

UNIT – 2

Topics to be covered:

Java Basics: Data types, variables, identifiers, Keywords, Literals, Operators, Exceptions, Precedence Rules and Associativity, Type Conversion and Casting, Flow of Control: Branching, Conditional, Loops, classes and objects, creating objects, methods, constructors, constructor-overloading, Cleaning up unused objects and garbage collection, overloading methods and constructors, Class variables and methods, Static keyword, this keyword, Arrays and command line arguments

Data Types

Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict type. There are two reasons for this: First, every variable and expression must be defined using any one of the type. Second, the parameters to the method also should have some type and also verified for type compatibility. *Java language 8 primitive data types:*

The primitive data types are: char, byte, short, int, long, float, double, boolean. These are again grouped into 4 groups.

- 1. Integer Group:** The integer group contains byte, short, int, long. These data types will need different sizes of the memory. These are assigned positive and negative values. The width and ranges of these values are as follow:

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

byte:

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from -128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword.

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

```
byte b, c;
```

short:

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

```
short s;
short t;
```

int:

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. We can store byte and short values in an int.

Example

```
int x=12;
```

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.

Example

```
long x=123456;
```

2. Floating-Point Group

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. These are used with operations such as square root, cosine, and sine etc. There are two types of Floating-Point numbers: float and double. The float type represents single precision and double represents double precision. Their width and ranges are as follows:

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. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.

Example:

```
float height, price;
```

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 the math functions, such as **sin()**, **cos()**, and **sqrt()**, return **double** values.

Example:

```
double area,pi;
```

Example program to calculate the area of a circle

```
import java.io.*;
class Circle
{
    public static void main(String args[])
    {
        double r,area,pi;
        r=12.3;
        pi=3.14;
        area=pi*r*r;
        System.out.println("The Area of the Circle is:"+area);
    }
}
```

3. Characters Group

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages.

Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **chars**. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Here is a program that demonstrates char variables:

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

4. Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**.

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest
```

```

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

```

Identifiers

Identifiers are used for **class names, method names, and variable names**. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are:

Average	Height	A1	Area_Circle
---------	--------	----	-------------

Invalid Identifiers are as follow:

2types	Area-circle	Not/ok
--------	-------------	--------

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] ...] ;
```

Here the type is any primitive data types, or class name. The identifier is the name of the variable. We can initialize the variable by specifying the equal sign and value.

Example

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

Dynamic Initialization of the variable

We can also assign a value to the variable dynamically as follow:

```
int x=12;
int y=13;
float z=Math.sqrt(x+y);
```

The Scope and Lifetime of Variables

- ✓ Java allows, to declare a variable within any block.
- ✓ A block begins with opening curly brace and ended with end curly brace.
- ✓ Thus, each time we start new block, we create new scope.
- ✓ A scope determines what objects are visible to parts of your program. It also determines the life time of the objects.
- ✓ Many programming languages define two scopes: **Local and Global**
- ✓ As a general rule a variable defined within one scope, is not visible to code defined outside of the scope.
- ✓ Scopes can be also nested. The variable defined in **outer scope** are visible to the **inner scopes**, but reverse is not possible.

Example code

```
void function1()
{//outer block
    int a;
    //here a,b,c are visible to the inner scope
    int a=10;
    if(a==10)
    {// inner block
        int b=a*20;
        int c=a+30;
    }//end of inner block
    b=20*2;
    // b is not known here, which declared in inner scope
} //end of the outer block
```

Literals

A constant value can be created using a literal representation of it. Here are some literals:

int x=25;	char ch=88;	flaot f=12.34	byte b=12;
-----------	-------------	---------------	------------

Comments

In java we have three types of comments: single line comment, Multiple line comment, and document type comment.

Single line comment is represented with // (two forward slashes), Multiple comment lines represented with /*.....*/ (slash and star), and the document comment is represented with /**.....*/.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

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

The Java Keywords

There are **50 keywords** currently defined in the Java language (see Table below). These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method.

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

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 operands of the arithmetic operations are of numeric type. The Boolean operands are not allowed to perform arithmetic operations. The basic arithmetic operators are: addition, subtraction, multiplication, and division.

Example program to perform all the arithmetic operations

Arith.java

```
import java.io.*;
class Arith
{
    public static void main(String args[])
    {
        int a,b,c,d;
        a=5;
        b=6;
        //arithmetic addition
        c=a+b;
        System.out.println("The Sum is :"+c);
        //arithmetic subtraction
        d=a-b;
        System.out.println("The Subtraction is :"+d);
        //arithmetic division
        c=a/b;
        System.out.println("The Division is :"+c);
    }
}
```

```

        //arithmetic multiplication
        d=a*b;
        System.out.println("The multiplication is :"+d);
    }
}

```

The Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following demonstrates the %

Modulus.java

```

// Demonstrate the % operator.
class Modulus
{
    public static void main(String args[])
    {
        int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}

```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

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

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

```
a += 4;
```

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

Here is another example,

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

which can be expressed as

```
a %= 2;
```

Increment and Decrement

The ++ and the -- are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

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

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

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

Note: If we write increment/decrement operator after the operand such expression is called post increment/decrement expression, if written before operand such expression is called pre increment/decrement expression

The following program demonstrates the increment and decrement operator.

IncDec.java

```
// Demonstrate ++ and --
class IncDec
{
    public static void main(String args[])
    {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b; //pre increment
        d = a--; //post decrement
        c++; //post increment
        d--; //post decrement
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

The Bitwise Operators

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

These operators are again classified into 3 categories: **Logical operators, Shift operators, and Relational operator**

The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation. The bitwise operators are applied to each individual bit within each operand.

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

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

```
00101010
becomes
11010101
after the NOT operator is applied.
```

The Bitwise AND

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

```
  00101010  42
& 00001111  15
-----
  00001010  10
```

The Bitwise OR

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

```

00101010 42
| 00001111 15
-----
00101111 47

```

The Bitwise XOR

The XOR operator, `^`, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the `^`. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```

00101010 42
^ 00001111 15
-----
00100101 37

```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

BitLogic.java

```

// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{

int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b); int
g = ~a & 0x0f;
System.out.println(" a|b = " +c);
System.out.println(" a&b = " +d);
System.out.println(" a^b = " +e);
System.out.println("~a&b|a&~b = " + f);
System.out.println(" ~a = " + g);
}
}

```

Shift Operators: (left shift and right shift)

The Left Shift

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

```
value << num
```

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

The Right Shift

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

```
value >> num
```

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

ShiftBits.java

```
class ShiftBits
{
    public static void main(String args[])
    {
        byte b=6;
        int c,d;
        //left shift
        c=b<<2;
        //right shift
        d=b>>3;
        System.out.println("The left shift result is :"+c);
        System.out.println("The right shift result is :"+d);
    }
}
```

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

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Any type in Java, including **integers**, **floating-point** numbers, **characters**, and **Booleans** can be compared using the equality test, `==`, and the inequality test, `!=`. Notice that in Java equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is **greater or less than the other**.

Short-Circuit Logical Operators (`||` and `&&`)

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

When we use `||` operator if left hand side expression is true, then the result will be true, no matter what is the result of right hand side expression. In the case of `&&` if the left hand side expression results true, then only the right hand side expression is evaluated.

Example 1: `(expr1 || expr2)` Example2: `(expr1 && expr2)`

The Assignment Operator

The *assignment operator* is the single equal sign, `=`. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of *var* must be compatible with the type of *expression*. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

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

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the `?`. It can seem somewhat confusing at first, but the `?` can be used very effectively once mastered. The `?` has this general form:

expression1 `?` *expression2* `:` *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the `?` operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**. Here is an example of the way that the `?` is employed:

Test.java

```
class Test
{
public static void main(String args[])
{
    int x=4,y=6;
    int res= (x>y)?x:y;
    System.out.println("The result is :"+res);
}
}
```

Operator Precedence

Table shows the order of precedence for Java operators, from highest to lowest. Notice that the first row shows items that you may not normally think of as operators: **parentheses, square brackets, and the dot operator**. Technically, these are called *separators*, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects.

Highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
=	!=		
&			
^			
&&			
?:			
=	op=		
Lowest			

Using Parentheses

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

```
a >> b + 3
```

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

```
a >> (b + 3)
```

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

```
(a >> b) + 3
```

1. Control Statements

The control statements are used to control the flow of execution and branch based on the status of a program. The control statements in Java are categorized into 3 categories:

- i. **Selection statements**
- ii. **Iteration statements**
- iii. **Jump statements.**

- i. **The selection statements** include: **if** and **switch**. These are used to choose different paths of execution based on the outcome of the conditional expression.

if statement: This is the Java's conditional branch statement. This is used to route the execution through two different paths. The general form of the if statement will be as follow:

```
if (conditional expression)
{
    statement1
}
else
{
    statement2
}
```

Here the statements inside the block can be single statement or multiple statements. The conditional expression is any expression that returns the **Boolean** value. The else clause is optional. The if works as follows: if the conditional expression is true, then statement1 will be executed. Otherwise statement2 will be executed.

Example:

Write a java program to find whether the given number is even or odd?

EvenOdd.java

```
import java.io.*;
class EvenOdd
{
    public static void main(String args[])
    {
        int n;
        System.out.println("Enter the value of n");
        DataInputStream dis=new DataInputStream(System.in);
        n=Integer.parseInt(dis.readLine());
        if(n%2==0)
        {
            System.out.println(n+" is the Even Number");
        }
        else
        {
            System.out.println(n+" is the ODD Number");
        }
    }
}
```

Nested if: The nested if statement is an if statement, that contains another if and else inside it. The nested if are very common in programming. When we nest ifs, the else always associated with the nearest if.

The general form of the nested if will be as follow:

```

if(conditional expresion1)
{
    if(conditional expression2)
    {
        statements1;
    }
    else
    {
        satement2;
    }
}
else
{
    statement3;
}

```

Example program:

Write a java Program to test whether a given number is positive or negative.

Positive.java

```

import java.io.*;
class Positive
{
public static void main(String args[]) throws IOException
{
    int n;
    DataInputStream dis=new DataInputStream(System.in);
    n=Integer.parseInt(dis.readLine());
    if(n>-1)
    {
        if(n>0)
            System.out.println(n+ " is positive no");
        }
        else
            System.out.println(n+ " is Negative no");
    }
}
}

```

The if-else-if Ladder

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

if(condition)

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

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed.

Example Program:

Write a Java Program to test whether a given character is Vowel or Consonant?

Vowel.java

```
import java.io.*;  
class Vowel  
{  
public static void main(String args[]) throws IOException  
{  
    char ch;  
    ch=(char)System.in.read();  
    if(ch=='a')  
        System.out.println("Vowel");  
    else if(ch=='e')  
        System.out.println("Vowel");  
  
    else if(ch=='i')  
        System.out.println("Vowel");  
    else if(ch=='o')  
        System.out.println("Vowel");  
    else if(ch=='u')  
        System.out.println("Vowel");  
    else  
  
        System.out.println("consonant");  
  
    }  
}
```

The Switch statement

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.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of -jumping out|| of the **switch**.

Write a Java Program to test whether a given character is Vowel or Consonant? (Using Switch)

SwitchTest.java

```
import java.io.*;
class SwitchTest
{
    public static void main(String args[]) throws IOException
    {
        char ch;
        ch=(char)System.in.read();
```

```
switch(ch)
{
  //test for small letters
    case 'a': System.out.println("vowel");
        break;
    case 'e': System.out.println("vowel");
        break;
    case 'i': System.out.println("vowel");
        break;
    case 'o': System.out.println("vowel");
        break;
    case 'u': System.out.println("vowel");
        break;
  //test for capital letters
    case 'A': System.out.println("vowel");
        break;
    default: System.out.println("Consonant");
}
}
```

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **cases** without **break** statements between them. For example, consider the following program.

```
class Switch
{
public static void main(String args[])
{
    int month = 4;
    String season;
    switch (month)
    {
        case 12:
        case 1:
        case 2: season = "Winter";
                break;
        case 3:
        case 4:
        case 5: season = "Spring";
                break;
        case 6:
        case 7:
        case 8: season = "Summer";
                break;
    }
}
```

```

        case 9:
        case 10:
        case 11: season = "Autumn";
                break;
        default: season = "Bogus Month";
    }
    System.out.println("April is in the " + season + ".");
}
}

```

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```

switch(expression) //outer switch
{
    case 1: switch(expression) // inner switch
        {
            case 4: //statement sequence
                break;
            case 5: //statement sequence
                break;
        } //end of inner switch
        break;
    case 2: //statement sequence
        break;
    default: //statement sequence
} //end of outer switch

```

There are three important features of the **switch** statement to note:

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

2. 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.

i. **while**

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

```
while(condition)
{
    // body of loop
    increment or decrement statement
}
```

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:

Write a java program to add all the number from 1 to 10.

WhileTest.java

```
import java.io.*;
class WhileTest
{
    public static void main(String args[])
    {
        int i=1,sum=0;
        while(i<=10)
        {
            sum=sum+i;
            i++;
        }
        System.out.println("The sum is :"+sum);
    }
}
```

ii. **do-while statement**

However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.

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);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Example program:

Write a java program to add all the number from 1 to 10. (using do-while)

WhileTest.java

```
import java.io.*;  
class WhileTest  
{  
    public static void main(String args[])  
    {  
        int i=1,sum=0;  
        do  
        {  
            sum=sum+i;  
            i++;  
        } while(i<=10);  
        System.out.println("The sum is :"+sum);  
    }  
}
```

Note 1: Here the final value of the i will be 11. Because the body is executed first, then the condition is verified at the end.

Note 2: The **do-while** loop is especially useful when you process a **menu** selection, because you will usually want the body of a menu loop to execute at least once.

Example program: Write a Java Program to perform various operations like addition, subtraction, and multiplication based on the number entered by the user. And Also Display the Menu.

DoWhile.java

```
import java.io.*;  
class DoWhile  
{  
    public static void main(String args[]) throws IOException  
    {  
        int n,sum=0,i=0;
```

```
DataInputStream dis=new DataInputStream(System.in);

do
{
System.out.println("Enter your choice");
System.out.println("1 Addition");
System.out.println("2 Subtraction");
System.out.println("3 Multiplicaton");
n=Integer.parseInt(dis.readLine());
System.out.println("Enter two Numbers");
int a=Integer.parseInt(dis.readLine());
int b =Integer.parseInt(dis.readLine());
int c;
switch(n)
{
    case 1: c=a+b;
        System.out.println("The addition is :"+c);
        break;
    case 2: c=a-b;
        System.out.println("The addition is :"+c);
        break;
    case 3: c=a*b;
        System.out.println("The addition is :"+c);
        break;
    default:System.out.println("Enter Correct Number");

}

} while(n<=3);

}
}
```

iii. for statement

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. Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new `-for-each||` form. Both types of **for** loops are discussed here, beginning with the traditional form. Here is the general form of the traditional **for** statement:

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

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.

Example program: same program using the for loop

ForTest.java

```
import java.io.*;
class ForTest
{
    public static void main(String args[])
    {
        int i, sum=0;
        for(i=1;i<=10;i++)
        {
            sum=sum+i;
        }
        System.out.println("The sum is :"+sum);
    }
}
```

There are some important things about the for loop

1. The initialization of the loop controlling variables can be done inside the for loop.

Example:

```
for(int i=1;i<=10;i++)
```

2. We can write any boolean expression in the place of the condition for second part the loop.

Example: where b is a boolean data type

```
boolean b=false;
```

```
for(int i=1; !b;i++)
```

```
{
```

```
    //body of the loop
```

```
    b=true;
```

```
}
```

This loop executes until the b is set to the true;

3. We can also run the loop infinitely, just by leaving all the three parts empty.

Example:

```
for( ; ;)
```

```
{
```

```
    //body of the loop
```

```
}
```

For each version of the for loop:

The for loop also provides another version, which is called **Enhanced Version** of the for loop. The general form of the for loop will be as follow:

```

        for(type itr_var:collection)
        {
            //body of the loop
        }

```

Here, type is the type of the iterative variable of that receives the elements from collection, one at a time, from beginning to the end. The collection is created using the array.

Example program:

Write a java program to add all the elements in an array?

ForEach.java

```

import java.io.*;
class ForEach
{
    public static void main(String args[])
    {
        int i, a[], sum=0;
        a=new int[10];
        a={12,13,14,15,16};
        for(int x:a)
        {
            sum=sum+x;
        }
        System.out.println("The sum is :"+sum);
    }
}

```

3. The Jump Statements

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

i. break statement

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

Using break to Exit a Loop

By using **break**, 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:

// Using break to exit a loop.

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

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. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code.

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

<code>break label;</code>

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing bloc.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement.

Example code:

```
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.");
}
}
}

```

Running this program generates the following output:

```

Before the break.
This is after second block.

```

ii. continue statement

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. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses **continue** to cause two numbers to be printed on each line:

// Demonstrate continue.

```

class Continue
{
public static void main(String args[])
{
for(int i=1; i<=10; i++)
{
    if (i%5 == 0) continue;
    System.out.print(i + " ,");
}
}
}

```

Here all the numbers from 1 to 10 except 5 are printed. as 1,2,3,4,6,7,8,9,10.

iii. return statement

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**. As such, it is categorized as a jump statement.

Example code

```
class Test
{
    p s v main(String args[])    // caller of the method
    {
        int a=3,b=4;
        int x=method(a,b);
        System.out.println("The sum is :"+x);
    }
    int method(int x,int y)    // called method
    {
        return (x+y);
    }
}
```

After computing the result the control is transferred to the caller method, that main in this case.

Type Conversion and casting

There are two types of conversion. They are Implicit Conversion, Explicit Conversion.

Implicit Conversion

In the case of the implicit conversion type conversion is automatically performed by java when the types are compatible. For example, the **int** can be assigned to **long**. The **byte** can be assigned to **short**. However, not all the types are compatible, thus not all the type conversions are implicitly allowed. For example, double is not compatible with byte.

Conditions for automatic conversion

1. the two types must be compatible
2. the destination type must be larger than the source type

When automatic type conversion takes place the **widening conversion** takes place. For example,

```
int a; //needs 32 bits
byte b=45; //needs the 8 bits
a=b; // here 8 bits data is placed in 32 bit storage. Thus widening takes place.
```

Explicit Conversion

Fortunately, it is still possible to obtain the conversion between the incompatible types. This is called explicit type conversion. Java provides a special keyword "**cast**" to facilitate explicit conversion. For example, sometimes we want to assign int to byte, this will not be performed

automatically, because byte is **smaller** than int. This kind of conversion is sometimes called "*narrowing conversion*". Since, you are explicitly making the value narrow. The general form of the cast will be as follow:

```
destination_variable=(target type) value;
```

Here the target type specifies the destination type to which the value has to be converted.

Example

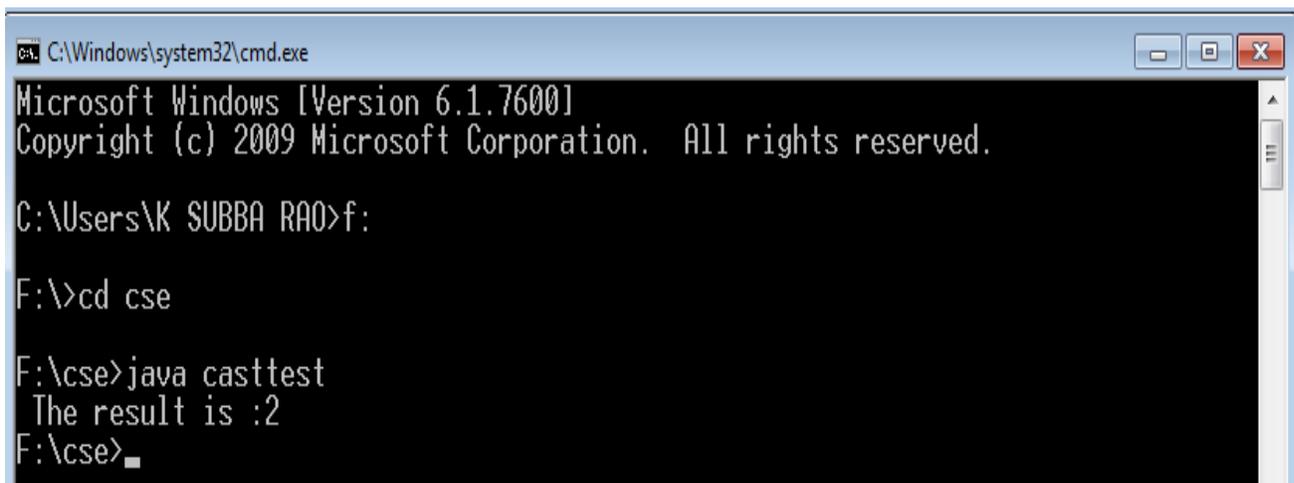
```
int a=1234;  
byte b=(byte) a;
```

The above code converts the int to byte. If the integer value is larger than the byte, then it will be reduced to modulo byte's range.

casttest.java

```
import java.io.*;  
class casttest  
{  
    public static void main(String args[])  
    {  
        int a=258;  
        byte b;  
        b=(byte) a;  
  
        System.out.print(" The result is :"+b);  
    }  
}
```

output:



```
C:\Windows\system32\cmd.exe  
Microsoft Windows [Version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
  
C:\Users\K SUBBA RAO>f:  
  
F:\>cd cse  
  
F:\cse>java casttest  
The result is :2  
F:\cse>_
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

Automatic Type Promotion in Expressions

The expression contains the three things: operator, operand and literals (constant). In an expression, sometimes the sub expression value exceeds the either operand.

For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

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

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

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

The code is attempting to store $50 * 2$, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast.

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

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

which yields the correct value of 100.

The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows: First,

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

Introduction to Arrays

An Array is a collection of elements that share the same type and name. The elements from the array can be accessed by the index. To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows:

```
type var_name[];
```

Here type declares the base type of the array. This base type determine what type of elements that the array will hold.

Example:

```
int month_days[];
```

Here type is int, the variable name is month_days. All the elements in the month are integers. Since, the base type is int.

In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

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

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

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

```
month_days = new int[10];
```

month_days

Element	0	0	0	0	0	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

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.

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

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

Element	0	28	0	0	0	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

The next line displays the value stored at index 3.

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

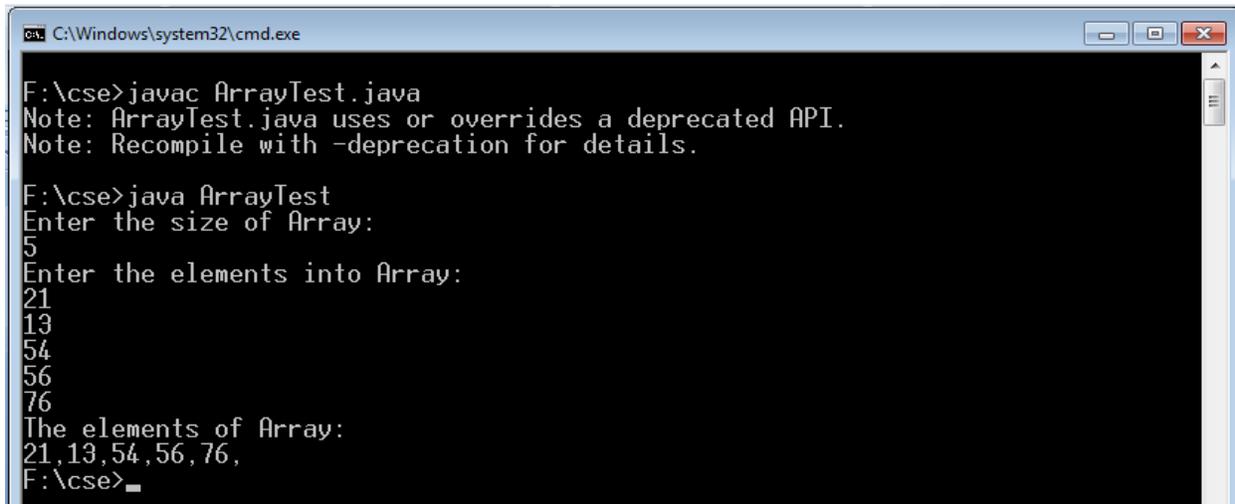
Example Program: Write a Java Program to read elements into array and display them?

ArrayTest.java

```
import java.io.*;
class ArrayTest
{
    public static void main(String args[]) throws IOException
    {
        DataInputStream dis=new DataInputStream(System.in);
        int a[]; //declaring array variable int n, i; //size
        of the array System.out.println("Enter the size
        of Array:");
        n=Integer.parseInt(dis.readLine());
        a=new int[n]; //allocating memry to array a and all the elements are set zero

        //read the elements into array
        System.out.println("Enter the elements into Array:");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(dis.readLine());
        }
        //displaying the elements
        System.out.println("The elements of Array:");
        for(i=0;i<n;i++)
        {
            System.out.print(a[i]+",");
        }
    }
}
```

Output



```

C:\Windows\system32\cmd.exe
F:\cse>javac ArrayTest.java
Note: ArrayTest.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.

F:\cse>java ArrayTest
Enter the size of Array:
5
Enter the elements into Array:
21
13
54
56
76
The elements of Array:
21,13,54,56,76,
F:\cse>

```

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][4];`

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

	Right Index Determines the Columns			
Left index determines the Rows	[0,0]	[0,1]	[0,2]	[0,3]
	[1,0]	[1,1]	[1,2]	[1,3]
	[2,0]	[2,1]	[2,2]	[2,3]
	[3,0]	[3,1]	[3,2]	[3,3]

Example Program for Matrix Addition

```

import java.io.*;
class AddMatrix
{
    public static void main(String args[]) throws IOException
    {
        int m, n, c, d;
        DataInputStream dis=new DataInputStream(System.in);

        System.out.println("Enter the number of rows and columns of matrix");

```

```

m = Integer.parseInt(dis.readLine());
n = Integer.parseInt(dis.readLine());

int first[][] = new int[m][n];
int second[][] = new int[m][n];
int sum[][] = new int[m][n];

System.out.println("Enter the elements of first matrix");

for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

System.out.println("Enter the elements of second matrix");

for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
        second[c][d] = Integer.parseInt(dis.readLine());

for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
        sum[c][d] = first[c][d] + second[c][d]; //replace '+' with '-' to subtract matrices

System.out.println("Sum of entered matrices:-");

for ( c = 0 ; c < m ; c++ )
{
    for ( d = 0 ; d < n ; d++ )
        System.out.print(sum[c][d]+" ");

    System.out.println();
}
}
}

```

Example program for Matrix Multiplication

```

import java.io.*;

class MulMatrix
{
    public static void main(String args[]) throws IOException
    {
        int m, n, p, q, sum = 0, c, d, k;

        DataInputStream dis = new DataInputStream(System.in);

```

```
System.out.println("Enter the number of rows and columns of first matrix");
m = Integer.parseInt(dis.readLine());
n = Integer.parseInt(dis.readLine());

int first[][] = new int[m][n];

System.out.println("Enter the elements of first matrix");

for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

System.out.println("Enter the number of rows and columns of second matrix");
p = Integer.parseInt(dis.readLine());
q = Integer.parseInt(dis.readLine());

if ( n != p )
    System.out.println("Matrices with entered orders can't be multiplied with each other.");
else
{
    int second[][] = new int[p][q];
    int multiply[][] = new int[m][q];

    System.out.println("Enter the elements of second matrix");

    for ( c = 0 ; c < p ; c++ )
        for ( d = 0 ; d < q ; d++ )
            second[c][d] = Integer.parseInt(dis.readLine());

    for ( c = 0 ; c < m ; c++ )
    {
        for ( d = 0 ; d < q ; d++ )
        {
            for ( k = 0 ; k < p ; k++ )
            {
                sum = sum + first[c][k]*second[k][d];
            }

            multiply[c][d] = sum;
            sum = 0;
        }
    }

    System.out.println("Product of entered matrices:-");

    for ( c = 0 ; c < m ; c++ )
```

```
{
  for ( d = 0 ; d < q ; d++ )
    System.out.print(multiply[c][d]+"\\t");

  System.out.print("\\n");
}
}
```

Alternative Array Declaration Syntax

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

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

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

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

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

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

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

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

Command Line arguments

The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt. The command line arguments can be accessed easily, because they are stored as Strings in String array passed to the args in the main method. The first command line argument is stored in args[0], the second argument is stored in args[1], the third argument is stored in args[2], and so on.

Example program:

Write a Java program to read all the command line arguments?

```
import java.io.*;
class CommnadLine
{
    public static void main(String args[])
    {
        for(int i=0;i<args.length();i++)
        {
            System.out.println(args[i]+" ");
        }
    }
}
```

Here the String class has a method length(), which is used to find the length of the string. This length can be used to read all the arguments from the command line.

Introduction to Strings

String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

- The **first thing** to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects.

For example, in the statement

```
System.out.println("This is a String, too");
```

the string `"This is a String, too"` is a **String** constant.

- The **second thing** to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:
- If you need to change a string, you can always create a new one that contains the modifications.
 - Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java

Strings can be created in a many ways. The easiest is to use a statement like this:

1. String initialization
String myString = "this is a test";
2. Reading from input device
DataInputStream dis=new DataInputStream(System.in);
String st=dis.readLine();

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

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

Java defines one operator for **String** objects: +. It is used to concatenate two strings. For example, this statement:

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing -I like Java.||

The **String** class contains several methods that you can use.

Here are a few.

1. **equals()** - used to test whether two strings are equal or not
2. **length()** –used to find the length of the string
3. **charAt(i)** - used to retrieve the character from the string at the index i.
4. **compareTo(String)** –returns 0, if the string lexicographically equals to the argument, returns greater than 0 if the argument is lexicographically greater than this string, returns less than 0 otherwise.
5. **indexOf(char)** –returns the index of first occurrence of the character.
6. **lastIndexOf(char)**- returns the last index of the character passed to it.
7. **concat(String)** -Concatenates the string with the specified argument

Example : **int n=myString.length(); //gives the length of the string**

Introduction to classes

Fundamentals of the class

A class is a group of objects that has common properties. It is a **template** or blueprint from which objects are created. The objects are the **instances** of class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

A class is used to define new type of the data. Once defined, this new type can be used to create objects of its type. The class is the logical entity and the object is the logical and physical entity.

The general form of the 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. A simplified general form of a **class** definition is shown here:

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    .....
    .....
    type instance-variableN;

    type method1(parameterlist)
    {
        //body of the method1
    }
    type method2(parameterlist)
    {
        //body of the method2
    }
    .....
    .....
    type methodN(parameterlist)
    {
        //body of the methodN
    }
}
```

- The data or variables, defined within the class are called, instance variable.
- The methods also contain the code.
- The methods and instance variable collectively called as members.
- Variable declared within the methods are called local variables.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines

three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods.

```
class Box
{
    //instance variables
    double width;
    double height;
    double depth;
}
```

As stated, a class defines new data type. The new data type in this example is, **Box**. This defines the template, but does not actually create object.

Creating the Object

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Step 1:

```
Box b;
```

Effect: b

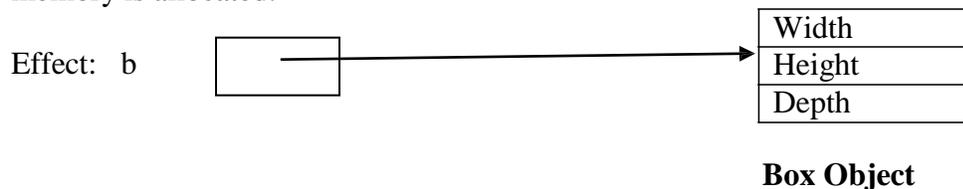
null

Declares the class variable. Here the class variable contains the value **null**. An attempt to access the object at this point will lead to Compile-Time error.

Step 2:

```
Box b=new Box();
```

Here new is the keyword used to create the object. The object name is b. The **new** operator allocates the memory for the object, that means for all instance variable inside the object, memory is allocated.



Step 3:

There are many ways to initialize the object. The object contains the instance variable. The variable can be assigned values with reference of the object.

```
b.width=12.34;
b.height=3.4;
b.depth=4.5;
```

Here is a complete program that uses the **Box** class:

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[])
    {
        //declaring the object (Step 1) and instantiating (Step 2) object
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables (Step 3)
        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);
    }
}
```

When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file.

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

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.
class Box
{
    double width;
    double height;
    double depth;
```

```
}
class BoxDemo2
{
public static void main(String args[])
{
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
// assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
// compute volume of first box
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Volume is " + vol);
// compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
}
}
```

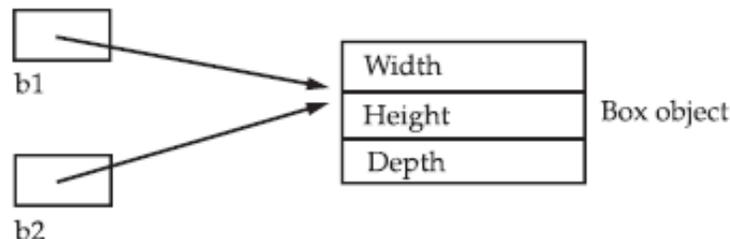
Assigning Object Reference Variables

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

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Introduction to Methods

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. In fact, much of the next chapter is devoted to methods.

This is the general form of a method:

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

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

Adding a method to the Box class

Box.java

```
class Box
{
    double width, height, double depth;
    // display volume of a box
    void volume()
    {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
```

Here the method name is "volume()". This methods contains some code fragment for computing the volume and displaying. This method can be accessed using the object as in the following code:

BoxDemo3.java

```
class BoxDemo3
{
    public static void main(String args[])
    {
        Box mybox1 = new Box();
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
    }
}
```

```
/* assign different values to mybox2's  
  
// display volume of first box  
mybox1.volume();  
// display volume of second box  
}  
}
```

Returning a Value

A method can also return the value of specified type. In this case the type of the method should be clearly mentioned. The method after computing the task returns the value to the **caller** of the method.

BoxDemo3.java

```
Box  
{  
    double width, height, depth;  
  
double volume()  
    {  
        return (width*height*depth);  
    }  
}  
  
class BoxDemo3  
{  
public static void main(String args[])  
{  
    Box mybox1 = new Box();  
    // assign values to mybox1's instance variables  
    mybox1.width = 10;  
    mybox1.height = 20;  
    mybox1.depth = 15;  
    double vol;  
/* assign different values to mybox2's  
//calling the method vol=  
mybox1.volume();  
System.out.println("the Volume is:"+vol);  
}  
}
```

Adding a method that takes the parameters

We can also pass arguments to the method through the object. The parameters separated with comma operator. The values of the actual parameters are copied to the formal parameters in the method. The computation is carried with formal arguments, the result is returned to the caller of the method, if the type is mentioned.

```
double volume(double w,double h,double d)
{
    width=w;
    height=h;
    depth=d;
    return (width*height*depth);
}
```

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even if we use some method to initialize the variable, it would be better this initialization is done at the time of the object creation.

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.

Example Program:

```
class Box
{
    double width;
    double height;
    double depth;
// This is the constructor for Box.
    Box()
    {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
double vol;
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
} }
```

Parameterized Constructors

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

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

```
*/
```

```
class Box
```

```
{
```

```
    double width;
```

```
    double height;
```

```
    double depth;
```

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

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

```
{
```

```
    width = w;
```

```
    height = h;
```

```
    depth = d;
```

```
}
```

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

```
double volume() {
```

```
    return width * height * depth;
```

```
}
```

```
}
```

```
class BoxDemo7 {
```

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

```
        // declare, allocate, and initialize Box objects
```

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

```
        Box mybox2 = new Box(3, 6, 9);
```

```
        double vol;
```

```
        // get volume of first box vol =
```

```
        mybox1.volume();
```

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

```
        // get volume of second box vol =
```

```
        mybox2.volume();
```

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

```
    }
```

```
}
```

Constructor-overloading

In Java it is possible to define two or more class constructors that share the same name, as long as their parameter declarations are different. This is called constructor overloading.

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

Example: All the constructors names will be same, but their parameter list is different.

OverloadCons.java

```
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

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*.

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs during the execution of your program. The main job of this is to release memory for the purpose of reallocation. Furthermore, different Java run-time implementations will take varying approaches to 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 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.

To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects

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

```
protected void finalize( )
{

// finalization code here

}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

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 supports **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. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// 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);
    }
}

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

The test() method is overloaded two times, first version takes no arguments, second version takes one argument. When an overloaded method is invoked, Java looks for a match between arguments of the methods. Method overloading supports **polymorphism** because it is one way that Java implements the `-one interface, multiple methods` paradigm.

static keyword

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.

To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

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

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

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

static variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12

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.

To better understand what **this** refers to, consider the following version of **Box()**:

```
// A redundant use of this.
Box(double w, double h, double d)
{
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Note: This is mainly used to hide the local variables from the instance variable.

Example:

```
class Box
{
    //instance variable
    double width, height, depth;

    Box(double width, double height, double depth)
    {
        //local variables are assigned, but not the instance variable
        width=width;
        height=height;
        depth=depth;
    }
}
```

To avoid the confusion, this keyword is used to refer to the instance variables, as follows:

```
class Box
{
    //instance variable
    double width, height, depth;

    Box(double width, double height, double depth)
    {
        //the instance variable are assigned through the this keyword.
        this.width=width;
        this.height=height;
        this.depth=depth;
    }
}
```