ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES: : TIRUPATI

Venkatapuram(Village),Renigunta(Mandal),Tirupati,Chitoor District, Andhra Pradesh-517520.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME OF THE FACULTY: A.MRINALINI

REGULATION: B.TECH II-II SEMESTER

SUBJECT: (15A05403) OBJECT ORIENTED PROGRAMMING USING THOUGH JAVA

Unit-2

UNIT II:
Operators:
Arithmetic Operators, The Bitwise Operators, Relational Operators, Boolean Logic operators, The assignment operator, The ? Operator, Operator Precedence, Using Parentheses.

Control Statements:
Java's selection Statements, Iteration statements, Jump Statements.
Introducing Classes:

Class Fundamentals, Declaring Objects, Assuming Object reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection, The Finalize() method, A Stack class. Overloading Methods, Using Object as Parameter, Argument Passing, Returning Objects, Recursion, Introducing Access control, Understanding static, Introducing Nested and Inner classes, Exploring the String class, Using Command line Arguments, Varargs: variable-Length Arguments.

Operators in java Language

Arithmetic operators

Operator	Use	Description
`+`	`x + y`	Adds `x` and `y`
`-`	`x - y`	Subtracts `y` from `x`
	`-x`	Arithmetically negates `x`
`*`	`x * y`	Multiplies `x` by `y`
`/`	`x / y`	Divides `x` by `y`
`%`	`x % y`	Computes the remainder of dividing `x` by `y`

Operator	Use	Description
`++`	`x++`	`y = x++;` is the same as `y = x; x = x + 1;`
	`++x`	`y = ++x;` is the same as `x = x + 1; y = x;`
`--`	`x--`	`y = x--;` is the same as `y = x; x = x - 1;`
	`--x`	`y = --x;` is the same as `x = x - 1; y = x;`

Java Example:

```
public class BasicArithmeticDemo
{
     public static void main(String[] args)
     {
         int number1 = 10;
         int number2 = 5;

         //calculating number1 + number2;
         int sum = number1 + number2;

         //calculating number1 - number2;
         int difference = number1 - number2;

         //calculating number1 * number2;
         int product = number1 * number2;

         //calculating number1 / number2;
         int quot = number1 / number2;

         //calculating number1 % number2;
         int rem = number1 % number2;

         //Displaying the values
         System.out.println("number1 : "+number1);
         System.out.println("number2 : "+number2);
         System.out.println("sum : "+sum);
         System.out.println("difference : "+difference);
         System.out.println("product : "+product);
         System.out.println("quot : "+quot);
         System.out.println("rem : "+rem);
     }
```

Relational Operators

Operator	Use	Description
`>`	`x > y`	`x` is greater than `y`
`>=`	`x >= y`	`x` is greater than or equal to `y`
`<`	`x < y`	`x` is less than `y`
`<=`	`x <= y`	`x` is less than or equal to `y`
`==`	`x == y`	`x` is equal to `y`
`!=`	`x != y`	`x` is not equal to `y`

```
class ComparisonDemo {

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

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

Bitwise Operators

Operator	Use	Evaluates to `true` if
`~`	`~x`	Bitwise complement of `x`
`&`	`x & y`	AND all bits of `x` and `y`
`|`	`x | y`	OR all bits of `x` and `y`
`^`	`x ^ y`	XOR all bits of `x` and `y`
`>>`	`x >> y`	Shift `x` right by `y` bits, with sign extension
`>>>`	`x >>> y`	Shift `x` right by `y` bits, with `0` fill
`<<`	`x << y`	Shift `x` left by `y` bits

public class BitwiseLogicalOpDemo {

public static void main(String[] args) {

//Integer bitwise logical operator

int a = 65; // binary representation 1000001

int b = 33; // binary representation 0100001

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= " + ~a);

}}

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
```

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. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand,bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63

Logical Boolean Operators

Operator	Use	Evaluates to `true` if
`&&`	`x && y`	Both `x` and `y` are `true`
`||`	`x || y`	Either `x` or `y` are `true`
`!`	`!x`	`x` is not `true`

Eg:-

```
public class BitwiseLogicalOpDemo {
public static void main(String[] args) {
//Integer bitwise logical operator
int a = 65; // binary representation 1000001
int b = 33; // binary representation 0100001
System.out.println("a && b= " + (a && b));
System.out.println("a | |b= " + (a | |b));
System.out.println("!a= " + !a);

}}
```

Assignment Operators

Operator	Use	Shortcut for
`=`	`x = y`	`x = y`
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`
`&=`	`x &= y`	`x = x & y` (also works for `boolean` values)
`|=`	`x |= y`	`x = x | y` (also works for `boolean` values)
`^=`	`x ^= y`	`x = x ^ y` (also works for `boolean` values)
`>>=`	`x >>= y`	`x = x >> y`
`>>>=`	`x >>>= y`	`x = x >>> y`
`<<=`	`x <<= y`	`x = x << y`

```
class AssignOptrDemo
{
  public static void main(String[] args)
   {
int a = 10, b = 15, c = 15;
System.out.println("Assignment and shortcut assignment operators");
System.out.println(" a = " + (a = 15));
System.out.println(" Addition  = " + (a += b));
System.out.println(" Subtraction  = " + (c -= b));
System.out.println(" Division = " + (a /= 2));
System.out.println(" Multiplication = " + (a *= 2));

  }
}
```

Other Operators

Operator	Use	Description
`()`	`(x + y) * z`	Require operator precedence
`?:`	`z = b ? x : y`	Equivalent to: `if (b) { z = x; } else { z = y; }`
`[]`	`array[0]`	Access array element
`.`	`str.length()`	Access object method or field
(*type*)	`int x = (int) 1.2;`	Cast from one type to another
`new`	`d = new Date();`	Create a new object
`instanceof`	`o instanceof String`	Check for object type, returning `boolean`

Short-Circuit Logical Operators :

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators.

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

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero

<u>The ? Operator</u>

The value of a variable often depends on whether a particular boolean expression is or is not true and on nothing else. For instance one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) {
  max = a;
}
else {
  max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of `if-else` that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

`(a > b) ? a : b;` is an expression which returns one of two values, `a` or `b`. The condition, `(a > b)`, is tested. If it is true the first value, `a`, is returned. If it is false, the second value, `b`, is returned. Whichever value is returned is dependent on the conditional test, `a > b`. The condition can be any expression which returns a Boolean value.

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
}
}
```

The output generated by this program is shown here:

```
Original value of a: 64
i and b: 256 0
```

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

```
int a = 35;
a = a >> 2; // a contains 8
```

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

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

Unsigned right shift Operator >>>

Unsigned right shift operator >>> is effectively same as >> except that it is unsigned, it fills the left most positions with bit 0 always. (Irrespective the sign of the underlying number)

For example, for positive numbers

1. 11001100 >>> 1 becomes 01100110 (shown in diagram)
2. 10000000 >>> 3 becomes 00010000 in binary
3. 256 >>> 3 becomes 256 / 2^3 = 16.

for Negative numbers -

1. -2 (signed value 4294967294) >>> 30 becomes 3
2. 11111111111111111111111111111110 >>> 30 becomes 00000000000000000000000000000011

Operator Precedence

Highest
++ (postfix) – – (postfix)
++ (prefix) – – (prefix) ~ ! + (unary) – (unary) (*type-cast*)
* / %
+ –
>> >>> <<
> >= < <= instanceof
== !=
&
^
|
&&
||
?:
–>
= op=
Lowest

Using Parentheses

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

a >> b + 3

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

a >> (b + 3)

However, if you want to first shift a right by b positions and then add 3 to that result, you

will need to parenthesize the expression like this:

(a >> b) + 3

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example, which of the following expressions is easier to read?

a | 4 + c >> b & 7

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

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively

affect your program.

unit-2
Control Statements

Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion.

Java's Selection Statements

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

if

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

if (*condition*) *statement1*;

else *statement2*;

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

The if works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;

//...

if(a < b) a = 0;

else b = 0;
```

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

Nested ifs

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

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

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-elseif* 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. The final **else** acts as a default condition; that is, if all other conditional tests fail,then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

eg:-

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;

if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

switch

The **switch** statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of 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
}
```

```
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

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 **case**s without **break** statements between them. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
public static void main(String args[]) {
for(int i=0; i<12; i++)
switch(i) {
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
```

```
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
}
```
This program generates the following output:
```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

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(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
break;
case 1: // no conflicts with outer switch
System.out.println("target is one");
break;
}
break;
case 2: // ...
```
Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement in the outer switch. The **count** variable is compared only with the list of cases at the outer level. If **count** is 1, then **target** is compared with the inner list cases.
In summary, there are three important features of the **switch** statement to note:
• The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. 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 **if**s.

Iteration Statements

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

while

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

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

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.
Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.
class While {
public static void main(String args[]) {
int n = 10;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

do-while

if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. 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.

```
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.

```
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n--;
} while(n > 0);
}
}
```

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
tick 0
```

for

The first is the traditional form that has been in use since the original version of Java. The second is the newer “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
}
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

```
// Declare a loop control variable inside the for.
class ForTick {
public static void main(String args[]) {
// here, n is declared inside of the for loop
for(int n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop. For example, consider the loop in the following program:

```
class Sample {
public static void main(String args[]) {
int a, b;
b = 4;
for(a=1; a<b; a++) {
System.out.println("a = " + a);
System.out.println("b = " + b);
b--;
}
}
}
```

using multiple variable declaration

```
// Using the comma.
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b--) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

In this example, the initialization portion sets the values of both **a** and **b**. The two commaseparated
statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

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

for loop without initialization and inc/dec

```
// Parts of the for loop can be empty.
class ForVar {
public static void main(String args[]) {
int i;
boolean done = false;
i = 0;
for( ; !done; ) {
System.out.println("i is " + i);
if(i == 10) done = true;
i++;
}
}
```

The For-Each Version of the for Loop

loop.

The general form of the for-each version of the **for** is shown here:

for(*type itr-var* : *collection*) *statement-block*

```
// Use a for-each style for loop.
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Jump Statements

Using break

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.");
}
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4

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

Using break as a Form of Goto

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

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a standalone block of code but it can also be a block that is the target of another statement

```
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.
```

Using continue

The **continue** statement performs such an action. 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=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

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

```
// Using continue with a label.
class ContinueLabel {
public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
for(int j=0; j<10; j++) {
if(j > i) {
System.out.println();
continue outer;
}
System.out.print(" " + (i * j));
}
}
System.out.println();
}
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return

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

```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return; // return to caller
System.out.println("This won't execute.");
}
}
```

The output from this program is shown here:

```
Before the return.
```

unit-2

Introducing Classes

The General Form of a Class

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

class *classname* {

```
type instance-variable1;

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

A Simple Class

```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

```
Volume is 3000.0
```

Declaring Objects

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

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

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

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

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

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object.

Assigning Object Reference Variables

```
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.

Introducing Methods

This is the general form of a method:

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

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. 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

```
class Box {
double width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// 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;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

Returning a Value

```
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo4 {
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;
// 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);
}
}
```

Adding a Method That Takes Parameters

```
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// 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);
}
}
```

As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

Constructors

class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when 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.

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

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

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

when you allocate an
object, you use the following general form:

class-var = new *classname* ( );

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

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

Parameterized Constructors

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

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

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

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.

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;
}
```

Instance Variable Hiding:using this

Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However,when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width,** for example, would have referred to the formal parameter, hiding the instance variable **width**.While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

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

Garbage Collection

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 sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

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.
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 Java run time calls the **finalize( )** method on the object.
The **finalize( )** method has this general form:

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

Here, the keyword **protected** is a specifier that limits access to **finalize( )**.

A Stack Class

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

unit-2

A Closer Look at Methods and Classes

Overloading Methods:

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.

```
class OverloadDemo {
void test() {
```

```
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return 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);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

eg:-2

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

Overloading Constructors

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

```
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;
}
}
```

As you can see, the **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

eg:-2

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

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

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equalTo(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.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 OverloadCons2 {
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);
Box myclone = new Box(mybox1); // create copy of mybox1
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 cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```