Specially Prepared for Engineering Students

Learning JAVA in a Magical Way

By

H. Ateeq Ahmed, **M.Tech.,**

Assistant Professor of CSE,

Kurnool.

As per JNTUA R09 CSE Syllabus

ACKNOWLEDGEMENT

First I thank Almighty God for giving me the knowledge to learn and teach various students.

It's my privilege to thanks my parents as without their right guidance and support, these book will be a dream for me.

Finally, I thank my colleagues and friends for helping me during tough period of time.

"Interest & Focus are two KEYWORDS of a perfect programmer"

"The goal of this book is to find the next JAVA Programmer in YOU in a magical way…"

H. Ateeq Ahmed, M.Tech.,

Mobile no: 9948378994,

E-mail ID: ateeqh25@gmail.com.

CONTENTS

UNIT-II

History of Java (Genesis of Java):

Creation of Java:

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak" but was renamed "Java" in 1995.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is also a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of 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.

Java Applets and Applications

Java can be used to create two types of programs: applications and applets. An *application* is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer language. Rather, it is Java's ability to create applets that makes it important. An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an *intelligent program*, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over.

Java's Magic: The Bytecode

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

Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.

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

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

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis.

The Java Buzzwords

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

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

Let us Examine each of them briefly.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner.

Security

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

When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java.

Portability

As discussed earlier, many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry

about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance.

Distributed

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

Dynamic

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

Data Types:

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. 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.

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, including C/C++, support both signed and unsigned integers.

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 us look at each type of Integers briefly.

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 To 127.

Eg:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767.

Eg:

```
Short b,c;
```

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.

Eg:

```
int b,c;
```

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.

Eg:

```
long b,c;
```

Floating-Point Types

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

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.

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.

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

   System.out.println("Area of circle is " + a);
  }
}
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integer type that 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.

Example Program

```
// 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
```

Booleans

Java has a simple 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, such as **a** < **b**.

Example Program

```
// 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);

  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
```

Variables

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

Declaring a Variable

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

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

Examples:

```
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.
```

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.

Example Program

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

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. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

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.

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

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.

One-Dimensional Arrays

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

type var-name[ ];

For example, the following declares an array named **month_days** with the type "array of int":

```
int month_days[];
```

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

you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

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

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to **month_days**.

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

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

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];
```

Example Program

```
// Demonstrate One dimensional array
Class Array1
{
Public static void main(String arg[])
{

int month_days[]=new int[3];

month_days[0]=31;

month_days[1]=28;

month_days[2]=31;
System.out.println("January has "+month_days[0]+" days.");
}
}
```

Expected output:

```
January has 31 days.
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

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

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1.

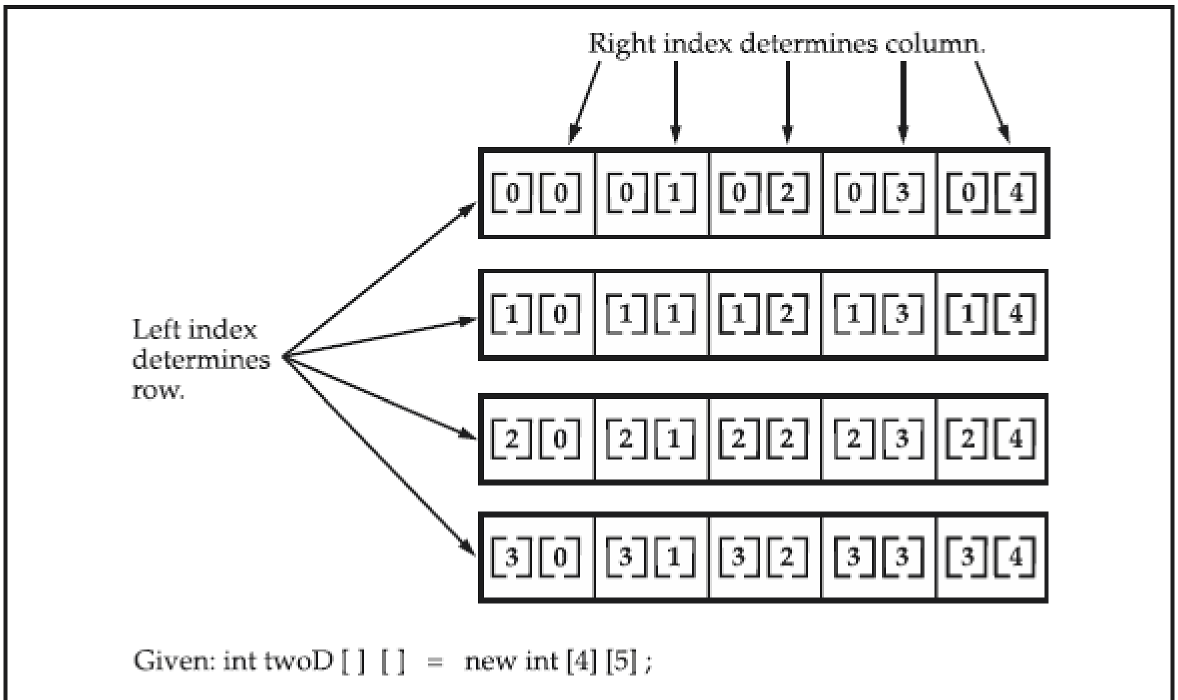


Figure 3-1. *A conceptual view of a 4 by 5, two-dimensional array*

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];
```

<u>Procedure for Compiling and Executing a Java program</u>

<u>Step-1</u>
Java Compilation:
C:\Program Files\Java\jdk1.5.0_05\bin>**javac String1.java**

<u>Step-2</u>
Java Execution:
C:\Program Files\Java\jdk1.5.0_05\bin>**java String1**

Output:
Hi Every one! This is Ateeq Ahmed, Asst. Professor of CSE Dept. & I am dealing you Object Oriented Programming...

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.

Arithmetic Operators

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

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

The Bitwise Operators

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

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

Relational Operators

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

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

Boolean Logical Operators

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

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

Control Statements:

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

Java's Selection Statements

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

if

It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

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

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

Example Program

```
class Mytest
{
public static void main(String z[])
{
int age=20;

if(age>=18)
System.out.println("Eligible for voting!");
else
System.out.println("Not Eligible");
}
}
```

Expected Output

```
Eligible for voting!
```

Nested ifs

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

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

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. It looks like this:

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

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

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

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

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

Example Program

```
class Switch1
{
public static void main(String z[])
{
int i=2;
        switch(i)
        {
        case 1:
        System.out.println("One");
        break;

        case 2:
        System.out.println("Two");
        break;

        default:
        System.out.println("Invalid Choice!");
        }


}
}
```

Iteration Statements

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

while

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

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

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

Example Program

```
class While1
{
public static void main(String z[])
{
int i=1;
        while(i<=5)
        {
        System.out.println(i);
        i++;
        }
}
}
```

do-while

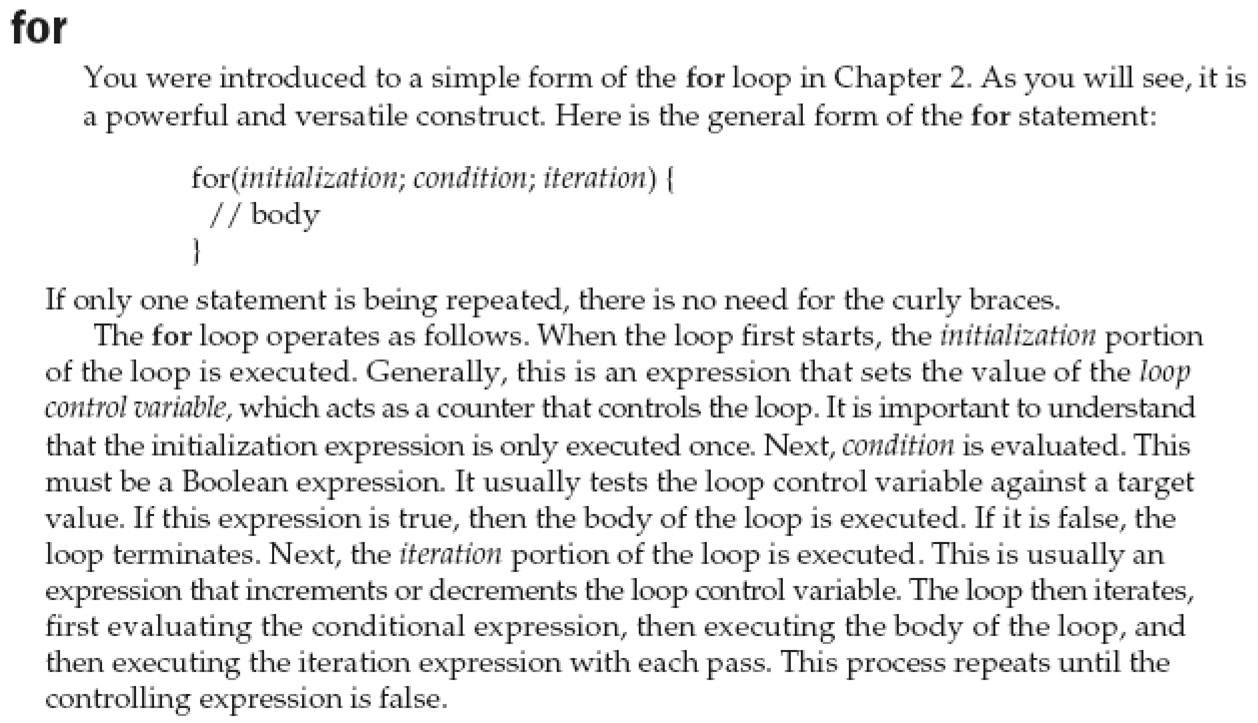
As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In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

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

Example Program

```
class Dowhile
{
public static void main(String z[])
{
int i=1;
        do
        {
        System.out.println(i);
        i++;
        } while(i<=5);

}
}
```



for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct. Here is the general form of the **for** statement:

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

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Example Program

```
class For1
{
public static void main(String z[])
{
int i;
        for(i=1;i<=5;i++)
        {
        System.out.println(i);
        }
}
}
```

Declaring Loop Control Variables Inside the for Loop

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

Example Program

```
class For2
{
public static void main(String z[])
{
        for(int i=1;i<=5;i++)                    // i is declared inside for
        {
        System.out.println(i);
        }
}
}
```

Using the Comma

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

Example Program

```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

Jump Statements

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

Using break

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

Using break to Exit a Loop

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

Example Program

```
class Break1
{
public static void main(String z[])
{
int i;
        for(i=1;i<=5;i++)
        {
        if(i==3)
        break;
        System.out.println(i);
        }
        System.out.println("Loop Breaked!");
}
}
```

Expected Output:

```
1
2
Loop Breaked!
```

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement. Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner.

The general form of the labeled **break** statement is shown here:

break *label;*

Here, *label* is the name of a label that identifies a block of code. When this form of **break** executes, control is transferred out of the named block of code. The labeled block of code must enclose the **break** statement, but it does not need to be the immediately enclosing block.

Example Program

```
// Using break as a civilized form of goto.
class Break {
  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.");
    }
  }
}
```

Expected Output:

```
Before the break.
This is after second block.
```

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration.

Example Program

```
class Cont1
{
public static void main(String z[])
{
int i;
        for(i=1;i<=5;i++)
        {
        if(i==3)
        continue;
        System.out.println(i);
        }
}
}
```

Expected Output:

```
1
2
4
5
```

return

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

Example Program:

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;
    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Expected Output:

```
Before the return.
```

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 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.

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*

Example:

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

Example Program

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);
   }
 }
```

Expected Output:

```
Conversion of int to byte.
i and b 257 1
```

Class Fundamentals

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.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once 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

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. The 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.

Example Program

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}


// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

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

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

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

Expected Output:

```
Volume is 3000.0
```

Declaring Objects

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. 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**.

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

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

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

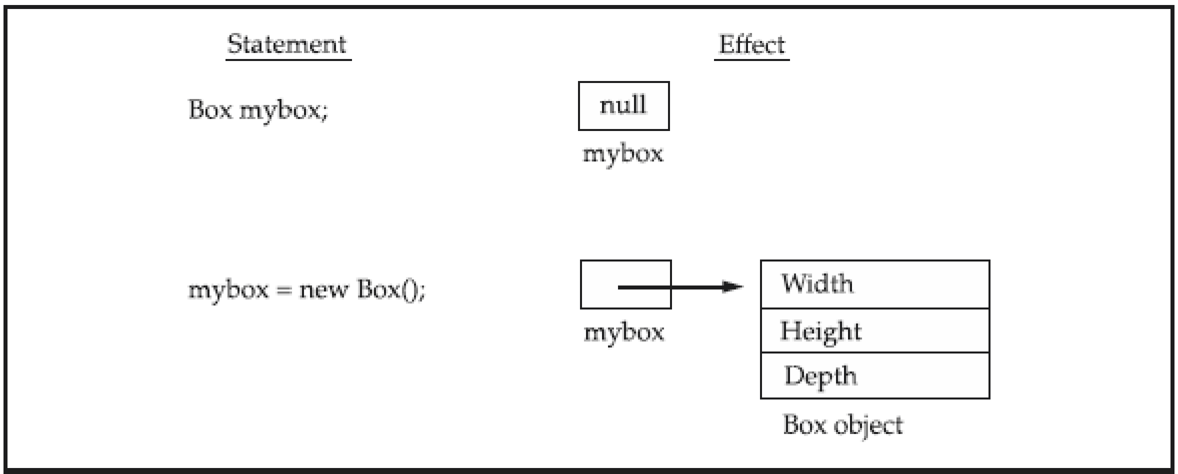


Figure 6-1. *Declaring an object of type* Box

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility.

This is the general form of a method:

```
type name(parameter-list) {
   // body of method
}
```

Example Programs

//Program on methods

```
class Box
{
double width,depth,height,vol;

        void volume()
        {
        vol=width*depth*height;
        }

        void display()
        {
        System.out.println("Volume is "+vol);
        }

}

class Method1
{
public static void main(String z[])
{
Box b=new Box();
b.width=2;
b.depth=2;
b.height=3;
b.volume();
b.display();

}
}
```

Expected Output:

```
Volume is 12.0
```

//Program on Paramaterised method

```
class Box
{
double width,depth,height,vol;

        void dimensions(int w,int d,int h)
        {
        width=w;
        depth=d;
        height=h;
        }

        void volume()
        {
        vol=width*depth*height;
        }

        void display()
        {
        System.out.println("Volume is "+vol);
        }

}

class Method2
{
public static void main(String z[])
{
Box b=new Box();

b.dimensions(2,2,2);
b.volume();
b.display();

}
}
```

Expected Output:

```
Volume is 8.0
```

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.

<u>Example Program</u>

```
//Program on Constructor
class Box
{
double width,depth,height,vol;

        Box()
        {
        width=3;
        depth=3;
        height=3;
        }
        void volume()
        {
        vol=width*depth*height;
        }
        void display()
        {
        System.out.println("Volume is "+vol);
        }

}

class Construct1
{
public static void main(String z[])
{
Box b=new Box();
b.volume();
b.display();
}
}
```

<u>Expected Output:</u>

```
Volume is 27.0
```

//Program on Paramaterised Constructor

```
class Box
{
double width,depth,height,vol;

        Box(int w,int d,int h)
        {
        width=w;
        depth=d;
        height=h;
        }

        void volume()
        {
        vol=width*depth*height;
        }

        void display()
        {
        System.out.println("Volume is "+vol);
        }

}

class Construct2
{
public static void main(String z[])
{
Box b=new Box(4,4,4);
b.volume();
b.display();
}
}
```

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

Example:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.*

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

It is important to understand that **finalize( )** is only called just prior to garbage collection.

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control.* Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.

Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access specifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

Example:

```
public int i;
private double j;
```

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1. *Class Member Access*

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java implements polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

Example Program

```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }


  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }
}
```

```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
  }
}
```

Expected Output:

```
No parameters
a: 10
a and b: 10 20
```

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods.

Example Program

//Program on constructor overloading

```
class Box
{
double w,d,h,vol;
	Box()
	{
	w=2;
	d=2;
	h=2;
	}

	Box(int w,int d,int h)
	{
	this.w=w;
	this.d=d;
	this.h=h;
	}

	void volume()
	{
	vol=w*d*h;
	}

	void display()
	{
	System.out.println("Volume="+vol);
	}
}

class Constover
{
```

```
public static void main(String z[])
{
Box b1=new Box();
Box b2=new Box(3,3,3);
b1.volume();
b1.display();

b2.volume();
b2.display();
}
}
```

Expected Output:

```
Volume=8.0
Volume=27.0
```

Parameters 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 method 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 method, 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.

Example Program

//Call by value

```
class Call
{
	void change(int x,int y)
	{
	x++;
	y++;
	System.out.println("Changed values are "+x+" "+y);
	}
}

class Callvalue
{
public static void main(String z[])
{
Call ob=new Call();
int a=10,b=20;
System.out.println("Values before call are "+a+" "+b);
ob.change(a,b);
System.out.println("Values after call are "+a+" "+b);
}
}
```

Expected Output:

```
Values before call are 10 20
Changed values are 11 21
Values after call are 10 20
```

When you pass an object to a method, the situation changes dramatically, because objects are passed 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:

Example Program

//Call by reference

```
class Call
{
int a,b;
         Call(int x,int y)
         {
         a=x;
         b=y;
         }

        void change(Call ob)
        {
        ob.a++;
        ob.b++;
        System.out.println("Changed values are "+ob.a+" "+ob.b);
        }
}

class Callreference
{
public static void main(String z[])
{
Call ob=new Call(10,20);
System.out.println("Values before call are "+ob.a+" "+ob.b);
ob.change(ob);
System.out.println("Values after call are "+ob.a+" "+ob.b);
}
}
```

Expected Output:

```
Values before call are 10 20
Changed values are 11 21
Values after call are 11 21
```

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 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

Example Program

```
// A simple example of recursion.
class Factorial {
  // this is a recursive function
  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));
  }
}
```

Expected Output:

```
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.

String Handling:

As is the case in most other programming languages, in Java a *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction.

However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.

One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, you call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Example Program

```
// Construct one String from another.
class MakeString {
  public static void main(String args[]) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

Expected Output:

```
Java
Java
```

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

int length( )

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

Example Program

//Concatenating long string using '+'

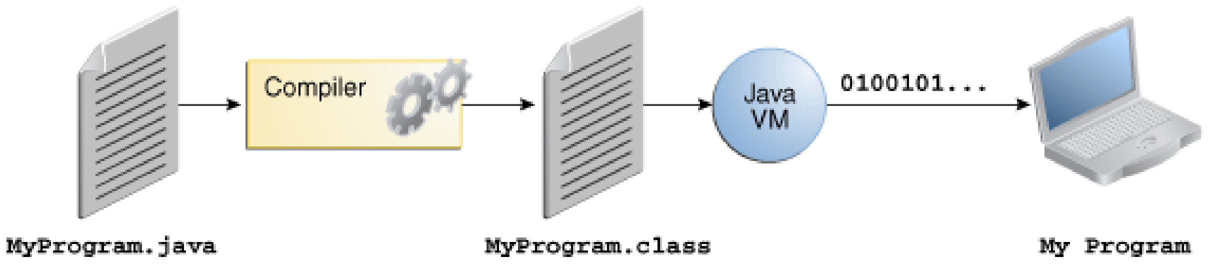
```
class String1
{
public static void main(String z[])
{
String str="Hi Every one!"+
        " This is Ateeq Ahmed,"+
        " Asst. Professor of CSE Dept."+
        " & I am dealing you Object Oriented Programming...";

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

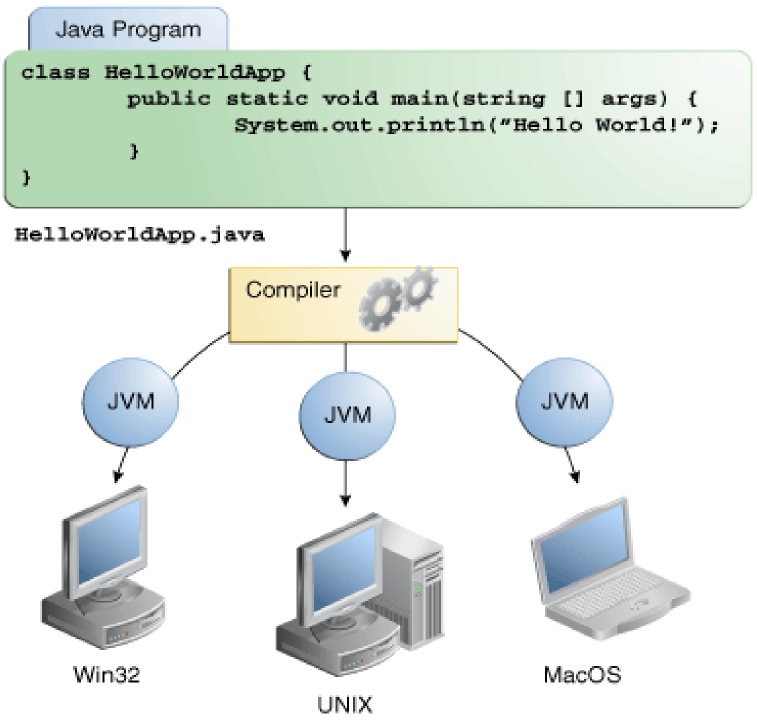
Expected Output:

```
Hi Every one! This is Ateeq Ahmed, Asst. Professor of CSE Dept. & I am dealing you Object Oriented
Programming...
```

Compilation & Execution of a Java program



- In the Java programming language, all source code is first written in plain text files ending with the **.java** extension.
- Those source files are then compiled into .class files by the **javac** compiler.
- A **.class** file does not contain code that is native to your processor; it instead contains ***bytecode*** — the machine language of the Java Virtual Machine[1] (Java VM).
- The java launcher tool then runs your application with an instance of the Java Virtual Machine.



```
class HelloWorldApp {
        public static void main(string [] args) {
                System.out.println("Hello World!");
        }
}
```

- Because the Java VM is available on many different operating systems, the same **.class** files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.

****

UNIT-III

Abstraction

- An essential element of object-oriented programming is *abstraction.*
- Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts.
- They think of it as a well-defined object with its own unique behavior.
- This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car.
- They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a whole.

Hierarchical Abstractions

- A powerful way to manage abstraction is through the use of hierarchical classifications.
- This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.
- From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.
- The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.
- Hierarchical abstractions of complex systems can also be applied to computer programs.
- The data from a traditional process-oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior.
- You can treat these objects as concrete entities that respond to messages telling them to *do something.* This is the essence of object-oriented programming.
- Object-oriented concepts form the heart of Java just as they form the basis for human understanding.
- It is important that you understand how these concepts translate into programs.

Thus object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project.

Subclass and Subtype

It is therefore useful to define two separate concepts:

- To say that A is a subclass of B merely asserts that A is formed using *inheritance.*
- To say that A is a subtype of B asserts that A *preserves* the meaning of all the operations in B.

It is possible to form subclasses that are not subtypes; and form subtypes that are not subclasses. The term subtype is used to refer to a subclass relationship in which the principle of substitution is maintained to distinguish such forms from the general subclass relationship.

Substitutability

"The principle of substitution says that if we have two classes, A and B, such that class B is a subclass of class A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect."

All object oriented languages will support the principle of substitution.
Most support this concept this concept in a straightforward way i.e. the parent class simple holds a value from the child class.

Inheritance

"Inheritance is the process by which an object of one class known as subclass acquires the properties of object of another class known as superclass."

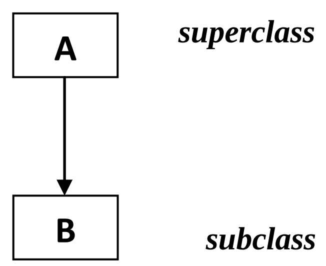
Types of Inheritance

The following are the various types of inheritance

- Single level Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

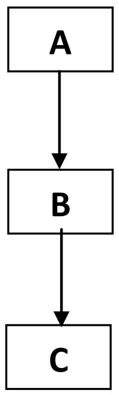
Single level Inheritance

The process of deriving a single class known as subclass from a single class known as superclass in known as single level inheritance.

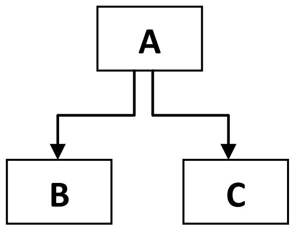


Multilevel Inheritance
The process of deriving a new subclass from the already existing subclass is known as Multilevel inheritance.

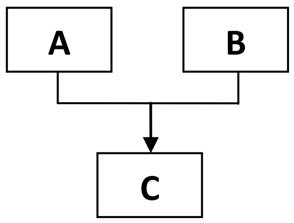
In this inheritance, a super class will have many levels of subclasses.



Hierarchical Inheritance
The process of deriving multiple subclasses from the same superclass is known as Hierarchical inheritance.



Multiple Inheritance
The process of deriving a single subclass from multiple superclasses is known as Multiple Inheritance.



Java doesn't support multiple inheritance through classes but it supports this concept by using interfaces.

Hybrid Inheritance
It is a combination of Hierarchical and Multiple inheritance.

Forms of Inheritance

Inheritance is used in a number of ways for different purposes.
Many of these types of inheritance are given their own special names.
The following are some of these specialized forms of inheritance.

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Let us discuss each of them briefly.

Subclassing for specialization

- It is the most common use of inheritance.
- In subclassing for specialization, the new class is a specialized form of the parent class or superclass but satisfies all the specifications of the parent class.
- Hence subclassing for specialization is the most common and popular form of inheritance in which a subclass is derived from the parent class.

Subclassing for specification

- Another frequent use of inheritance is to guarantee that classes maintain a certain common interface i.e. they implement same methods.
- This is a special case of subclassing for specialization, except that the subclasses are not refinements of an existing type but rather realization of an incomplete abstract class.
- In such cases, the parent class is sometimes known as an abstract class.

Subclassing for construction

- A class can inherit almost all of its desired functionality from a parent class.
- If the parent class is used as a *source* for behavior, but the child class has *no is-a* relationship to the parent, then we say the child class is using inheritance for construction.
- It is generally not a good idea to use subclassing for construction, since it can break the principle of substitutability, but nevertheless sometimes used practically.

Subclassing for extension

- If a child class generalizes or extends the parent class by providing *more functionality*, but does not override any method. It is called inheritance for generalization whereas subclassing for extension adds totally new abilities.

- Extension only adds new methods to those of the parent class.
- An example of subclassing for extension is a StringSet class that inherits from a generic Set class, but is specialized for holding string values.
- The child class doesn't change anything inherited from the parent, it simply *adds* new features.

Subclassing for limitation

- Subclassing for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class.
- If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it as inheritance for limitation.
- For example, you have an existing **List** data type that allows items to be inserted at either end, and you override methods allowing insertion at one end in order to create a **Stack**.
- Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice.

Subclassing for combination

- A common situation is a subclass that represents a combination of features from two or more parent classes.
- A teaching assistant, for example, may have characteristics of both a teacher and a student and can therefore logically behave as both.
- The ability of a class to inherit from two or more parent classes is known as Multiple Inheritance.

Benefits of Inheritance

The following are some of the important benefits of the proper use of inheritance.

- Software reusability
- Code sharing
- Consistency of interface
- Software components
- Rapid prototyping
- Information hiding

Let us examine each of them briefly.

Software Reusability

When behavior is inherited from another class, the code that provides the behavior does not have to be rewritten in the subclasses.

Code Sharing

- Code Sharing can occur on several levels with object oriented techniques.
- At one level, many users can use the same classes.
- Another example is a single parent class which can be shared by many number of subclasses.
- In simple terms, many users can share the same part of code which avoids rewriting of code.

Consistency of Interface

It states that, when two or more classes inherit from the same superclass, it is assured that the behavior they inherit will be the same in all cases.

Software Components

Inheritance provides programmers with the ability to construct reusable software components. The goal is to develop new applications with little actual coding

Rapid Prototyping

- When a software system is developed by using large number of reusable components then the development can be done very fast or rapid.
- Thus software systems can be generated more quickly and easily.

Information Hiding

- A programmer who uses a software component needs only to understand the nature of the component and its interface.
- It is not necessary for the programmer to have detailed information such as the techniques used to implement the component.

Costs of Inheritance

Although the benefits of inheritance in object oriented programming are great, but inheritance also have some disadvantages when it is not used in a correct way.

- Execution Speed
- Program Size
- Message Passing Overhead
- Program Complexity.

Member access rules

Refer **Unit-II** Topic: **Access Control**

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

Example Program

```
class A
{
private int a=10;
public int b=20;
protected int c=30;
int d=40;       // d is a no modifier variable
}

class B extends A
{
        void display()
        {
        System.out.println("a="+a);  // generates an error becoz a is declared as private
        System.out.println("b="+b);
        System.out.println("c="+c);
        System.out.println("d="+d);
        }
}

class Access
{
public static void main(String z[])
{
B ob=new B();
ob.display();
}
}
```

Super keyword

"A super is a keyword in java used to refer to the super class."

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

The keyword **super** has two uses

- The first calls the superclass constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to call superclass constructor

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass constructor.

Example Program

```
//to call super class constructor
class A
{
        A(int x)
        {
        x++;
        System.out.println("x="+x);
        }
}
class B extends A
{
        B(int y)
        {
        super(y);
        System.out.println("y="+y);
        }

}
class Super1
{
public static void main(String z[])
{
B b=new B(10);
}
}
```

Expected Output

```
x=11
y=10
```

Using super to access members of a superclass

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Syntax

super.*member*

Here, *member* can be either a method or an instance variable.

Example Program1

```
//Using Super keyword to access variables of a super class

class A
{
int a=10;
        void displayA()
        {
        System.out.println("a in class A="+a);
        }

}
class B extends A
{
int a=20;
        void displayB()
        {
         System.out.println("a in class B="+super.a);
        }

}

class Super2
{
public static void main(String z[])
{
B b=new B();
b.displayA();
b.displayB();
}
}
```

Expected Output

```
a in class A=10
a in class B=10
```

Example Program2

```
//Using Super to call overridden method of a super class
class A
{
        void show()
        {
        System.out.println("show() in class A");
        }
}
class B extends A
{
        void show()
        {
        super.show();   //invokes super class method show()
        System.out.println("show() in class B");
        }

}

class Super3
{
public static void main(String z[])
{
B b=new B();
b.show();
}
}
```

Expected Output

```
show() in class A
show() in class B
```

final keyword

- The keyword **final** has three uses.
- First, it can be used to create the equivalent of a named constant.
- The other two uses of **final** apply to inheritance.

Using final with variables

- 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. (In this usage, **final** is similar to **const** in C/C++.)

Example
final int a=10;

Using final with Inheritance

(a) Using final to prevent Overriding

- While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.
- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- Methods declared as **final** cannot be overridden.

Example Program

```
class A
{
final void show()
{
System.out.println("show() in class A is final");
}
}
class B extends A
{
void show()            // creates ERROR: can't override final method
{
System.out.println("show() in class B");
}
}
class Final1
{
public static void main(String z[])
{
B ob=new B();
ob.show();
}
}
```

(b)Using final to prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Example

```
final class A
{
//…
}
class B extends A    // ERROR: can’t inherit a final class
{
//…
}
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

The version of the method defined by the superclass will be hidden.

Example Program

```
class A
{
void show()
{
System.out.println(“show() in class A”);
}
}
class B extends A
{
void show()             // subclass method overrides the superclass method “show()”
{
System.out.println(“show() in class B”);
}
}
class Overriding
{
public static void main(String z[])
{
B ob=new B();
ob.show();
}
}
```

Expected Output

```
show() in class B
```

Polymorphism through Method Overriding

Dynamic Method Dispatch (Runtime Polymorphism)

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.*
- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object.
- Java uses this fact to resolve calls to overridden methods at run time.
- Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Example Program

```
//Runtime Polymorphism or Dynamic method dispatch
class A
{
        void show()
        {
        System.out.println("show() in class A");
        }
}

class B extends A
{
        void show()
        {
        System.out.println("show() in class B");
        }
}

class C extends A
{
        void show()
        {
        System.out.println("show() in class C");
        }
}
```

```
class Runtime
{
public static void main(String z[])
{
A a=new A();
B b=new B();
C c=new C();
A ref;
ref=a;
ref.show();
ref=b;
ref.show();
ref=c;
ref.show();
}
}
```

Expected Output

```
show() in class A
show() in class B
show() in class C
```

Abstract class

- Sometimes we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.
- In java, the class that contains abstract methods is known as "abstract class"
- Abstract class can also contains concrete methods like a normal class.
- Variables can also be declared in the abstract class.
- Abstract methods are incomplete methods and hence abstract class also becomes incomplete class.
- Objects cannot be created to an abstract class.
- Thus the class that extends the abstract class should provide definition to all the methods declared in the abstract class.

To declare an abstract method, use this general form

abstract *type name(parameter-list)*;

Example

abstract void show();

The above method show() is an abstract method as it doesn't have body and its declaration should end with semicolon(;).

Example Program

```
//program to show the use of abstract class
abstract class Marks
{
int tot=100;
        abstract void internal();                    // abstract methods
        abstract void external();

        void total()         // concrete method
        {
        System.out.println("Total Marks:"+tot);
        }
}

class Detail extends Marks
{
        void internal()
        {
        System.out.println("Int:30 marks");
        }

        void external()
        {
        System.out.println("Ext:70 marks");
        }
}

class Abstract1
{
public static void main(String z[])
{
Detail d=new Detail();
d.internal();
d.external();
d.total();
}
}
```

Expected Output

```
Int:30 marks
Ext:70 marks
Total Marks:100
```

UNIT-IV

Packages

"A Package is a collection of classes which provides high level of access protection and name space management."

- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- Packages are stored in a hierarchical manner and are manually imported into new class definitions.
- For example, a package allows you to create a class named **A**, which you can store in your own package without concern that it will collide with some other class named **A** stored elsewhere.
- Thus package is both a naming and a visibility control mechanism.
- We can define classes inside a package that are not accessible by code outside that package.

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If we omit the **package** statement, the class names are put into the default package, which has no name.

The general form of the package statement is as follows

package nameofpackage;

Example

package pack1;

- The **package** statement simply specifies to which package the classes defined in a file belong.
- Java uses file system directories to store packages.
- For example, the **.class** files for any classes you declare to be part of **pack1** must be stored in a directory called **pack1**.
- Package name is case sensitive, and the directory name must match the package name exactly.

Java also allows us to create a hierarchy of packages.
To do so, simply separate each package name from the one above it by use of a period(.)
The general form of a multileveled package statement is shown below.

package pack1.pack2.pack3;

where pack3 is a sub package of pack2 which is a sub package of pack1.

Example

```
package pack1;

public class A
{
        public void showA()
        {
        System.out.println("showA() method.");
        }
}
```

The class that is to be stored in the package must be declared as public because if we want to access it from outside the class.

Importing Packages

Java provides import statement to access individual or all the classes belonging to a package. In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

The general form of the **import** statement is as follows

import packagename.*classname*;

Example

import pack1.pack2.**A**;

- Here, **pack1** is the name of a top-level package, and ***pack2*** is the name of a subordinate package inside the outer package separated by a dot (**.**).
- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.

Example Program

// program to show how to define and access the package

A.java

```
package pack1;      // defining a package

public class A
{
        public void show()
        {
        System.out.println("show() method.");
        }
}
```

Package1.java

```
import pack1.A;      // accessing a package

class Package1
{
public static void main(String z[])
{
A ob=new A();
ob.show();
}
}
```

Understanding CLASSPATH

Before an example that uses a package is presented, a brief discussion of the **CLASSPATH** environmental variable is required. While packages solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by **CLASSPATH**.

Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory (**.**) is usually in the **CLASSPATH** environmental variable defined for the Java run-time system, by default. However, things are not so easy when packages are involved. Here's why.

Assume that you create a class called **PackTest** in a package called **test**. Since your directory structure must match your packages, you create a directory called **test** and put **PackTest.java** inside that directory. You then make **test** the current directory and compile **PackTest.java**. This results in **PackTest.class** being stored in the **test** directory, as it should be. When you try to run **PackTest**, though, the Java interpreter reports an error message similar to "can't find class PackTest." This is because the class is now stored in a package called **test**. You can no longer refer to it simply as **PackTest**. You must refer to the class by enumerating its package hierarchy, separating the packages with dots. This class must now be called **test.PackTest**. However, if you try to use **test.PackTest**, you will still receive an error message similar to "can't find class test/PackTest."

The reason you still receive an error message is hidden in your **CLASSPATH** variable. Remember, **CLASSPATH** sets the top of the class hierarchy. The problem is that there's no **test** directory in the current working directory, because *you are in* the **test** directory, itself.

You have two choices at this point: change directories up one level and try **java test.PackTest**, or add the top of your development class hierarchy to the **CLASSPATH** environmental variable. Then you will be able to use **java test.PackTest** from any directory, and Java will find the right **.class** file. For example, if you are working on your source code in a directory called **C:\\myjava**, then set your **CLASSPATH** to .;C:\myjava;C:\java\classes.

Interfaces

- Using the keyword **interface**, you can fully abstract a class from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, But they contain final variables, and their methods are declared without any body.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Multiple Inheritance is achieved in java through interfaces.

Defining an Interface

An interface is defined by using a keyword “interface”.

Syntax

```
access interfacename
 {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Explanation

- Here, *access* is either **public** or **no modifier**.
- Here *name* is the name of the interface.
- Notice that the methods which are declared have no bodies. They end with a semicolon
  after the parameter list. They are, essentially, abstract methods.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside the interface declarations.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Example

```
interface Marks
{
void internal();
void external();
}
```

The above interface named “Marks” contain two abstract methods which are automatically public.

Implementing Interface

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** keyword in a class definition, and then implement the methods defined by the interface.

Syntax

```
access class classname  [extends superclass]  implements  [interface1, interface2,….]
{
//class body
}
```

Explanation

- Here, *access* is a modifier.
- If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature.
- specified in the **interface** definition.

Example Program

```
// program to show the use of interface
interface Marks
{
int tot=100;

void internal();
void external();
}
class Detail implements Marks
{
        public void internal()
        {
        System.out.println("Int Marks=30");
        }
```

```
  public void external()
        {
        System.out.println("Ext Marks=70");
        }

        void total()
        {
        System.out.println("Total Marks="+tot);
        }
}

class Interface1
{
public static void main(String z[])
{
Detail d=new Detail();
d.internal();
d.external();
d.total();
}
}
```

Expected Output

```
Int Marks=30
Ext Marks=70
Total Marks=100
```

Applying Interfaces

- To understand the power of interfaces, let's look at a more practical example. Earlier we have developed a class called **Stack** that implemented a simple fixed-size stack.
- However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable."
- The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same.
- That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics.

Example Program

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}

// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos—];
  }
}

class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
```

```
      for(int i=0; i<8; i++) mystack2.push(i);

      // pop those numbers off the stack
      System.out.println("Stack in mystack1:");
      for(int i=0; i<5; i++)
         System.out.println(mystack1.pop());

      System.out.println("Stack in mystack2:");
      for(int i=0; i<8; i++)
         System.out.println(mystack2.pop());
   }
}
```

Variables in Interfaces

- The variables in the interfaces are automatically declared as static and final.
- The final modifier ensures the value assigned to the interface variable is a true constant that cannot be re-assigned by program code.
- It means that their values cannot be changed by the classes that implement the interface.
- We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.
- When you include that interface in a class (that is, when you "implement" the interface),
  all of those variable names will be in scope as constants.

Example Program

```
// variables in interface

interface Constants
{
int internal=30;     // all three variables are final by default
int external=70;
int total=100;
}

class Marks implements Constants
{
void display()
{
  //internal=40; // creates an error becoz "internal" is a final variable
System.out.println("Internal Marks="+internal);
System.out.println("External Marks="+external);
System.out.println("Total Marks="+total);
}
}
```

```
class Interface2
{
public static void main(String z[])
{
Marks m=new Marks();
m.display();
}
}
```

Expected Output

```
Internal Marks=30
External Marks=70
Total Marks=100
```

Extending Interfaces

- Like classes, interfaces can also be extended by using "extends" keyword.
- When one interface extends another interface, the properties one interface is inherited to another.
- The class that implements the sub interface should provide definitions to all the methods of both the interfaces.

Syntax

```
interface Interface1
{
}
interface Interface2 extends Interface1
{
}
```

Example Program

```
//extending interfaces
interface A
{
void show1();
void show2();
}

interface B extends A
{
void show3();            // Now interface 'B' contains 3 abstract methods.
}

class Test implements B
{
    public void show1()
```

```
        {
        System.out.println("show1() method");
        }
        public void show2()
        {
        System.out.println("show2() method");
        }
        public void show3()
        {
        System.out.println("show3() method");
        }
}

class Interface3
{
public static void main(String z[])
{
Test t=new Test();
t.show1();
t.show2();
t.show3();
}
}
```

Expected Output

```
show1() method
show2() method
show3() method
```

Exploring Packages

java.io

- Let us now explore java.io package.
- As all programmers learn early on, most programs cannot accomplish their goals without accessing external data.
- Data is retrieved from an *input* source.
- The results of a program are sent to an *output* destination.
- In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these devices are all handled by the same abstraction: the *stream.* A stream is a logical entity that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices they are linked to differ.

The following are some of the classes and interfaces present in java.io. package

BufferedReader

BufferedWriter

BufferedInputStream

BufferedOutputStream etc.

BufferedReader

BufferedReader improves performance by buffering input.
It has two constructors:

BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*.

Example Program

```
import java.io.*;
class Sample
{
public static void main(String z[]) throws IOException
{
BufferedReader x=new BufferedReader(new InputStreamReader(System.in));
int a,b,c;
System.out.println("Enter any two numbers");
a=Integer.parseInt(x.readLine());
b=Integer.parseInt(x.readLine());
c=a+b;
System.out.println("Sum="+c);
}
}
```

Expected Output

```
Enter any two numbers
20
10
Sum=30
```

java.util

The java.util package contains classes that deal with collections, events, date and time, and various helpful utilities.

★★★★★★★

UNIT-V

Exception Handling

An *exception* is an abnormal condition that arises in a code sequence at run time.
In other words , an exception is a runtime error that indicates that some problem has occurred during program execution.
In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.
Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.
- Your code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java runtime system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed before a method returns is put in a **finally** block.

General form

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
// ...
finally
{
// block of code to be executed before try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

All exception types are subclasses of the built-in class **Throwable**.
Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.

There is an important subclass of **Exception**, called **RuntimeException**.
Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
Stack overflow is an example of such an error.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits.

(1) First, it allows you to fix the error.
(2) Second, it prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, includes a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

Example Program

```
//use of try and catch

class Prog1
{
public static void main(String z[])
{
int a,b,c;
a=10;
b=0;

try
{
c=a/b;
System.out.println("Result="+c);
}

catch(ArithmeticException e)
{
System.out.println("Denominator should not be zero.");
}

System.out.println("After catch block.");
}
}
```

Expected Output

```
Denominator should not be zero.
After catch block.
```

Explanation

- Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.
- Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line " After catch block." is not displayed.
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism.

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch** clauses, each catching a
- different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.

- After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.

Example Program
//try with multiple catch blocks

```
class Prog2
{
public static void main(String z[])
{
int a,b,c;
try
{
a=Integer.parseInt(z[0]);
b=Integer.parseInt(z[1]);
c=a/b;
System.out.println("Result="+c);
}

catch(ArithmeticException e)
{
System.out.println("Denominator should not be zero.");
}

catch(NumberFormatException e)
{
System.out.println("Enter only numerical values");
}

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Enter only two numbers");
}

System.out.println("After catch block.");
}
}
```

Expected Output

```
C:\>java Prog2 20 two
Enter only numerical values
After catch block.
```

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.

Example Program
//nested try blocks

```
import java.io.*;

class Prog3
{
public static void main(String z[]) throws IOException
{
BufferedReader x=new BufferedReader(new InputStreamReader(System.in));
int a,b,c;
try
{
System.out.println("Enter any two numbers:");
a=Integer.parseInt(x.readLine());
b=Integer.parseInt(x.readLine());
        try
        {
        c=a/b;
        System.out.println("Result="+c);
        }
        catch(ArithmeticException e)
        {
        System.out.println("Denominator should not be zero.");
        }
}
catch(NumberFormatException e)
{
System.out.println("Enter only numerical values");
}
System.out.println("After catch block.");
}
}
```

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Example Program

```
//use of throw
import java.io.*;

class Prog4
{
public static void main(String z[]) throws IOException
{
BufferedReader x=new BufferedReader(new InputStreamReader(System.in));
int a,b,c;
try
{
System.out.println("Enter any two numbers:");
a=Integer.parseInt(x.readLine());
b=Integer.parseInt(x.readLine());
c=a/b;
System.out.println("Result="+c);
throw new ArithmeticException();    // manually throwing an exception
}

  catch(ArithmeticException e)
        {
        System.out.println("Denominator should not be zero.");
        }

System.out.println("After catch block.");
}
}
```

Expected Output

```
Enter any two numbers:
20
10
Result=2
Denominator should not be zero.
After catch block.
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

type method-name(parameter-list) **throws** *exception-list*
{
// body of method
}
Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Example Program

```
//use of throws
class Prog5
{
static int res;
        static void display(int x,int y) throws NoSuchMethodException
        {
        res=x/y;
        System.out.println("Result="+res);
        }

public static void main(String z[])
{

  try
        {
        display(20,0);
        }
        catch( NoSuchMethodException e)
        {
        System.out.println("Not Zero!");
        }
}
}
```

finally

- **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional.
- However, each **try** statement requires at least one **catch** or a **finally** clause.

Example Program

```
// use of finally

class Prog6
{
static int res;
        static void div(int x,int y)
        {
        try
        {
        res=x/y;
        System.out.println("Result="+res);
        }

        finally        //finally block is executed first than catch block
        {
        System.out.println("Finally block!");
        }
        }

public static void main(String z[])
{
try
{
div(20,0);
}
catch(ArithmeticException e)
{
System.out.println("Not Zero!");
}
}
}
```

Expected Output

```
Finally block!
Not Zero!
```

Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples.
Hence Exceptions are generally classified into the following two categories.
(i)Unchecked exceptions
(ii)Checked exceptions.

(i) Unchecked exceptions

- The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.
- Furthermore, they need not be included in any method's **throws** list.
- In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in **java.lang** are listed in Table 10-1.

(ii) Checked exceptions

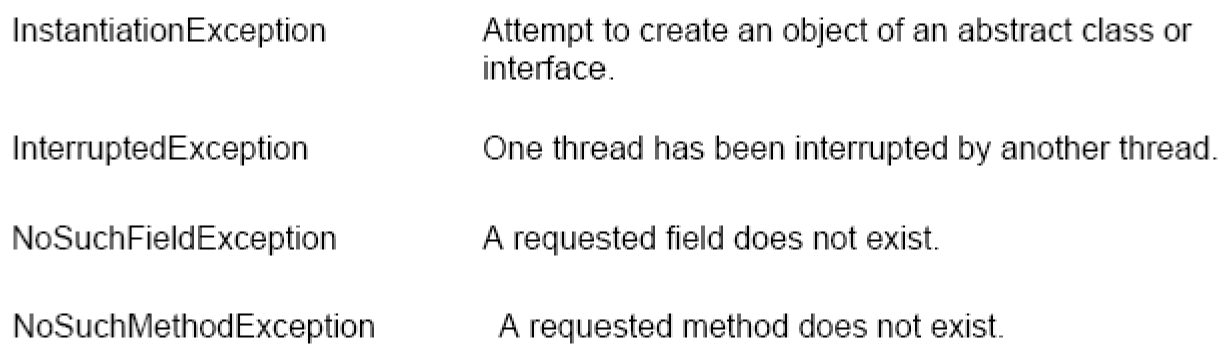
- Table 10-2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.
- These are called *checked exceptions.*

Table 10-1. Java's Unchecked RuntimeException Subclasses

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-2. Java'S Checked Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.



InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating Your Own Exception Subclasses

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Example Program

```
class Retire extends Exception
{
int age;
        Retire(int age)
        {
        this.age=age;
        }

        public String toString()
        {
        return "Age should be <=58 years "+age+" is wrong age";
        }
}

class UserExcep
{
public static void main(String z[])
{

int age=Integer.parseInt(z[0]);
int sal;
try
{
if(age>58) throw new Retire(age);
if(age<=30)
sal=5000;

else if(age<=40)
sal=10000;

else
sal=15000;
System.out.println("Salary:"+sal);
}
catch(Retire e)
{
System.out.println(e);
}
}
}
```

Expected Output

```
C:\>java Sample 59

Age should be <=58 years 59 is wrong age
```

Multithreaded Programming

Unlike most other computer languages, Java provides built-in support for *multithreaded programming.*
A multithreaded program contains two or more parts that can run concurrently.
Each part of such a program is called a *thread,* and each thread defines a separate path of execution.
Thus, multithreading is a specialized form of multitasking.
There are two distinct types of multitasking process-based and thread-based.
It is important to understand the difference between the two.

Differences between process based and thread based multitasking

Process based multitasking

- For most readers, process-based multitasking is the more familiar form.
- A *process* is, in essence, a program that is executing.
- Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- Processes are heavyweight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

Thread based multitasking

- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.
- Multitasking threads require less overhead than multitasking processes.
- Threads, on the other hand, are lightweight.
- They share the same address space and cooperatively share the same heavyweight process.
- Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java.
However, multithreaded multitasking is under the control of java.

Thread life cycle:

During its life cycle, every thread undergoes the following five states.

(1) Born state state

(2) Runnable state

(3) Running state

(4) Blocked state

(5) Dead state

Let us now discuss each of them briefly.

(1) Born State:
A thread is said to be in born state when a new thread object is created.
During this state, a thread doesn't perform any work.

(2) Runnable State:
A thread is said to be in runnable state if it is ready for execution and is waiting for the availability of the processor in a queue.
All the threads waiting in the queue have their priorities.
The thread with the highest priority will go to the running state first.

(3) Running State:
A thread is said to be in running state if it is under the execution of the processor.

(4) Blocked State:
A thread is said to be in blocked state, if it is avoided from entering from runnable to running state using suspend(), sleep() and wait() methods.

(5) Dead State:
It is the final state in the life cycle of the thread.
A thread may die in the following two ways.
(i) *Natural death:* After completion of execution, the thread generally dies.
(ii) *Premature death:* A thread can be killed before the completion of its execution.

The Main Thread

When a Java program starts up, one thread begins running immediately.
This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:
• It is the thread from which other "child" threads will be created.
• It must be the last thread to finish execution.
When the main thread stops, your program terminates.

Example Program

```
class Example1
{
public static void main(String z[])
{
System.out.println("Main thread created.");
Thread t=Thread.currentThread();
System.out.println(t);
t.setName("New Thread");
System.out.println("After name changed:"+t);
}
}
```

Expected Output

```
Main thread created.
Thread[main,5,main]
After name changed:Thread[New Thread,5,main]
```

Creating a Thread

Threads can be created in one of the following two ways.

(1) By extending Thread class
(2) By implementing Runnable interface

Let us now discuss these two methods briefly.

(1) Extending Thread

- The first way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

Example Program

```
//By extending "Thread" class
class User extends Thread
{
String name;
        User(String s)
        {
        super(s);
        System.out.println("Child thread created!");
        name=s;
        start();
        }

        public void run()
        {
        try
        {
        for(int i=101;i<=105;i++)
        {
        System.out.println(name+":"+i);
        sleep(500);
        }
        }

        catch(InterruptedException e)
        {
        System.out.println("Child thread interrupted!");
        }
        }
}

class Thread1
{
public static void main(String z[])
{
System.out.println("Main thread created!");
User u=new User("Child1");

try
{
for(int i=1;i<=5;i++)
{
System.out.println("Main:"+i);
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
System.out.println("Main thread Interrupted!");
}
}
}
```

Expected Output

```
Main thread created!
Child thread created!
Main:1
Child1:101
Child1:102
Main:2
Child1:103
Child1:104
Main:3
Child1:105
Main:4
Main:5
```

(2) Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code.
- You can construct a thread on any object that implements **Runnable**.
- To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

- Inside **run( )**, you will define the code that constitutes the new thread.
- It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program.

Example Program

```
//By implementing "Runnable" interface
class User implements Runnable
{
Thread t;
String name;
        User(String s)
        {
        t=new Thread(this,s);
        System.out.println("Child thread created!");
        name=s;
        t.start();
        }

        public void run()
        {
        try
        {
        for(int i=101;i<=105;i++)
        {
        System.out.println(name+":"+i);
        t.sleep(500);
        }
        }

        catch(InterruptedException e)
        {
        System.out.println("Child thread interrupted!");
        }
        }
}

class Thread2
{
public static void main(String z[])
{
System.out.println("Main thread created!");
User u=new User("Child1");

try
{
for(int i=1;i<=5;i++)
{
System.out.println("Main:"+i);
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
System.out.println("Main thread Interrupted!");
}
}
}
```

Expected Output

```
Main thread created!
Child thread created!
Main:1
Child1:101
Child1:102
Main:2
Child1:103
Child1:104
Main:3
```

```
Child1:105
Main:4
Main:5
```

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread.
However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

Example Program

```
//Creating multiple threads
class User extends Thread
{
String name;
        User(String s)
        {
        super(s);
        System.out.println("Child thread created!");
        name=s;
        start();
        }
        public void run()
        {
        try
        {
        for(int i=101;i<=105;i++)
        {
        System.out.println(name+":"+i);
        sleep(1000);
        }
        }
        catch(InterruptedException e)
        {
        System.out.println("Child thread interrupted!");
        }
        }
}

class Thread3
{
public static void main(String z[])
{
User u1=new User("Child1");
User u2=new User("Child2");
User u3=new User("Child3");
}
}
```

Example Program

```
Child thread created!
Child thread created!
Child1:101
Child thread created!
Child2:101
Child3:101
Child1:102
```

```
Child3:102
Child2:102
Child3:103
Child1:103
Child2:103
Child3:104
Child1:104
Child2:104
Child3:105
Child2:105
Child1:105
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads and threads of equal priority should get equal access to the CPU.
- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

Its general form is given below

final void **setPriority**(int *level*)

- Here, *level* specifies the new priority setting for the calling thread.
- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.
- These priorities are defined as **final** variables within **Thread**.

Thread Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called *synchronization.*
- Key to synchronization is the concept of the monitor (also called a *semaphore*).
- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.*
- Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor.
- These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Example Program

```
//Program on thread synchronization
class Message
{
	synchronized void show(String s)
```

```
        {
        System.out.print("["+s);
                try
                {
                Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                }
                System.out.println("]");
         }
}
class User implements Runnable
{
Thread t;
String s;
Message m;
        User(Message m,String s)
        {
        t=new Thread(this);
        this.m=m;
        this.s=s;
        t.start();
        }
        public void run()
        {
        m.show(s);
        }
}

class Thread4
{
public static void main(String z[])
{
Message m=new Message();
User u1=new User(m,"Hello");
User u2=new User(m,"Synchronized");
User u3=new User(m,"World");
}
}
```

Expected Output

```
[Hello]
[Synchronized]
[World]
```

****