10

Interfaces: Multiple Inheritance

KEY TERMS

Interface | Implementation | Multiple inheritance

10.1 INTRODUCTION

In Chapter 8, we discussed about classes and how they can be inherited by other classes. We also learned about various forms of inheritance and pointed out that Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

```
class A extends B extends C
{
      ..............
      ..............
}
```

is not permitted in Java. However, the designers of Java could not overlook the importance of multiple inheritance. A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. Since C++ like implementation of multiple inheritance proves difficult and adds complexity to the language, Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

10.2 DEFINING INTERFACES

An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

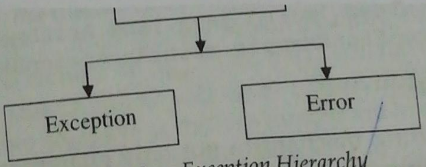


Fig. 3.4 Java Exception Hierarchy

The syntax for defining an interface is very similar to that for defining a class. The general form an interface definition is

```
interface InterfaceName
{
     variables declaration;
     methods declaration;
}
```

Here, **interface** is the key word and *InterfaceName* is any valid Java variable (just like cla names). Variables are declared as follows:

```
static final type VariableName = Value;
```

Note that all variables are declared as constants. Methods declaration will contain only a list methods without any body statements. Example:

```
return-type methodName1 (parameter_list);
```

Here is an example of an interface definition that contains two variables and one method:

```
interface Item
{
     static final int code = 1001;
     static final String name = "Fan";
     void display ( ) ;
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method. Another example of an interface is

```
interface Area
{
     final static float pi = 3.142F;
     float compute (float x, float y);
     void show ( );
}
```

Table 10.1 lists the differences between class and interface.

Table 10.1 *Difference between class and interface*

Class	Interface
The members of a class can be constant or variables.	The members of an interface are always declared as constant i.e., their values are final.
The class definition can contain the code for each of its methods. That is, the methods can be abstract or non-abstract.	The methods in an interface are abstract in nature, i.e., there is no code associated with them. It is later defined by the class that implements the interface.
It can be instantiated by declaring objects.	It cannot be used to declare objects. It can only be inherited by a class.
It can use various access specifiers like public, private, or protected.	It can only use the public access specifier.

10.3 EXTENDING INTERFACES

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword **extends** as shown below.

```
interface name2 extends name1
{
          body of name2
}
```

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required. Example:

```
interface ItemConstants
{
          int code = 1001;
          string name = "Fan";
}
interface Item extends ItemConstants
{
          void display ( );
}
```

The interface **Item** would inherit both the constants **code** and **name** into it. Note that the variables **name** and **code** are declared like simple variables. It is allowed because all the variables in an interface are treated as constants although the keywords **final** and **static** are not present.

We can also combine several interfaces together into a single interface. Following declarations are valid:

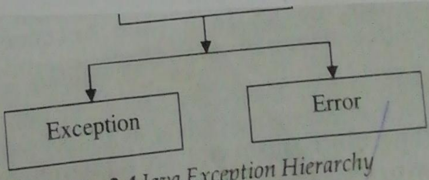
```
interface ItemConstants
{
          int code = 1001;
          String name = "Fan";
}
interface ItemMethods
{
          void display( );
}
interface Item extends ItemConstants, ItemMethods
{
          .............
          .............
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas.

It is important two remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.

10.4 IMPLEMENTING INTERFACES

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:



3.4 Java Exception Hierarchy

It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Multithreading is a powerful programming tool that makes Java distinctly different from its fellow programming languages. Multithreading is useful in a number of ways. It enables programmers to do multiple things at one time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. For example, we can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

Threads are extensively used in Java-enabled browsers such as HotJava. These browsers can download a file to the local computer, display a Web page in the window, output another Web page to a printer, and so on.

Any application we are working on that requires two or more things to be done at the same time is probably a best one for use of threads.

Table 12.1 *Difference between multithreading and multitasking*

Multithreading	***Multitasking***
It is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time in parallel.	It is an operating system concept in which multiple tasks are performed simultaneously.
It supports execution of multiple parts of a single program simultaneously.	It supports execution of multiple programs simultaneously.
The processor has to switch between different parts or threads of a program.	The processor has to switch between different programs or processes.
It is highly efficient.	It is less efficient in comparison to multithreading.
A thread is the smallest unit in multithreading.	A program or process is the smallest unit in a multitasking environment.
It helps in developing efficient programs.	It helps in developing efficient operating systems.
It is cost-effective in case of context switching.	It is expensive in case of context switching.

12.2 CREATING THREADS

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called **run( )**. The **run( )** method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behavior can be implemented. A typical **run( )** would appear as follows:

```
public void run( )
{
        ..........
        .......... (statements for implementing thread)
        ..........
}
```

The **run**( ) method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start** ( ).

A new thread can be created in two ways.

1. **By creating a thread class:** Define a class that extends **Thread** class and override its run( ) method with the code required by the thread.

2. **By converting a class to a thread:** Define a class that implements **Runnable** interface. The **Runnable** interface has only one method, **run( ),** that is to be defined in the method with the code to be executed by the thread.

The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the **Runnable** interface, since Java classes cannot have two superclasses.

12.3 EXTENDING THE THREAD CLASS

We can make our class runnable as thread by extending the class **java.lang.Thread.** This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the **Thread** class.
2. Implement the **run( )** method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the **start( )** method to initiate the thread execution.

Declaring the Class

The **Thread** class can be extended as follows:

```
class MyThread extends Thread
{
     ...............
     ...............
     ...............
}
```

Now we have a new type of thread **MyThread**.

Implementing the *run()* Method

The **run( )** method has been inherited by the class **MyThread**. We have to override this method in order to implement the code to be executed by our thread. The basic implementation of **run()** will look like this:

```
public void run()
{
     ................
     ................          // Thread code here
     ................
}
```

When we start the new thread, Java calls the thread's **run( )** method, so it is the **run( )** where all the action takes place.

Starting New Thread

To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread( );
aThread.start( );                        // invokes run() method
```

The first line instantiates a new object of class **MyThread**. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a *newborn* state.

The second line calls the **start( )** method causing the thread to move into the *runnable* state. Then, the Java runtime will schedule the thread to run by invoking its **run( )** method. Now, the thread is said to be in the *running* state.

12.5 LIFE CYCLE OF A THREAD

During the lifetime of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig. 12.4.

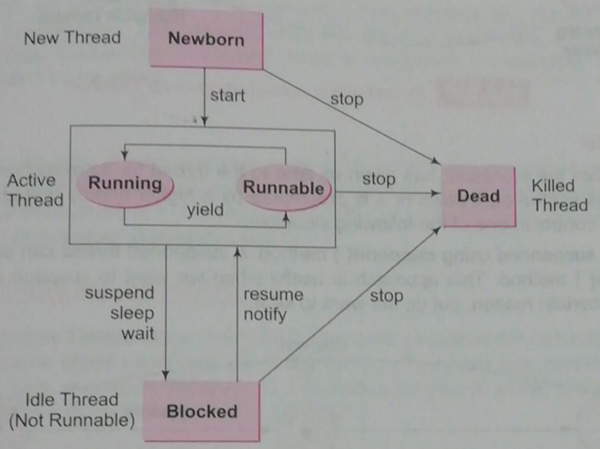


Fig. 12.4 State transition diagram of a thread

Newborn State

When we create a thread object, the thread is born and is said to be in *newborn* state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- Schedule it for running using **start( )** method.
- Kill it using **stop( )** method.

If scheduled, it moves to the runnable state (Fig. 12.5). If we attempt to use any other method at this stage, an exception will be thrown.

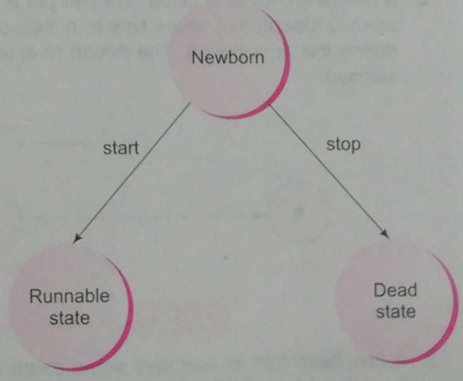


Fig. 12.5 Scheduling a newborn thread

Runnable State

The *runnable* state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority,

Fig. 3.4 Java Exception Hierarchy

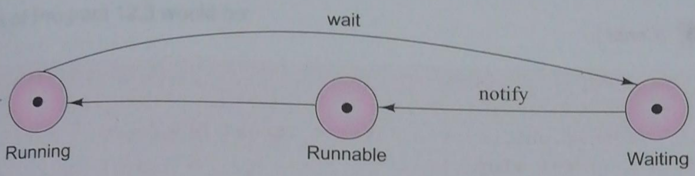


Fig. 12.9 *Relinquishing control using wait( ) method*

Blocked State

A thread is said to be *blocked* when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain ~~requirements.~~ A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead State

Every thread has a life cycle. A running thread ends its life when it has completed exe~~cuting its~~ **run( )** method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not ~~runnable" (blocked)~~ condition.

12.6 USING THREAD METHODS

We have discussed how **Thread** class methods can be used to control the behavior of a thread. We have used the methods **start( )** and **run( )** in Program 12.1. There are also methods that can move a thread from one state to another. Program 12.3 illustrates the use of **yield( )**, **sleep( )** and **stop( )** methods. Compare the outputs of Programs 12.1 and 12.3.

Program 12.3 *Use of yield( ), stop( ), and sleep( ) methods*

```
class A extends Thread
{
   public void run( )
   {
        for(int i  =  1; i<=5; i++)
        {
          if(i==1) yield( );
          System.out.println("\tFrom Thread A : i = " +i);
        }
       System.out.println("exit from A " );
   }
}
class B extends Thread
{
   public void run( )
   {
```

Program 12.5 *(Contd.)*

```
                System.out.println("\nThis is the main Thread\t Thread
                ID: "+Thread.currentThread().getId()+"\tThread Priority:
                "+Thread.currentThread().getPriority()+"\n");
                System.out.println("Let's call the other threads in the
                sequence A->B->C\n");
                threadA.start();
                threadB.start();
                threadC.start();
        }
    }
```

The output of Program 12.5 would be:

```
This is the main Thread              Thread ID: 1         Thread Priority: 5
Let's call the other threads in the sequence A->B->C
This is Thread C                     Thread ID: 11        Thread Priority: 10
This is Thread A                     Thread ID: 9         Thread Priority: 1
This is Thread B                     Thread ID: 10        Thread Priority: 5
```

12.9 SYNCHRONIZATION

So far, we have seen threads that use their own data and methods provided inside their run() methods. What happens when they try to use data and methods outside themselves? On such occasions, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronization*.

In case of Java, the keyword **synchronized** helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as **synchronized.** Example:

```
synchronized void update( )
{
    ................
    ................          // code here is synchronized
    ................
}
```

When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as synchronized as shown below:

```
synchronized  ( lock-object )
{
    ................          // code here is synchronized
    ................
}
```

5

Operators and Expressions

KEY TERMS

Operands | Integer arithmetic | Real arithmetic | Mixed-mode arithmetic | Relational expression | Logical expression | Truth table | Ternary operator | Conditional operator | Increment operator | Decrement operator | Dot operator | instanceof operator | Casting | Operator precedence | Associativity

5.1 INTRODUCTION

Java supports a rich set of operators. We have already used several of them, such as =, +, – and *. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions.

Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

In this chapter, we discuss each one of these categories with illustrations.

5.2 ARITHMETIC OPERATORS

Arithmetic operators are used to construct mathematical expressions as in algebra. Java provides all the basic arithmetic operators. They are listed in Table 5.1. The operators +, –, *, and / all work the same way as they do in other languages.

Table 5.1 *Arithmetic operators*

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Remainder)

These can operate on any built-in numeric data type of Java. We cannot use these operators on boolean type. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Arithmetic operators are used as shown below:

a – b   a + b
a * b   a/b
a % b   – a * b

Here **a** and **b** may be variables or constants and are known as operands.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a + b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. In the above examples, if a and b are integers, then for a = 14 and b = 4 we have the following results:

a – b = 10
a + b = 18
a * b = 56
a/b = 3 (decimal part truncated)
a % b = 2 (remainder of integer division)

a/b, when **a** and **b** are integer types, gives the result of division of **a** by **b** after truncating the divisor. This operation is called the *integer division*.

For modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

–14 % 3 = –2
–14 % –3 = –2
14 % –3 = 2

*(Note that modulo division is defined as: a % b = a – (a/b) * b, where a/b is the integer division.)*

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Unlike C and C++, modulus operator % can be applied to the floating point data as well. The floating point modulus operator returns the floating point equivalent of an integer division. What this means is that the division is carried out with both floating point operands, but the resulting divisor is treated as an integer, resulting in a floating point remainder. Program 5.1 shows how arithmetic operators work on floating point values.

Program 5.1 *Floating point arithmetic*

```
class FloatPoint
{
    public static void main(String args[])
    {
        float a = 20.5F, b = 6.4F;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
```

(Contd.)

Program 5.1 *(Contd.)*

```
        System.out.println(" a+b = " + (a+b));
        System.out.println(" a-b = " + (a-b));
        System.out.println(" a*b = " + (a*b));
        System.out.println(" a/b = " + (a/b));
        System.out.println(" a%b = " + (a%b));
    }
}
```

The output of Program 5.1 would be:

```
        a = 20.5
        b = 6.4
        a+b = 26.9
        a-b = 14.1
        a*b = 131.2
        a/b = 3.20313
        a%b = 1.3
```

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mo arithmetic* expression. If either operand is of the real type, then the other operand is converted to and the real arithmetic is performed. The result will be a real. Thus,

```
        15/10.0  produces the result   1.5
```

whereas

```
        15/10    produces the result   1
```

More about mixed operations will be discussed later when we deal with the evaluation expressions.

5.3 RELATIONAL OPERATORS

We often compare two quantities, and depending on their relation, take certain decisions. Fo example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol < meaning 'less than'. An expression such as

```
    a < b or x < 20
```

containing a relational operator is termed as a *relational expression*. The value of relational expression is either true or false. For example, if x = 10, then

```
    x < 20 is true
```

while

```
    20 < x is false.
```

Java supports six relational operators in all. These operators and their meanings are shown in Table 5.2.

A simple relational expression contains only one relational operator and is of the following form:

Table 5.2 *Relational operators*

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

5.4 LOGICAL OPERATORS

In addition to the relational operators, Java has three logical operators, which are given in Table 5.4.

The logical operators && and || are used when we want to form compound conditions by combining two or more relations. An example is:

```
a > b && x == 10
```

An expression of this kind which combines two or more relational expressions is termed as a *logical expression* or a *compound relational expression.* Like the simple relational expressions, a logical expression also yields a value of true or false, according to the *truth table* shown in Table 5.5. The logical expression given above is true only if both **a** > **b** and **x** == **10** are true. If either (or both) of them are false the expression is false.

Table 5.4 Logical opera

Operator	Meaning
&&	logical AND
||	logical OR
!	logical NOT

Table 5.5 *Truth table*

op – 1	*op – 2*	*Value of the expression*	
		op – 1 && op – 2	*op – 1 || op – 2*
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Note:

- *op–1 && op–2 is true if both op–1 and op–2 are true and false otherwise.*
- *op–1 || op–2 is false if both op–1 and op–2 are false and true otherwise.*

Some examples of the usage of logical expressions are:

```
1. if (age>55 && salary<1000)
2. if (number<0 || number>100)
```

5.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, ' = '. In addition, Java has a set of 'shorthand' assignment operators which are used in the form

```
v op= exp;
```

where v is a variable, *exp* is an expression and *op* is a Java binary operator. The operator **op =** is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op(exp);
```

with v accessed only once. Consider an example:

```
X += y+1;
```

This is same as the statement

```
x = x+(y+1);
```

The shorthand operator += means 'add y+1 to x' or 'increment x by y+1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 5.6.

Table 5.6 *Shorthand assignment operators*

Statement with simple assignment operator	***Statement with shorthand operator***
a = a + 1	a += 1
a = a–1	a –= 1
a = a*(n+l)	a *= n+1
a = a/(n+l)	a /= n+1
a = a%b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. Use of shorthand operator results in a more efficient code.

5.6 INCREMENT AND DECREMENT OPERATORS

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators:

```
++ and --
```

The operator ++ adds 1 to the operand while –– subtracts 1. Both are unary operators and are used in the following form:

```
++m; or          m++;
--m; or          m--;
++ m;    is   equivalent to m = m + 1; (or m += 1;)
--m; is   equivalent to m = m - 1; (or m -= 1;)
```

We use the increment and decrement operators extensively in **for** and **while** loops. (See Chapter 7.)

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Program 5.3 illustrates this.

Program 5.3 *Increment operator illustrated*

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

(Contd.)

Program 5.3 *(Contd.)*

```
        int m = 10, n = 20;
        System.out.println(" m = " + m);
        System.out.println(" n = " + n);
        System.out.println(" ++m = " + ++m);
        System.out.println(" n++ = " + n++);
        System.out.println(" m = " + m) ;
        System.out.println(" n = " + n);
    }
}
```

The output of Program 5.3 would be:

```
m = 10
n = 20
++m = 11
n++ = 20
m = 11
n = 21
```

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10
```

is equivalent to

```
a[i]= 10
i   = i+1
```

5.7 CONDITIONAL OPERATOR

The character pair ? : is a ternary operator available in Java. This operator is used to con conditional expressions of the form

```
exp1 ? exp2 : exp3
```

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is non-zero (true), then the expres *exp2* is evaluated and becomes the value of the conditional expression. If *exp1* is false, ex evaluated and its value becomes the value of the conditional expression. Note that only one of expressions (either *exp2* or *exp3)* is evaluated. For example, consider the following statements:

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the if statement as follows:

```
if(a > b)
    x = a;
else
    x = b;
```

5.8 BITWISE OPERATORS

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to **float** or **double.** Table 5.7 lists the bitwise operators. They are discussed in detail in Appendix D.

Table 5.7 *Bitwise operators*

Operator	**Meaning**
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

5.9 SPECIAL OPERATORS

Java supports some special operators of interest such as **instanceof** operator and member selection operator (.).

Instanceof Operator

The **instanceof** is an object reference operator and returns ***true*** if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

Example:

```
    person        instanceof        student
```

is ***true*** if the object **person** belongs to the class **student;** otherwise it *is* ***false***.

Dot Operator

The dot operator (.) is used to access the instance variables and methods of class objects. Examples:

```
person1.age          //  Reference to the variable age
person1.salary( )    //  Reference to the method salary( )
```

It is also used to access classes and subpackages from a package.

5.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. Java can handle any complex mathematical expressions. Some of the examples of Java expressions are shown in Table 5.8. Remember that Java does not have an operator for exponentiation.

Table 5.8 *Expressions*

Algebraic expression	**Java expression**
ab–c	a*b–c
(m+n) (x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2+2x+l$	3*x*x+2*x+l
$\frac{x}{y}+c$	x/y+c

5.11 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form

```
variable = expression;
```

variable is any valid Java variable name. When the statement is encountered, the *expression* is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All

variables used in the expression must be assigned values before evaluation is attempted. Exa
evaluation statements are

```
x = a*b-c;
y = b/c*a;
z = a-b/c+d;
```

The blank space around an operator is optional and is added only to improve readabili these statements are used in program, the variables **a,b,c** and **d** must be defined before they a in the expressions.

5.12 TYPE CONVERSIONS IN EXPRESSIONS

Automatic Type Conversion

Java permits mixing of constants and variables of different types in an expression, but during eva it adheres to very strict rules of type conversion. We know that the computer, considers one ope a time, involving two operands. If the operands are of different types, the 'lower' type is autom converted to the 'higher' type before the operation proceeds. The result is of the higher type.

If **byte, short** and **int** variables are used in an expression, the result is always promoted to avoid overflow. If a single **long** is used in the expression, the whole expression is promoted to Remember that all integer values are considered to be **int** unless they have the 1 or L appen them. If an expression contains a **float** operand, the entire expression is promoted to float operand is **double**, result is **double**. Table 5.9 provides a reference chart for type conversion.

Table 5.9 *Automatic type conversion chart*

	char	**byte**	**short**	**int**	**long**	**float**	**dou**
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

The final result of an expression is converted to the type of the variable on the left of assignment sign before assigning the value to it. However, the following changes are introdu during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long** to **int** causes dropping of the excess higher order bits.

Casting a Value

We have already discussed how Java performs type conversion automatically. However, the are instances when we want to force a type conversion in a way that is different from the autom conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
ratio = female_number/male_number
```

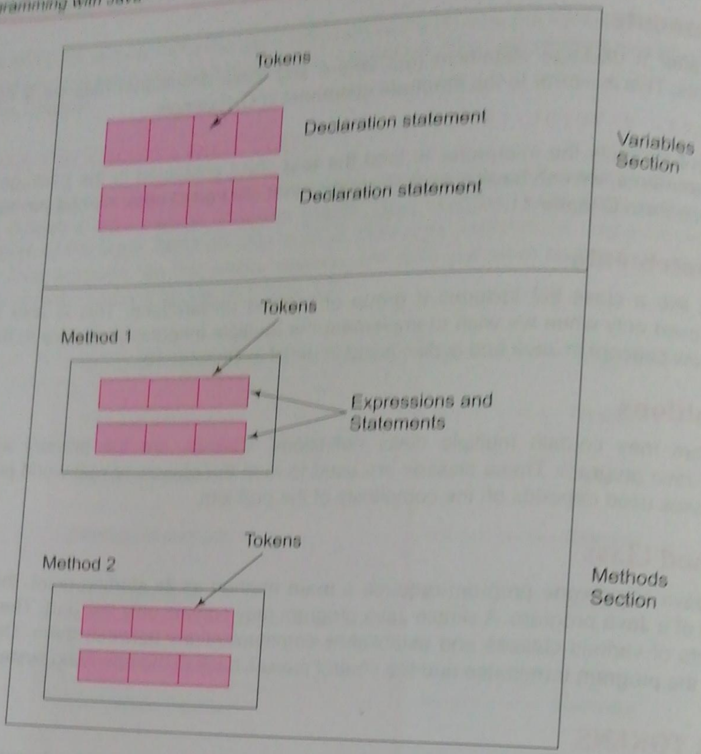


Fig. 3.3 *Elements of Java class*

Keywords

Keywords are an essential part of a language definition. They implement specific features of the language. Java language has reserved 50 words as keywords. Table 3.1 lists these keywords. These keywords, combined with operators and separators according to a syntax, form definition of the Java language. Understanding the meanings of all these words is important for Java programmers.

Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods, and so on. All keywords are to be written in lower-case letters. Since Java is case-sensitive, one can use these words as identifiers by changing one or more letters to upper case. However, it is a bad practice and should be avoided.

Identifiers

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

1. They can have alphabets, digits, and the underscore and dollar sign characters.
2. They must not begin with a digit.
3. Uppercase and lowercase letters are distinct.
4. They can be of any length.

We must save this program in a file called **Test.java** ensuring that the filename contains the c name properly. This file is called the *source file*. Note that all Java source files will have the exten **java**. Note also that if a program contains multiple classes, the file name must be the classname of class containing the **main** method.

Compiling the Program

To compile the program, we must run the Java Compiler **javac**, with the name of the source file on t command line as shown below:

```
javac Test.java
```

If everything is **OK**, the **javac** compiler creates a file called **Test.class** containing the bytecodes o the program. Note that the compiler automatically names the bytecode file as

```
<classname> .class
```

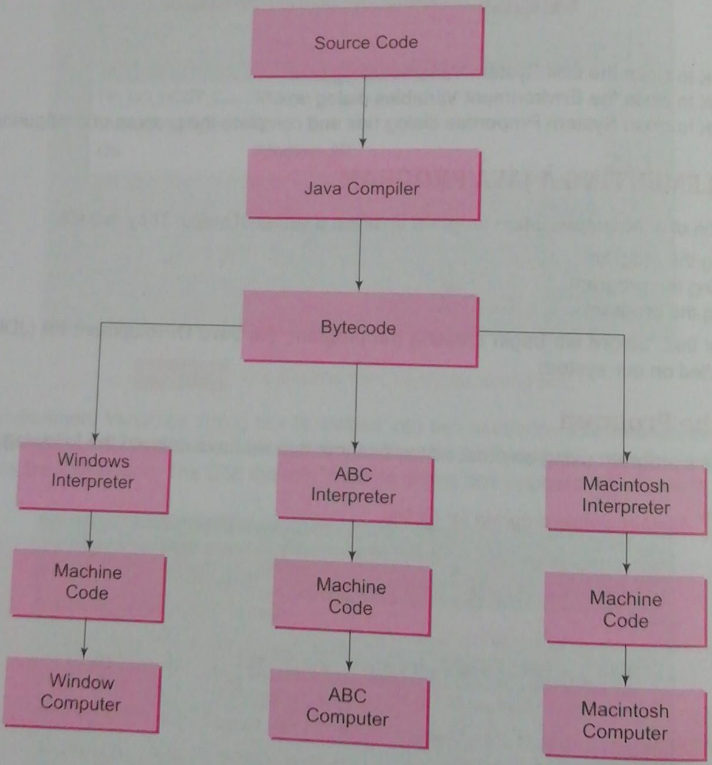


Fig. 3.15 *Implementation of Java programs*

Running the Program

We need to use the Java interpreter to run a standalone program. At the command prompt, type

```
java Test
```

Now, the interpreter looks for the main method in the program and begins execution from there. When executed, our program displays the following:

```
Hello!
Welcome to the world of Java.
Let us learn Java.
```

Note that we simply type "Test" at the command line and not "Test.class" or "Test.java".

Machine Neutral

The compiler converts the source code files into bytecode files. These codes are machine-independent and therefore can be run on any machine. That is, a program compiled on an IBM machine will run on a Macintosh machine.

Java interpreter reads the bytecode files and translates them into machine code for the specific machine on which the Java program is running. The interpreter is therefore specially written for each type of machine. Figure 3.15 illustrates this concept.

3.10 JAVA VIRTUAL MACHINE

All language compilers translate source code into *machine code* for a specific computer. Java compiler also does the same thing. Then, how does Java achieve architecture neutrality? The answer is that the Java compiler produces an intermediate code known as *bytecode* for a machine that does not exist. This machine is called the *Java Virtual Machine* and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer. Figure 3.16 illustrates the process of compiling a Java program into bytecode which is also referred to as *virtual machine code*.

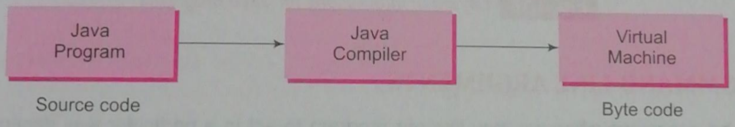


Fig. 3.16 *Process of compilation*

The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in Fig. 3.17. Remember that the interpreter is different for different machines.

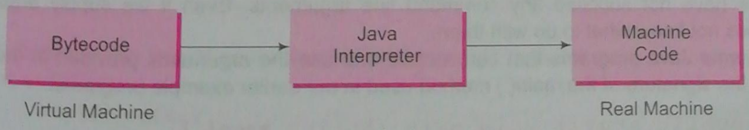


Fig. 3.17 *Process of converting bytecode into machine code*

4 Constants, Variables, and Data Types

KEY TERMS

Data | Information | Syntax | Constants | Variables | Integer | Decimal | Octal | Hexadecimal | Real constants | Floating point constants | Character constants | Backslash characters | Reference types | Boolean | NaN | Initialization | Scope | Instance variables | Class variables | Local variables | Nesting | Casting | Widening | Narrowing

4.1 INTRODUCTION

A programming language is designed to process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing data is accomplished by executing a sequence of instructions constituting a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must conform precisely to the syntax rules of the language.

Like any other language, Java has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to Java language.

4.2 CONSTANTS

Constants in Java refer to fixed values that do not change during the execution of a program. Ja supports several types of constants as illustrated in Fig. 4.1.

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *deci* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional minus sign. V examples of decimal integer constants are:

```
123   -321    0    654321
```

Embedded spaces, commas, and non-digit characters are not permitted between digits. For examp

```
15   750  20.000    $1000
```

are illegal numbers.

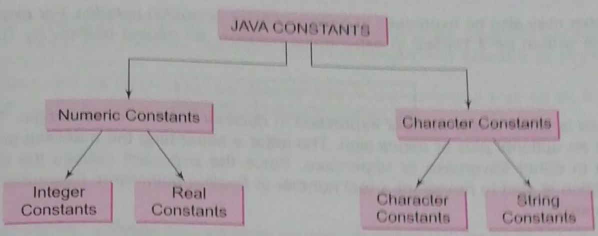


Fig. 4.1 *Java constants*

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

```
037      0      0435      0551
```

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer (hex integer). They may also include alphabets A through F or a through f. A letter A through F represents the numbers 10 through 15. Following are the examples of valid hex integers.

```
0X2      0X9F      0xbcd      0x
```

Java SE 7 introduces some language enhancements for defining integer constants. These are:

- Binary literals
- Numeric literals with underscore

Binary literals Just like octal and hexadecimal number systems, integer types can now be expressed in binary number system as well. The following example depicts how integer constant is defined in binary number system:

```
int num1 = 0b01010101;
int num2 = 0B10101010;
```

As shown above, '0b' is prefixed to the value representing binary number. Here, 'b' is case insensitive.

Numeric literals with underscore To enhance readability of large integers, Java 7 allows inserting underscores within the integer constants to mark the place values. Here's an example:

```
int num1 = 1_000;
long num2 = 1_000_000_000L;
```

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

```
0.0083     -0.75     435.36
```

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part, which is an integer. It is possible that the number may not have digits before the decimal point or digits after the decimal point. That is,

```
215.       .95       -.71
```

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific*) *notation*. For example, the va 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The gene form is

```
mantissa e exponent
```

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The exponen an integer with an optional *plus* or *minus* sign. The letter *e* separating the mantissa and the expone can be written in either lowercase or uppercase. Since the exponent causes the decimal point "float", this notation is said to represent a real number in *floating point form*. Examples of legal floati point constants are:

```
0.65e4    12e-2    1.5e+5    3.18E3    -1.2E-1
```

Embedded white (blank) space is not allowed, in any numeric constant.

Exponential notation is useful for representing numbers that are either very large or very small magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, –0.000000368 equivalent to –3.68E-7.

A floating point constant may thus comprise four parts:

1. a whole number
2. a decimal point
3. a fractional part
4. an exponent

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Examples of character constants are:

```
'5'    'X'    ';'    ' '
```

Note that the character constant '5' is not the same as the *number* 5. The last constant is a blank space.

String Constants

A string constant is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces. Examples are:

"Hello Java"   "1997"   "WELL DONE"   "?...!"   "5+3"   "X"

Backslash Character Constants

Java supports some special backslash character constants that are used in output methods. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 4.1. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

Table 4.1 *Backslash Character Constants*

Constant	Meaning
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\"	single quote
'\"'	double quote
'\\'	backslash

4.3 VARIABLES

A *variable* is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. In Chapter 3, we had used several variables. For instance, we used variables **length** and **breadth** to store the values of length and breadth of a room.

A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program. Some examples of variable names are:

- average
- height
- total_height
- classStrength

As mentioned earlier, variable names may consist of alphabets, digits, the underscore(_) and dollar characters, subject to the following conditions:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct. This means that the variable **Total** is not the same as **total** or **TOTAL.**
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of any length.

4.4 DATA TYPES

Every variable in Java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its *data types.* The variety of data types available allow the programmer to select the type appropriate to the needs of the application. Data types in Java under various categories are shown in Fig. 4.2. Primitive types (also called *intrinsic* or *built-in* types) are discussed in detail in this chapter. Derived types (also known as *reference* types) are discussed later as and when they are encountered.

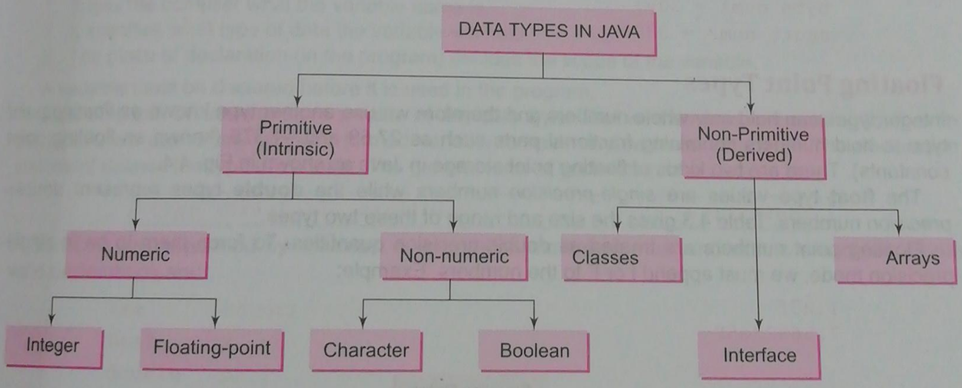


Fig. 4.2 *Data types in Java*

Integer Types

Integer types can hold whole numbers such as 123, –96, and 5639. The size of the values that can be stored depends on the integer data type we choose. Java supports four types of integers as shown in names when looking at an explicit reference to a class. We know that all class names, as per convention, begin with an uppercase letter. For example, look at the following statement:

```
double y = java.lang.Math.sqrt(x);
                ↑     ↑     ↑
          package  class  method
             name   name    name
```

This statement uses a fully qualified class name **Math** to invoke the method **sqrt( )**. Note that methods begin with lowercase letters. Consider another example:

```
java.awt.Point pts[ ];
```

This statement declares an array of **Point** type objects using the fully qualified class name.

Every package name must be unique to make the best use of packages. Duplicate names will cause run-time errors. Since multiple users work on Internet, duplicate package names are unavoidable. Java designers have recognized this problem and therefore suggested a package naming convention that ensures uniqueness. This suggests the use of domain names as prefix to preferred package names. For example:

```
cbe.psg.mypackage
```

Here **cbe** denotes city name and **psg** denotes organization name. Remember that we can create hierarchy of packages within packages by separating levels with dots.

11.5 CREATING PACKAGES

We have seen in detail how Java system packages are organised and used. Now, let us see how to create our own packages. We must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file (except comments and white spaces). Then we define a class, just as we normally define a class. Here is an example:

```
package firstPackage;                    / / package declaration
public class FirstClass                  / / class definition
{
    ............
    ............ (body of class)
    ............
}
```

Here the package name is **firstPackage**. The class **FirstClass** is now considered a part of this package. This listing would be saved as a file called **FirstClass.java**, and located in a directory named **firstPackage**. When the source file is compiled, Java will create a **.class** file and store it in the same directory.

Remember that the **.class** files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

To recap, creating our own package involves the following steps:

1. Declare the package at the beginning of a file using the form:

```
package packagename;
```

2. Define the class that is to be put in the package and declare it **public**.
3. Create a subdirectory under the directory where the main source files are stored.

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must watch out for catching are listed in Table 13.1.

Table 13.1 *Common Java exceptions*

Exception Type	*Cause of Exception*
`ArithmeticException`	Caused by math errors such as division by zero
`ArrayIndexOutOfBoundsException`	Caused by bad array indexes
`ArrayStoreException`	Caused when a program tries to store the wrong type of data in an array
`FileNotFoundException`	Caused by an attempt to access a nonexistent file
`IOException`	Caused by general I/O failures, such as inability to read from a file
`NullPointerException`	Caused by referencing a null object
`NumberFormatException`	Caused when a conversion between strings and number fails
`OutOfMemoryException`	Caused when there's not enough memory to allocate a new object
`SecurityException`	Caused when an applet tries to perform an action not allowed by the browser's security setting
`StackOverFlowException`	Caused when the system runs out of stack space
`StringIndexOutOfBoundsException`	Caused when a program attempts to access a nonexistent character position in a string

Exceptions in Java can be categorized into two types:

1. **Checked exceptions:** These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the **java.lang.Exception** class.
2. **Unchecked exceptions:** These exceptions are not essentially handled in the program code; instead the JVM handles such exceptions. Unchecked exceptions are extended from the java.lang. **RuntimeException** class.

It is important to note that checked and unchecked exceptions are absolutely similar as far as the functionality is concerned; the difference lies only in the way they are handled.

13.4 SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are throwing an exception and catching it. This is illustrated in Fig. 13.1.

Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements:

```
..........
..........
try
{
    statement;      // generates an exception
}
```