throws InterruptedException;    //  
belonging to Thread class
```

```
public native int read() throws IOException;    // belonging to
FileInputStream class
```

Observe **native keyword** in the above two statements. The functions' code is written in other language (means, not in Java). To execute these methods, Java takes the help of underlying OS.

volatile Modifier

The keyword **volatile** applies to variables and objects. A volatile variable value is more likely to get changed in the code. A volatile variable is treated differently by the JVM.

Volatile variables are not optimized (to minimize execution time) for **performance** by the compiler as their values are expected to get changed at any time without information. Volatile is better used with multiple threads that can change a variable value often.

Volatile can be used as follows.

```
public volatile int rate = 10;
```

Volatile does not have any meaning when applied to final variables or immutable objects or synchronized block.

The following two statements give compilation error.

```
final volatile int x = 10;
volatile final int x = 10;
```

transient Modifier:

It is used with RMI technology. "**transient**" is a keyword which means the data is not serialized. In RMI (Remote Method Invocation) technology, the objects along with the data are serialized. If the data is not required to be serialized and thereby not sent to the remote server, then declare the data as transient.

Following is the way how to declare a variable as transient.

```
public transient double goldRate = 210.5;
strictfp Modifier
```

A **floating-point value**, in a language, is platform-dependent. That is, the same floating value, in a Java program, executed on different operating systems may give different outputs (precision). To get the same precision (regardless of hardware, software and OS) on every operating system, declare the class or method as strictfp. "strictfp" is a keyword added in JDK 1.2 version.

"strictfp" is abbreviation for "strict floating-point".

```

public strictfp class Demo
public strictfp interface Demo
public strictfp void display()

```

If a class is strictfp, all the code in the class is evaluated with the strict floating point precision as laid in IEEE 754 standards. strictfp follows the rules formatted by IEEE 754. JVM does not apply its own precision.

Following table gives the list of access specifiers and modifiers that can be applied to variables, methods and classes.

specifier/modifier	local variable	instance variable	method	class
public	NA	A	A	A
protected	NA	A	A	NA
default	A	A	A	A
private	NA	A	A	NA
final	A	A	A	A
static	NA	A	A	NA
synchronized	NA	NA	A	NA
native	NA	NA	A	NA
volatile	NA	A	NA	NA
transient	NA	A	NA	NA
strictfp	NA	NA	A	A

A: Allowed NA: Not Allowed

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. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

On which memory, arrays are created in java?

Arrays are created on dynamic memory by JVM. There is no question of static memory in Java; everything (variable, array, object etc.) is created on dynamic memory only.

Arrays: An array represents a group of elements of same data type. Arrays are generally categorized into two types:

- Single Dimensional arrays (or 1 Dimensional arrays)
- Multi-Dimensional arrays (or 2 Dimensional arrays, 3 Dimensional arrays, ...)

Single 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.

A one dimensional array or single dimensional array represents a row or a column of elements. For example, the marks obtained by a student in 5 different subjects can be represented by a 1D array.

- We can declare a one dimensional array and directly store elements at the time of its declaration, as: `int marks[] = {50, 60, 55, 67, 70};`
- We can create a 1D array by declaring the array first and then allocate memory for it by using new operator, as:
`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.

```
int marks [ ]; //declare marks array
```

```
marks = new int[5]; //allot memory for storing 5 elements
```

These two statements also can be written as:

```
int marks [ ] = new int [5];
```

- We can pass the values from keyboard to the array by using a loop, as given here

```
for(int i=0;i<5;i++)  
{  
    //read integer values from keyboard and store into marks[i]  
    Marks[i]=Integer.parseInt(br.readLine());  
}
```

Let us examine some more examples for 1D array:

```
float salary[ ]={5670.55f,12000f};
```

```
float [ ] salary={5670.55f,12000f};
```

```
string names[ ]=new string[10];
```

```
string [ ] names={'v', 'r'};
```


Let's review: 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.

The advantage of using arrays is that they simplify programming by replacing a lot of statements by just one or two statements. In C/C++, by default, arrays are created on static memory unless pointers are used to create them. In java, arrays are created on dynamic memory i.e., allotted at runtime by JVM.

Program : Write a program to accept elements into an array and display the same.

```
// program to accept elements into an array and display the same.
```

```
import java.io.*;
```

```
class ArrayDemo1
```

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

```
{ //Create a BufferedReader class object (br)
```

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

```
    System.out.println ("How many elements: ");
```

```
int n = Integer.parseInt (br.readLine ());
```

```
//create a 1D array with size n
```

```
int a[] = new int[n];
```

```
System.out.print ("Enter elements into array : ");
```

```
for (int i = 0; i<n;i++)
```

```
    a [i] = Integer.parseInt ( br.readLine ());
```

```
System.out.print ("The entered elements in the array are: ");
```

```
for (int i =0; i < n; i++)
```

```
    System.out.print (a[i] + "\t");
```

```
}
```

```
}
```

Output:



```
C:\WINDOWS\system32\command.com
D:\JQR>javac ArrayDemo1.java
D:\JQR>java ArrayDemo1
How many elements: 5
Enter elements into array : 10
20
30
40
50
The entered elements in the array are: 10      20      30      40      50
D:\JQR>
```

Here is a program that creates an array of the number of days in each month.

// 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;
month_days[1] = 28;
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`. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// 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.");  
  
    }  
  
}
```

When you run this program, you see the same output as that generated by the previous version.

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

For example, the run-time system will check the value of each index into `month_days` to make sure that it is between 0 and 11 inclusive. 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 runtime error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.
```

```
class Average {  
    public static void main(String args[]) {  
        double nums[] = { 10.1, 11.2, 12.3, 13.4, 14.5};  
        double result = 0;  
        int i;  
        for(i=0; i<5; i++)  
            result = result + nums[i];  
        System.out.println("Average is " + result / 5);  
    }  
}
```

```
//sorting of names in ascending order
```

```
import java.io.*;  
import java.lang.*;  
import java.util.*;  
class Sorting  
{  
    public static void main(String[] args)  
    {  
  
        int k=args.length;  
        String temp=new String();  
        String names[]=new String[k+1];
```

```

for(int i=0;i<k;i++)
{
    names[i]=args[i];
}
for(int i=0;i<k;i++)
    for(int j=i+1;j<k;j++)
    {
        if(names[i].compareTo(names[j])<0)
        {
            temp=names[i];
            names[i]=names[j];
            names[j]=temp;
        }
    }
System.out.println("Sorted order is");
for(int i=k-1;i>=0;i--)
{
    System.out.println(names[i]);
}
}
}

```

Output: Java Sorting veer ram ajay

Ajay

Ram

Veer

Multi-Dimensional Arrays (2D, 3D ... arrays):

A two dimensional array is a combination of two or more (1D) one dimensional arrays. A three dimensional array is a combination of two or more (2D) two dimensional arrays.

- **Two Dimensional Arrays (2d array):** A two dimensional array represents several rows and columns of data. To represent a two dimensional array, we should use two pairs of square braces [] [] after the array name. For example, the marks obtained by a group of students in five different subjects can be represented by a 2D array.

o We can declare a two dimensional array and directly store elements at the time of its declaration, as:

```
int marks[] [] = {{50, 60, 55, 67, 70},{62, 65, 70, 70, 81}, {72, 66, 77, 80, 69}};
```

o We can create a two dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

```
int marks[ ] [ ]; //declare marks array
```

```
marks = new int[3][5]; //allot memory for storing 15 elements.
```

These two statements also can be written as: `int marks [][] = new int[3][5];`

Program : Write a program to take a 2D array and display its elements in the form of a matrix.

```
//Displaying a 2D array as a matrix
```

```
class Matrix
```

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

```
{ //take a 2D array
```

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

```
// display the array elements
```

```
for (int i = 0 ; i < 2 ; i++)
```

```
{ System.out.println ();
```

```
for (int j = 0 ; j < 3 ; j++)
```

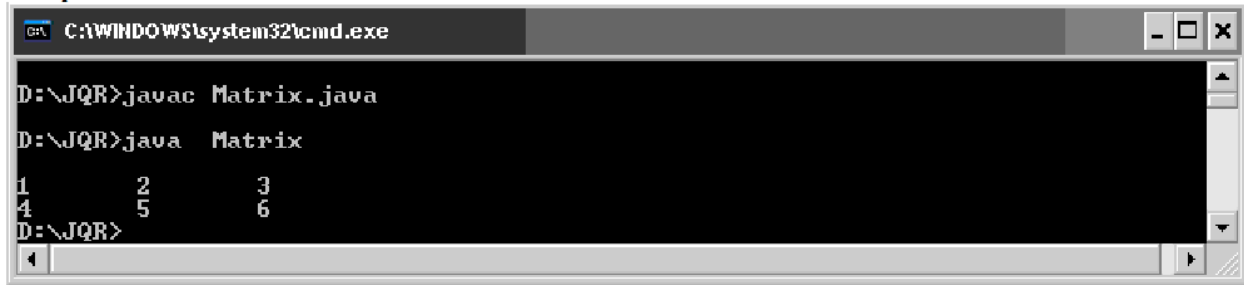
```
System.out.print(x[i][j] + "\t");
```

```
}
```

```
}
```

```
}
```

Output:



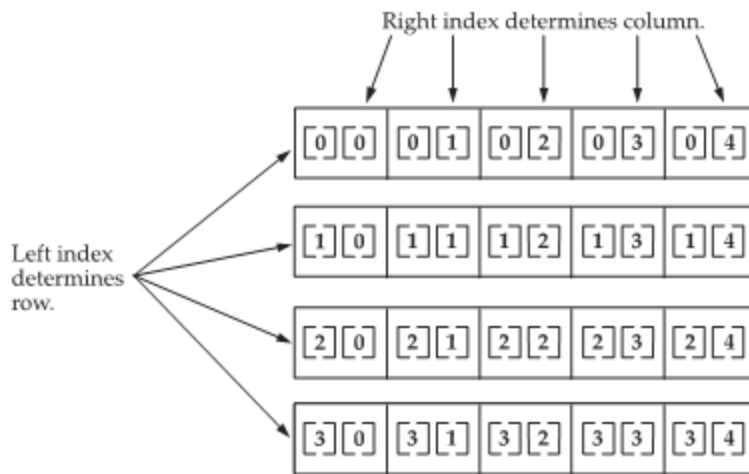
```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Matrix.java
D:\JQR>java Matrix
1 2 3
4 5 6
D:\JQR>
```

// Demonstrate a two-dimensional array.

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, 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();
}
}
}
```

This program generates the following output:

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



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

- **Three Dimensional arrays (3D arrays):** We can consider a three dimensional array as a combination of several two dimensional arrays. To represent a three dimensional array, we should use three pairs of square braces [] [] [] after the array name.

o We can declare a three dimensional array and directly store elements at the time of its declaration, as:

```
int arr[ ] [ ] [ ] = {{{50, 51, 52},{60, 61, 62}}, {{70, 71, 72}, {80, 81, 82}}};
```

o We can create a three dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

```
int arr[ ] [ ] = new int[2][2][3]; //allot memory for storing 15 elements.
```

//example for 3-D array

```
class ThreeD
```

```
{
```

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

```
    {
```

```
        int dept, student, marks, tot=0;
```

```
        int
```

```
arr[][][]={{50,51,52},{60,61,62}},{{70,71,72},{80,81,82}},{{65,66,67},{75,76,77}}};
```

```
        for(dept=0;dept<3;dept++)
```

```
        {
```



```

        System.out.println("dept"+(dept+1)+":");
        for(student=0;student<2;student++)
        {
            System.out.print("student"+(student+1)+"marks:");
            for(marks=0;marks<3;marks++)
            {
                System.out.print(arr[dept][student][marks]+" ");
                tot+=arr[dept][student][marks];
            }
            System.out.println("total:"+tot);
            tot=0;
        }
        System.out.println();
    }
}
}
}

```

arrayname.length: If we want to know the size of any array, we can use the property 'length' of an array. In case of 2D, 3D length property gives the number of rows of the array.

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:

```
int a1[] = 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
```

The alternative declaration form is also useful when specifying an array as a return type for a method.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy they are stored as strings in a String array passed to the `args` parameter of `main()`.

The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following **output**:

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.

Operators:

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

Operators: An operator is a symbol that performs an operation. An operator acts on variables called operands.

- **Arithmetic operators:** These operators are used to perform fundamental operations like addition, subtraction, multiplication etc.

Operator	Meaning	Example	Result
+	Addition	3 + 4	7
-	Subtraction	5 - 7	-2
*	Multiplication	5 * 5	25
/	Division (gives quotient)	14 / 7	2
%	Modulus (gives remainder)	20 % 7	6

Program : Write a program to perform arithmetic operations

```
//Addition of two numbers
```

```
class AddTwoNumbers
```

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

```
{ int i=10, j=20;
```

```
System.out.println("Addition of two numbers is : " + (i+j));
```

```
System.out.println("Subtraction of two numbers is : " + (i-j));
```

```
System.out.println("Multiplication of two numbers is : " + (i*j));
```

```

System.out.println("Quotient after division is : " + (i/j) );

System.out.println("Remainder after division is : " +(i%j) );

}

}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac AddTwoNumbers.java
D:\JQR>java AddTwoNumbers
Addition of two numbers is : 30
Subtraction of two numbers is : -10
Multiplication of two numbers is : 200
Quotient after division is : 0
Remainder after division is : 10
D:\JQR>

```

➤ **Assignment operator:** This operator (=) is used to store some value into a variable.

Simple Assignment	Compound Assignment
$x = x + y$	$x += y$
$x = x - y$	$x -= y$
$x = x * y$	$x *= y$
$x = x / y$	$x /= y$

Here is a sample program that shows several op= assignments in action:

```
// 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);
}
}

```

The output of this program is shown here:

```

a = 6
b = 8
c = 3

```

➤ **Unary operators:** As the name indicates unary operator's act only on one operand.

Operator	Meaning	Example	Explanation
-	Unary minus	j = -k;	k value is negated and stored into j
++	Increment Operator	b++; ++b;	b value will be incremented by 1 (called as post incrementation) b value will be incremented by 1 (called as pre incrementation)
--	Decrement Operator	b--; --b;	b value will be decremented by 1 (called as post decrementation) b value will be decremented by 1 (called as pre decrementation)

The following program demonstrates the increment operator.

```

// 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);
}
}

```

The output of this program follows:

```

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

```

➤ **Relational operators:** These operators are used for comparison purpose.

Operator	Meaning	Example
==	Equal	x == 3
!=	Not equal	x != 3
<	Less than	x < 3
>	Greater than	x > 3
<=	Less than or equal to	x <= 3

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==" , not "=", when testing if two primitive values are equal.

The following program, Comparison Demo, tests the comparison operators:

```

class ComparisonDemo {
    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
    }
}

```

```

        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}

```

Output:

```

value1 != value2
value1 < value2
value1 <= value2

```

The Type Comparison Operator instanceof

The `instanceof` operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, `InstanceofDemo`, defines a parent class (named `Parent`), a simple interface (named `MyInterface`), and a child class (named `Child`) that inherits from the parent and implements the interface.

```

class InstanceofDemo {
    public static void main(String[] args) {

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}

```

Output:

```

obj1 instanceof Parent: true

```

```
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the `instanceof` operator, keep in mind that `null` is not an instance of anything.

- **Logical operators:** Logical operators are used to construct compound conditions. A compound condition is a combination of several simple conditions.

Operator	Meaning	Example	Explanation
&&	and operator	if(a>b && a>c) System.out.print("yes");	If a value is greater than b and c then only yes is displayed
	or operator	if(a==1 b==1) System.out.print("yes");	If either a value is 1 or b value is 1 then yes is displayed
!	not operator	if(!(a==0)) System.out.print("yes");	If a value is not equal to zero then only yes is displayed

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

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

The following table shows the effect of each logical operation:

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

Here is a program that is almost the same as the `BitLogic` example shown earlier, but it operates on boolean logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
```

```
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);
}
}

```

After running this program, you will see that the same logical rules apply to Boolean values as they did to bits. As you can see from the following **output**, the string representation of a Java boolean value is one of the literal values true or false:

```

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

```

- **Bitwise operators:** These operators act on individual bits (0 and 1) of the operands. They act only on integer data types, i.e. byte, short, long and int.

Operator	Meaning	Explanation
&	Bitwise AND	Multiplies the individual bits of operands
	Bitwise OR	Adds the individual bits of operands
^	Bitwise XOR	Performs Exclusive OR operation
<<	Left shift	Shifts the bits of the number towards left a specified number of positions
>>	Right shift	Shifts the bits of the number towards right a specified number of positions and also preserves the sign bit.
>>>	Zero fill right shift	Shifts the bits of the number towards right a specified number of positions and it stores 0 (Zero) in the sign bit.
~	Bitwise complement	Gives the complement form of a given number by changing 0's as 1's and vice versa.

Program : Write a program to perform Bitwise operations

```
//Bitwise Operations
```

```
class Bits
```

```
{ public static void main(String args[])
{ byte x,y;
x=10;
y=11;
System.out.println ("~x="+(~x));
System.out.println ("x & y="+(x&y));
System.out.println ("x | y="+(x|y));
System.out.println ("x ^ y="+(x^y));
System.out.println ("x<<2="+(x<<2));
System.out.println ("x>>2="+(x>>2));
System.out.println ("x>>>2="+(x>>>2));
}
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Bits.java
D:\JQR>java Bits
^x=-11
x & y=10
x ! y=11
x ^ y=1
x << 2=40
x >> 2=2
x >>> 2=2
D:\JQR>_
```

- **Ternary Operator or Conditional Operator (? :):** This operator is called ternary because it acts on 3 variables.

The syntax for this operator is:

Variable = Expression1? Expression2: Expression3;

First Expression1 is evaluated. If it is true, then Expression2 value is stored into variable otherwise Expression3 value is stored into the variable.

e.g.: max = (a>b) ? a: b;

Here is a program that demonstrates the ? operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
public static void main(String args[]) {
int i, k;
i = 10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
i = -10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
```

```

System.out.println(i + " is " + k);
}
}

```

The **output** generated by the program is shown here:

Absolute value of 10 is 10

Absolute value of -10 is 10

Now that you've learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start.

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order.

The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence.

Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence	
Operators	Precedence
Postfix	<code>expr++ expr--</code>
Unary	<code>++expr --expr +expr -expr ~ !</code>
multiplicative	<code>* / %</code>
Additive	<code>+ -</code>

Shift	<< >> >>>
Relational	< > <= >= instanceof
Equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
Ternary	? :
Assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

Expressions:

Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value. You've already seen examples of expressions, illustrated in bold below:

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);
```

```
int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100 // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100 // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

```
x+y/100
```

```
x + (y / 100) // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

Regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data.

You must learn a specific syntax to create regular expressions — one that goes beyond the normal syntax of the Java programming language. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression.

This trail teaches the regular expression syntax supported by the [java.util.regex](#) API and presents several working examples to illustrate how the various objects interact. In the world of regular expressions, there are many different flavors to choose from, such as grep, Perl, Tcl, Python, PHP, and awk. The regular expression syntax in the java.util.regex API is most similar to that found in Perl.

How Are Regular Expressions Represented in This Package?

The java.util.regex package primarily consists of three classes: Pattern, Matcher, and PatternSyntaxException.

- A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument; the first few lessons of this trail will teach you the required syntax.
- A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object.
- A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

A regular expression is a pattern of characters that describes a set of strings. You can use the java.util.regex package to find, display, or modify some or all of the occurrences of a pattern in an input sequence.

The simplest form of a regular expression is a literal string, such as "Java" or "programming." Regular expression matching also allows you to test whether a string fits into a specific syntactic form, such as an email address.

To develop regular expressions, ordinary and special characters are used:

\\$	^	.	*
+	?	[]
\.			

Any other character appearing in a regular expression is ordinary, unless a `\` precedes it.

Special characters serve a special purpose. For instance, the `.` matches anything except a new line. A regular expression like `s.n` matches any three-character string that begins with `s` and ends with `n`, including `sun` and `son`.

There are many special characters used in regular expressions to find words at the beginning of lines, words that ignore case or are case-specific, and special characters that give a range, such as `a-e`, meaning any letter from `a` to `e`.

Regular expression usage using this new package is Perl-like, so if you are familiar with using regular expressions in Perl, you can use the same expression syntax in the Java programming language. If you're not familiar with regular expressions here are a few to get you started:

Construct	Matches
------------------	----------------

Characters

<code>x</code>	The character <code>x</code>
<code>\\</code>	The backslash character
<code>\0n</code>	The character with octal value <code>0n</code> ($0 \leq n \leq 7$)
<code>\0nn</code>	The character with octal value <code>0nn</code> ($0 \leq n \leq 7$)
<code>\0mnn</code>	The character with octal value <code>0mnn</code> ($0 \leq m \leq 3, 0 \leq n \leq 7$)
<code>\xhh</code>	The character with hexadecimal value <code>0xhh</code>
<code>\uhhhh</code>	The character with hexadecimal value <code>0xhhhh</code>

<code>\t</code>	The tab character (<code>\u0009</code>)
<code>\n</code>	The newline (line feed) character (<code>\u000A</code>)
<code>\r</code>	The carriage-return character (<code>\u000D</code>)
<code>\f</code>	The form-feed character (<code>\u000C</code>)
<code>\a</code>	The alert (bell) character (<code>\u0007</code>)
<code>\e</code>	The escape character (<code>\u001B</code>)
<code>\cx</code>	The control character corresponding to x

Character Classes

<code>[abc]</code>	a, b, or c (simple class)
<code>[^abc]</code>	Any character except a, b, or c (negation)
<code>[a-zA-Z]</code>	a through z or A through Z, inclusive (range)
<code>[a-z-[bc]]</code>	a through z, except for b and c: <code>[a-d-z]</code> (subtraction)
<code>[a-z-[m-p]]</code>	a through z, except for m through p: <code>[a-lq-z]</code>
<code>[a-z-[^def]]</code>	d, e, or f

Predefined Character Classes

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Classes and Methods

The following classes match character sequences against patterns specified by regular expressions.

Pattern Class

An instance of the Pattern class represents a regular expression that is specified in string form in a syntax similar to that used by Perl.

A regular expression, specified as a string, must first be compiled into an instance of the Pattern class. The resulting pattern is used to create a Matcher object that matches arbitrary character sequences against the regular expression. Many matchers can share the same pattern because it is stateless.

The compile method compiles the given regular expression into a pattern, then the matcher method creates a matcher that will match the given input against this pattern. The pattern method returns the regular expression from which this pattern was compiled.

The split method is a convenience method that splits the given input sequence around matches of this pattern. The following example demonstrates:

```
/*
 * Uses split to break up a string of input separated by
 * commas and/or whitespace.
 */
import java.util.regex.*;

public class Splitter {
    public static void main(String[] args) throws Exception {
        // Create a pattern to match breaks
        Pattern p = Pattern.compile("[,\\s]+");
        // Split input with the pattern
        String[] result =
            p.split("one,two, three four , five");
        for (int i=0; i<result.length; i++)
            System.out.println(result[i]);
    }
}
```

Matcher Class

Instances of the Matcher class are used to match character sequences against a given string sequence pattern. Input is provided to matchers using the CharSequence interface to support matching against characters from a wide variety of input sources.

A matcher is created from a pattern by invoking the pattern's matcher method. Once created, a matcher can be used to perform three different kinds of match operations:

- The matches method attempts to match the entire input sequence against the pattern.
- The lookingAt method attempts to match the input sequence, starting at the beginning, against the pattern.
- The find method scans the input sequence looking for the next sequence that matches the pattern.

Each of these methods returns a boolean indicating success or failure. More information about a successful match can be obtained by querying the state of the matcher.

This class also defines methods for replacing matched sequences by new strings whose contents can, if desired, be computed from the match result.

The appendReplacement method appends everything up to the next match and the replacement for that match. The appendTail appends the strings at the end, after the last match.

For instance, in the string `blahcatblahcatblah`, the first `appendReplacement` appends `blahdog`. The second `appendReplacement` appends `blahdog`, and the `appendTail` appends `blah`, resulting in: `blahdogblahdogblah`. See [Simple word replacement](#) for an example.

CharSequence Interface

The `CharSequence` interface provides uniform, read-only access to many different types of character sequences. You supply the data to be searched from different sources. `String`, `StringBuffer` and `CharBuffer` implement `CharSequence`, so they are easy sources of data to search through. If you don't care for one of the available sources, you can write your own input source by implementing the `CharSequence` interface.

Example Regex Scenarios

The following code samples demonstrate the use of the `java.util.regex` package for various common scenarios:

Simple Word Replacement

```
/*
 * This code writes "One dog, two dogs in the yard."
 * to the standard-output stream:
 */
import java.util.regex.*;

public class Replacement {
    public static void main(String[] args)
        throws Exception {
        // Create a pattern to match cat
        Pattern p = Pattern.compile("cat");
        // Create a matcher with an input string
        Matcher m = p.matcher("one cat," +
            " two cats in the yard");
        StringBuffer sb = new StringBuffer();
        boolean result = m.find();
        // Loop through and create a new String
        // with the replacements
        while(result) {
            m.appendReplacement(sb, "dog");
            result = m.find();
        }
        // Add the last segment of input to
        // the new String
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

```
}  
}
```

Email Validation

The following code is a sample of some characters you can check are in an email address, or should not be in an email address. It is not a complete email validation program that checks for all possible email scenarios, but can be added to as needed.

```
/*  
 * Checks for invalid characters  
 * in email addresses  
 */  
public class EmailValidation {  
    public static void main(String[] args)  
        throws Exception {  
  
        String input = "@sun.com";  
        //Checks for email addresses starting with  
        //inappropriate symbols like dots or @ signs.  
        Pattern p = Pattern.compile("^\\.|^\\|@");  
        Matcher m = p.matcher(input);  
        if (m.find())  
            System.err.println("Email addresses don't start" +  
                " with dots or @ signs.");  
        //Checks for email addresses that start with  
        //www. and prints a message if it does.  
        p = Pattern.compile("^www\\.");  
        m = p.matcher(input);  
        if (m.find()) {  
            System.out.println("Email addresses don't start" +  
                " with \"www.\", only web pages do.");  
        }  
        p = Pattern.compile("[^A-Za-z0-9\\.\\|@_\\|~#]+");  
        m = p.matcher(input);  
        StringBuffer sb = new StringBuffer();  
        boolean result = m.find();  
        boolean deletedIllegalChars = false;  
  
        while(result) {  
            deletedIllegalChars = true;  
            m.appendReplacement(sb, "");  
            result = m.find();  
        }  
    }  
}
```

```

    }

    // Add the last segment of input to the new String
    m.appendTail(sb);

    input = sb.toString();

    if (deletedIllegalChars) {
        System.out.println("It contained incorrect characters" +
            " , such as spaces or commas.");
    }
}
}
}

```

Removing Control Characters from a File

```

/* This class removes control characters from a named
 * file.
 */
import java.util.regex.*;
import java.io.*;

public class Control {
    public static void main(String[] args)
        throws Exception {

        //Create a file object with the file name
        //in the argument:
        File fin = new File("fileName1");
        File fout = new File("fileName2");
        //Open and input and output stream
        FileInputStream fis =
            new FileInputStream(fin);
        FileOutputStream fos =
            new FileOutputStream(fout);

        BufferedReader in = new BufferedReader(
            new InputStreamReader(fis));
        BufferedWriter out = new BufferedWriter(
            new OutputStreamWriter(fos));

        // The pattern matches control characters
        Pattern p = Pattern.compile("{cntrl}");
    }
}

```

```

Matcher m = p.matcher("");
String aLine = null;
while((aLine = in.readLine()) != null) {
    m.reset(aLine);
    //Replaces control characters with an empty
    //string.
    String result = m.replaceAll("");
    out.write(result);
    out.newLine();
}
in.close();
out.close();
}
}

```

File Searching

```

/*
 * Prints out the comments found in a .java file.
 */
import java.util.regex.*;
import java.io.*;
import java.nio.*;
import java.nio.charset.*;
import java.nio.channels.*;

public class CharBufferExample {
    public static void main(String[] args) throws Exception {
        // Create a pattern to match comments
        Pattern p =
            Pattern.compile("//.*$", Pattern.MULTILINE);

        // Get a Channel for the source file
        File f = new File("Replacement.java");
        FileInputStream fis = new FileInputStream(f);
        FileChannel fc = fis.getChannel();

        // Get a CharBuffer from the source file
        ByteBuffer bb =
            fc.map(FileChannel.MAP_RO, 0, (int)fc.size());
        Charset cs = Charset.forName("8859_1");
        CharsetDecoder cd = cs.newDecoder();
        CharBuffer cb = cd.decode(bb);
    }
}

```

```

// Run some matches
Matcher m = p.matcher(cb);
while (m.find())
    System.out.println("Found comment: "+m.group());
}
}

```

Conclusion

Pattern matching in the Java programming language is now as flexible as in many other programming languages. Regular expressions can be put to use in applications to ensure data is formatted correctly before being entered into a database, or sent to some other part of an application, and they can be used for a wide variety of administrative tasks. In short, you can use regular expressions anywhere in your Java programming that calls for pattern matching.

Control Statements:

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. In Java control statements are categorized into selection control statements, iteration control statements and jump control statements.

- **Java's Selection Statements:** Java supports two selection statements: if and switch. These statements allow us to control the flow of program execution based on condition.

o if Statement: if statement performs a task depending on whether a condition is true or false.

Syntax: 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.

Program : Write a program to find biggest of three numbers.

```

//Biggest of three numbers

class BiggestNo

```



```

{ public static void main(String args[])
{ int a=5,b=7,c=6;
  if ( a > b && a>c)
    System.out.println ("a is big");
  else if ( b > c)
    System.out.println ("b is big");
  else
    System.out.println ("c is big");
}
}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac BiggestNo.java
D:\JQR>java BiggestNo
b is big
D:\JQR>_

```

o Switch Statement: When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax: switch (expression)

```

{ case value1: //statement sequence
  break;
  case value2: //statement sequence
  break;
  .....
  case valueN: //statement sequence
  break;
  default: //default statement sequence

```

```
}
```

Here, depending on the value of the expression, a particular corresponding case will be executed.

Program : Write a program for using the switch statement to execute a particular task depending on color value.

```
//To display a color name depending on color value
```

```
class ColorDemo
```

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

```
{ char color = 'r';
```

```
switch (color)
```

```
{ case 'r': System.out.println ("red"); break;
```

```
case 'g': System.out.println ("green"); break;
```

```
case 'b': System.out.println ("blue"); break;
```

```
case 'y': System.out.println ("yellow"); break;
```

```
case 'w': System.out.println ("white"); break;
```

```
default: System.out.println ("No Color Selected");
```

```
}
```

```
}
```

```
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac ColorDemo.java
D:\JQR>java ColorDemo
red
D:\JQR>
```

- **Java's Iteration Statements:** Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

o **while Loop:** while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.

Syntax: while (condition)

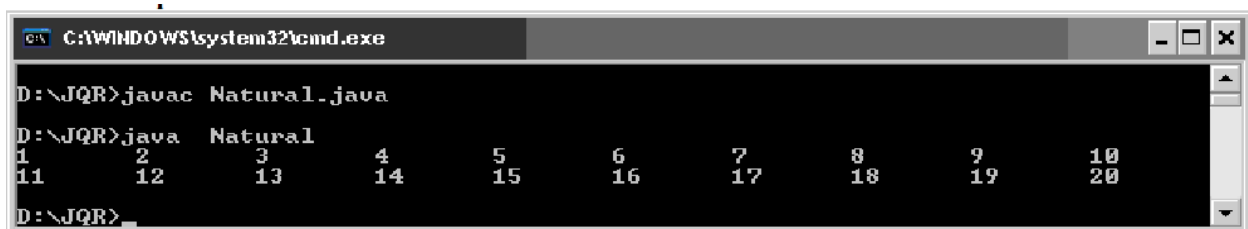
```
{  
    statements;  
}
```

Program : Write a program to generate numbers from 1 to 20.

//Program to generate numbers from 1 to 20.

```
class Natural  
{ public static void main(String args[])  
  { int i=1;  
    while (i <= 20)  
    { System.out.print (i + "\t");  
      i++;  
    }  
  }  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
D:\JQR>javac Natural.java  
D:\JQR>java Natural  
1 2 3 4 5 6 7 8 9 10  
11 12 13 14 15 16 17 18 19 20  
D:\JQR>
```

o **do...while Loop:** do...while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do...while loop is also called as exit control loop.

Syntax: do

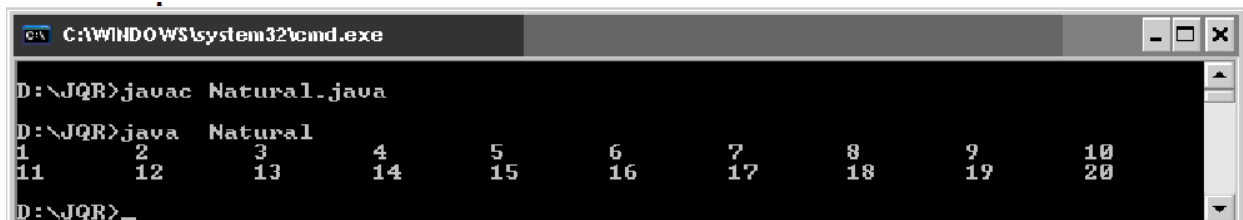
```
{
    statements;
} while (condition);
```

Program : Write a program to generate numbers from 1 to 20.

//Program to generate numbers from 1 to 20.

```
class Natural
{
    public static void main(String args[])
    {
        int i=1;
        do
        {
            System.out.print (i + "\t");
            i++;
        } while (i <= 20);
    }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Natural.java
D:\JQR>java Natural
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
D:\JQR>
```

o for Loop: The for loop is also same as do...while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)

```
{
    statements;
}
```

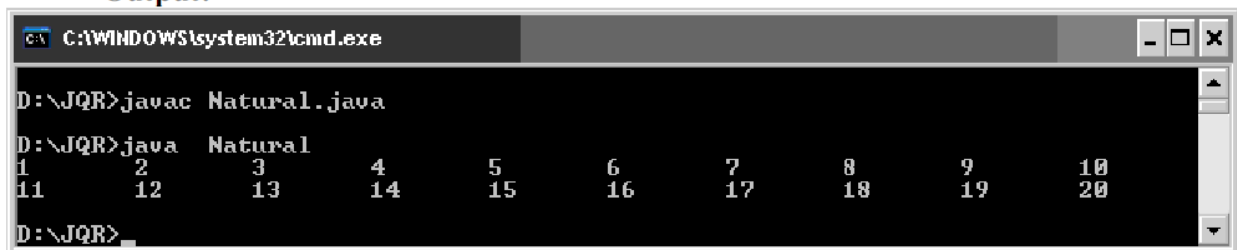
Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
```

```
class Natural
{
    public static void main(String args[])
    {
        int i;
        for (i=1; i<=20; i++)
            System.out.print (i + "\t");
    }
}
```

Output:



```

D:\JQR>javac Natural.java
D:\JQR>java Natural
1      2      3      4      5      6      7      8      9      10
11     12     13     14     15     16     17     18     19     20
D:\JQR>
```

· **Java's Jump Statements:** Java supports three jump statements: break, continue and return.

These statements transfer control to another part of the program.

o **break:**

Ø break can be used inside a loop to come out of it.

Ø break can be used inside the switch block to come out of the switch block.

Ø break can be used in nested blocks to go to the end of a block. Nested blocks represent a block written within another block.

Syntax: break; (or) break label;//here label represents the name of the block.

Program : Write a program to use break as a civilized form of goto.

```
//using break as a civilized form of goto
```

```
class BreakDemo
{
    public static void main (String args[])
```

```

{ boolean t = true;
  first:
{
  second:
{
  third:
{
    System.out.println ("Before the break");
    if (t) break second; // break out of second block
    System.out.println ("This won't execute");
  }
  System.out.println ("This won't execute");
}
  System.out.println ("This is after second block");
}
}
}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac BreakDemo.java
D:\JQR>java BreakDemo
Before the break
This is after second block
D:\JQR>

```

o continue: This statement is useful to continue the next repetition of a loop/ iteration. When continue is executed, subsequent statements inside the loop are not executed.

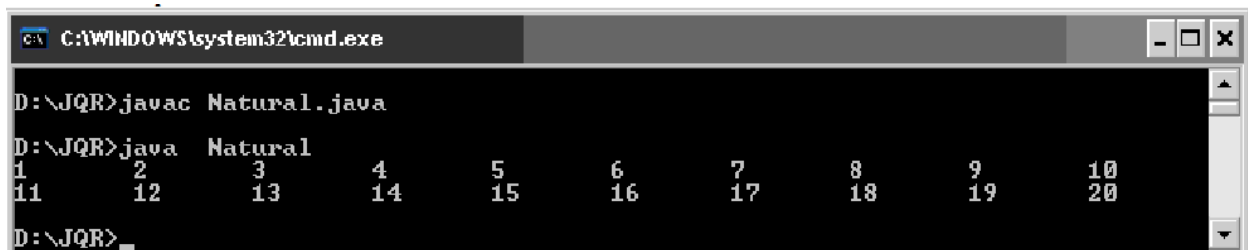
Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
```

```
class Natural
{
    public static void main (String args[])
    {
        int i=1;
        while (true)
        {
            System.out.print (i + "\t");
            i++;
            if (i <= 20 )
                continue;
            else
                break;
        }
    }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Natural.java
D:\JQR>java Natural
1      2      3      4      5      6      7      8      9      10
11     12     13     14     15     16     17     18     19     20
D:\JQR>
```

o **return statement:**

Ø return statement is useful to terminate a method and come back to the calling method.

Ø return statement in main method terminates the application.

Ø return statement can be used to return some value from a method to a calling method.

Syntax: return;

(or)

return value; // value may be of any type

Program : Write a program to demonstrate return statement.

```
//Demonstrate return  
  
class ReturnDemo  
{ public static void main(String args[])  
  { boolean t = true;  
    System.out.println (“Before the return”);  
    if (t)  
      return;  
    System.out.println (“This won’t execute”);  
  }  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
D:\JQR>javac ReturnDemo.java  
D:\JQR>java ReturnDemo  
Before the return  
D:\JQR>
```

Note: goto statement is not available in java, because it leads to confusion and forms infinite loops.

Concepts of classes, objects:

The class is at the core of Java. **It is the logical construct** upon which the entire Java language is built because it defines the shape and nature of an object.

As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java.

Class Fundamentals:

The classes created in the preceding chapters primarily exist simply to encapsulate the main() method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new **data type**, defined, this new type can be used to **create objects of that type**.

Thus, **a class is a template for an object**, and **an object is an instance of a class**.

Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The General Form of a Class:

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is **declared** by use of the **class keyword**. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

The data, or variables, defined within a class are called **instance variables**. The code is contained within methods. Collectively, the methods and variables defined within a class are called **members of the class**. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

All methods have the same general form as `main()`, which we have been using thus far. However, most methods will not be specified as `static` or `public`.

Notice that the general form of a class does not specify a `main()` method. Java classes do not need to have a `main()` method. You only specify one if that class is the starting point for your program. Further, applets don't require a `main()` method at all.

NOTE: C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large .java files, since any class must be entirely defined in a single source file. This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;  
public int gear;  
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as `public` or `private`.
2. The field's type.
3. The field's name.

The fields of Bicycle are named `cadence`, `gear`, and `speed` and are all of data type integer (`int`). The `public` keyword identifies these fields as public members, accessible by any object that can access the class.

A Simple Class:

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth.

```
// This program declares two Box objects.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        // compute volume of second box
```

```
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

Volume is 3000.0

Volume is 162.0

- As you can see, mybox1's data is completely separate from the data contained in mybox2.
- As stated, a class defines a new type of data. In this case, the new data type is called Box.
- You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

- After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.
- As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
- Thus, every Box object will contain its own copies of the instance variables width, height, and depth.
- To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

For example, to assign the width variable of mybox the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100.

- In general, you use the dot operator to access both the instance variables and the methods within an object.
- You should call the file that contains this program BoxDemo2.java, because the main() method is in the class called BoxDemo2, not the class called Box.
- When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo2.
- The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo2 class to actually be in the same source

file. You could put each class in its own file, called `Box.java` and `BoxDemo2.java`, respectively.

To run this program, you must execute `BoxDemo2.class`. When you do, you will see the following output:

Volume is 3000.0

Volume is 162.0

- As stated earlier, each object has its own copies of the instance variables.

This means that if you have two `Box` objects, each has its own copy of `depth`, `width`, and `height`. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

Declaring Objects

An object is an instance of a class. An object is known by a name and every object contains a state. The state is determined by the values of attributes (variables). The state of an object can be changed by calling methods on it. The sequence of state changes represents the behavior of the object.

An object is a software entity (unit) that combines a set of data with a set of operations to manipulate that data.

As just explained, **when you create a class, you are creating a new data type**. You can use this type to declare objects of that type.

However, **obtaining objects of a class is a two-step process**.

- **First**, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- **Second**, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the `new` operator.
- **Declaration**: The code set in variable declarations that associate a variable name with an object type.
- **Instantiation**: The `new` keyword is a Java operator that creates the object.
- **Initialization**: The `new` operator is followed by a call to a constructor, which initializes the new object.

The `new` operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by `new`. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type `Box`:

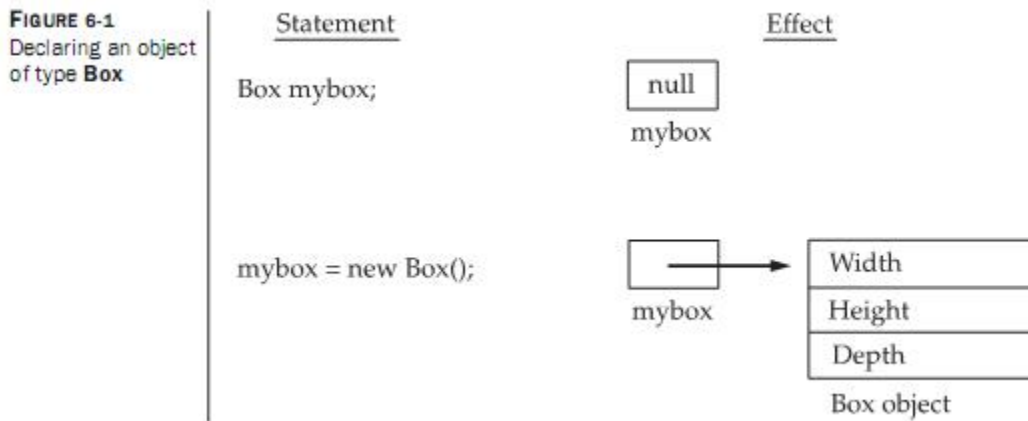
```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.

Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted in Figure.



NOTE: Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java’s safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

Assigning Object Reference Variables:

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

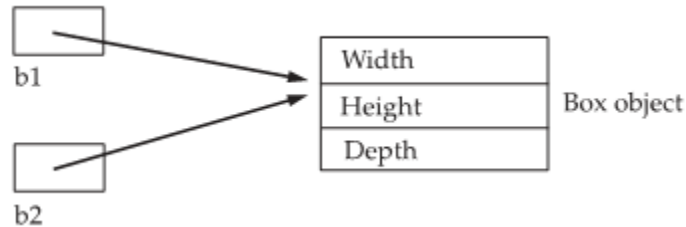
```
Box b1 = new Box();
```

```
Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original

object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although b1 and b2 both refer to the same object, they are not linked in any other way.

For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2. For example:

```
Box b1 = new Box();
```

```
Box b2 = b1;
```

```
// ...
```

```
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.

REMEMBER: When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Description2:

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the CreateObjectDemo program creates an object and assigns it to a variable:

```
Point originOne = new Point(23, 94);
```

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the Point class, and the second and third lines each create an object of the Rectangle class.

Each of these statements has three parts (discussed in detail below):

1. **Declaration:** The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation:** The new keyword is a Java operator that creates the object.

3. **Initialization:** The new operator is followed by a call to a constructor, which initializes the new object.

Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

```
type name;
```


This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

```
Point originOne;
```

If you declare `originOne` like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the new operator, as described in the next section. You must assign an object to `originOne` before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, `originOne`, plus a reference pointing to nothing):

```
originOne 
```

Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point(23, 94);
```


The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:
int height = new Rectangle().height;

This statement will be discussed in the next section.

Initializing an Object

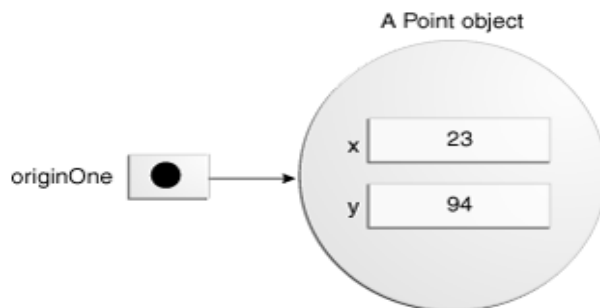
Here's the code for the Point class:

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    //constructor  
    public Point(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

```
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:



Here's the code for the Rectangle class, which contains four constructors:

```
public class Rectangle {  
    public int width = 0;  
    public int height = 0;  
    public Point origin;
```

```

// four constructors
public Rectangle() {
    origin = new Point(0, 0);
}
public Rectangle(Point p) {
    origin = p;
}
public Rectangle(int w, int h) {
    origin = new Point(0, 0);
    width = w;
    height = h;
}
public Rectangle(Point p, int w, int h) {
    origin = p;
    width = w;
    height = h;
}

// a method for moving the rectangle
public void move(int x, int y) {
    origin.x = x;
    origin.y = y;
}

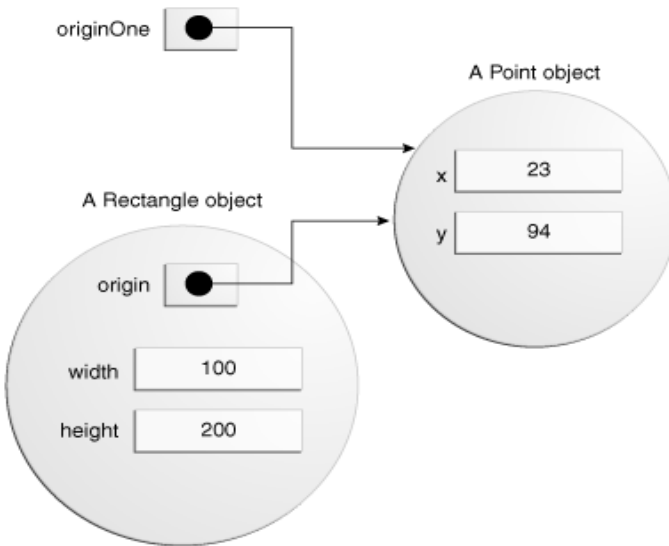
// a method for computing the area
// of the rectangle
public int getArea() {
    return width * height;
}
}

```

Each constructor lets you provide initial values for the rectangle's size and width, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the same Point object—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a *no-argument constructor*:

```
Rectangle rect = new Rectangle();
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the *default constructor*. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the Rectangle class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, width and height are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the CreateObjectDemo class is outside the code for the Rectangle class. So to refer to the origin, width, and height fields within the Rectangle object named rectOne, the CreateObjectDemo class must use the names rectOne.origin, rectOne.width, and rectOne.height, respectively. The program uses two of these names to display the width and the height of rectOne:

```
System.out.println("Width of rectOne: "
    + rectOne.width);
System.out.println("Height of rectOne: "
    + rectOne.height);
```

Attempting to use the simple names width and height from the code in the CreateObjectDemo class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about rectTwo. Objects of the same type have their own copy of the same instance fields. Thus, each Rectangle object has fields named origin, width, and height. When you access an instance field through an object reference, you reference that particular object's field. The two objects rectOne and rectTwo in the CreateObjectDemo program have different origin, width, and height fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. **Recall** that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
```

or:

```
objectReference.methodName();
```

The Rectangle class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the `CreateObjectDemo` code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());  
...  
rectTwo.move(40, 72);
```

The first statement invokes `rectOne`'s `getArea()` method and displays the results. The second line moves `rectTwo` because the `move()` method assigns new values to the object's `origin.x` and `origin.y`.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The `new` operator returns an object reference, so you can use the value returned from `new` to invoke a new object's methods:

```
new Rectangle(100, 50).getArea()
```

The expression `new Rectangle(100, 50)` returns an object reference that refers to a `Rectangle` object. As shown, you can use the dot notation to invoke the new `Rectangle`'s `getArea()` method to compute the area of the new rectangle.

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

Summary of Creating and Using Classes and Objects

A class declaration names the class and encloses the class body between braces. The class name can be preceded by modifiers. The class body contains fields, methods, and constructors for the class. A class uses fields to contain state information and uses methods to implement behavior. Constructors that initialize a new instance of a class use the name of the class and look like methods without a return type.

You control access to classes and members in the same way: by using an access modifier such as `public` in their declaration.

You specify a class variable or a class method by using the `static` keyword in the member's declaration. A member that is not declared as `static` is implicitly an instance member. Class variables are shared by all instances of a class and can be accessed through the class name as well as an instance reference. Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.

You create an object from a class by using the `new` operator and a constructor. The `new` operator returns a reference to the object that was created. You can assign the reference to a variable or use it directly.

Instance variables and methods that are accessible to code outside of the class that they are declared in can be referred to by using a qualified name. The qualified name of an instance variable looks like this:

objectReference.variableName

The qualified name of a method looks like this:

objectReference.methodName(argumentList)

or:

objectReference.methodName()

The garbage collector automatically cleans up unused objects. An object is unused if the program holds no more references to it. You can explicitly drop a reference by setting the variable holding the reference to `null`.

Methods :

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                             double length, double grossTons) {  
    //do the calculation here  
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, `()`, and a body between braces, `{}`.

More generally, method declarations have six components, in order:

1. Modifiers—such as `public`, `private`, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or `void` if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized.

Here are some examples:

```
run  
runFast
```

```
getBackground  
getFinalData  
compareTo  
setX  
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

```
return returnValue;
```

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The `getArea()` method in the `Rectangle` [Rectangle](#) class that was discussed in the sections on objects returns an integer:

```
// a method for computing the area of the rectangle  
public int getArea() {  
    return width * height;  
}
```

This method returns the integer that the expression `width*height` evaluates to.

The `getArea` method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate `Bicycle` objects, we might have a method like this:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                             Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, `drawString`, `drawInteger`, `drawFloat`, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named `draw`, each of which has a different parameter list.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, `draw(String s)` and `draw(int i)` are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Constructors:

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like `setDim()`, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the `new` operator completes. Constructors look a little strange because they have no return type, not even `void`. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the `Box` example so that the dimensions of a box are automatically

initialized when an object is constructed. To do so, replace `setDim()` with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box

to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
```

```
dimensions of a box.
```

```
*/
```

```
class Box {
```

```
    double width;
```

```
    double height;
```

```
    double depth;
```

```
// This is the constructor for Box.

Box() {

System.out.println("Constructing Box");

width = 10;

height = 10;

depth = 10;

}

// compute and return volume

double volume() {

return width * height * depth;

}

}

class BoxDemo6 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();
```

```
System.out.println("Volume is " + vol);  
  
}  
  
}
```

When this program is run, it generates the following results:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

As you can see, both `mybox1` and `mybox2` were initialized by the `Box()` constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both `mybox1` and `mybox2` will have the same volume. The `println()` statement inside `Box()` is for the sake of illustration only.

Most constructors will not display anything. They will simply initialize an object. Before moving on, let's reexamine the `new` operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname( );
```

Now **you can understand why the parentheses are needed after the class name**. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

`new Box()` is calling the `Box()` constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of `Box` that did not define a constructor. The default constructor automatically initializes all instance variables to zero.

The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the `Box()` constructor in the preceding example does initialize a `Box` object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct `Box` objects of various dimensions. The easy solution is to add parameters to the constructor.

As you can probably guess, this makes them much more useful. For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.

```
/* Here, Box uses a parameterized constructor to initialize the dimensions of a box.*/
```

```
class Box {  
  
    double width;  
  
    double height;  
  
    double depth;  
  
    // This is the constructor for Box.  
  
    Box(double w, double h, double d) {  
  
        width = w;  
  
        height = h;  
  
        depth = d;  
  
    }  
  
    // compute and return volume  
  
    double volume() {  
  
        return width * height * depth;  
  
    }  
  
    }  
  
    class BoxDemo7 {  
  
        public static void main(String args[]) {  
  
            // declare, allocate, and initialize Box objects  
  
            Box mybox1 = new Box(10, 20, 15);  
  
            Box mybox2 = new Box(3, 6, 9);
```

```
double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}
```

The output from this program is shown here:

```
Volume is 3000.0
```

```
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor.

For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the Box() constructor when new creates the object. Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

Overloading Constructors:

Since Box() requires three arguments, it's an error to call it without them.

This raises some important questions.

What if you simply wanted a box and did not care (or know)

what its initial dimensions were? Or,

what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?

As the Box class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described. Here is a program that contains an improved version of Box that does just that:

```
/* Here, Box defines three constructors to initialize  
the dimensions of a box various ways.
```

```
*/
```

```
class Box {
```

```
double width;
```

```
double height;
```

```
double depth;
```

```
// constructor used when all dimensions specified
```

```
Box(double w, double h, double d) {
```

```
width = w;
```

```
height = h;
```

```
depth = d;
```

```
}
```

```
// constructor used when no dimensions specified
```

```
Box() {
```

```
width = -1; // use -1 to indicate
```

```
height = -1; // an uninitialized
```

```
depth = -1; // box
```

```
}
```

```
// constructor used when cube is created
```

```
Box(double len) {
```

```
width = height = depth = len;
```

```
}
```

```
// compute and return volume
```

```

double volume() {
return width * height * depth;
}
}

class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box();

Box mycube = new Box(7);

double vol;

// get volume of first box
vol = mybox1.volume();

System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();

System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();

System.out.println("Volume of mycube is " + vol);

}

}

```

The output produced by this program is shown here:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

As you can see, the proper overloaded constructor is called based upon the parameters specified when new is executed.

parameter passing:

In general, **there are two ways that a computer language can pass an argument to a subroutine.**

The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

As you will see, Java uses both approaches, depending upon what is passed.

In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
// Primitive types are passed by value.

class Test {

void meth(int i, int j) {

i *= 2;

j /= 2;

}

}

class CallByValue {

public static void main(String args[]) {

Test ob = new Test();

int a = 15, b = 20;

System.out.println("a and b before call: " +

a + " " + b);

ob.meth(a, b);

System.out.println("a and b after call: " +

a + " " + b);
```

```
}  
}
```

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

As you can see, the operations that occur inside meth() have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.

Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

For example, consider the following program:

```
// Objects are passed by reference.  
  
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // pass an object  
    void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2;  
    }  
}  
  
class CallByRef {  
    public static void main(String args[]) {
```

```
Test ob = new Test(15, 20);  
System.out.println("ob.a and ob.b before call: " +  
ob.a + " " + ob.b);  
ob.meth(ob);  
System.out.println("ob.a and ob.b after call: " +  
ob.a + " " + ob.b);  
}  
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
```

```
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside meth() have affected the object used as an argument.

As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Note: When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

Recursion:

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself.

A method that calls itself is said to be recursive.

The classic **example** of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

For example, 3 factorial is $1 \times 2 \times 3$, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
```

```
class Factorial {  
    // this is a recursive method  
    int fact(int n) {  
        int result;  
        if(n==1) return 1;  
        result = fact(n-1) * n;  
        return result;  
    }  
}  
  
class Recursion {  
    public static void main(String args[]) {  
        Factorial f = new Factorial();  
        System.out.println("Factorial of 3 is " + f.fact(3));  
        System.out.println("Factorial of 4 is " + f.fact(4));  
        System.out.println("Factorial of 5 is " + f.fact(5));  
    }  
}
```

The output from this program is shown here:

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

If you are unfamiliar with recursive methods, then the operation of `fact()` may seem a bit confusing. Here is how it works. When `fact()` is called with an argument of 1, the function returns 1; otherwise, it returns the product of `fact(n-1)*n`. To evaluate this expression, `fact()` is called with `n-1`. This process repeats until `n` equals 1 and the calls to the method begin returning.

To better understand how the `fact()` method works, let's go through a short example.

When you compute the factorial of 3, the first call to `fact()` will cause a second call to be made with an argument of 2. This invocation will cause `fact()` to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of `n` in the second invocation). This result (which is 2) is then returned to the original invocation of `fact()` and multiplied by 3 (the original value of `n`). This yields the answer, 6. You might find it interesting to insert `println()` statements into `fact()`, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an `if` statement somewhere to force the method to return without the recursive call being executed. If you don’t do this, once you call the method, it will never return. This is a very common error in working with recursion.

Use `println()` statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method `printArray()` prints the first `i` elements in the array values.

```
// Another example that uses recursion.
```

```
class RecTest {  
    int values[];  
    RecTest(int i) {  
        values = new int[i];
```

```
}  
// display array -- recursively  
void printArray(int i) {  
    if(i==0) return;  
    else printArray(i-1);  
    System.out.println "[" + (i-1) + " ] " + values[i-1]);  
}  
}  
class Recursion2 {  
    public static void main(String args[]) {  
        RecTest ob = new RecTest(10);  
        int i;  
        for(i=0; i<10; i++) ob.values[i] = i;  
        ob.printArray(10);  
    }  
}
```

[0] 0

[1] 1

[2] 2

[3] 3

[4] 4

[5] 5

[6] 6

[7] 7

[8] 8

This program generates the following output: