S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.
Department of Computer Science & Engineering

Course: Programming in Java(18CS653)

Module 2: Operators and Control Statements

Prof. Prasanna Patil
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

- Operators are classified as
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators

Arithmetic Operators

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

- Arithmetic operators are used in mathematical expressions.
- The operands of the arithmetic operators must be of a numeric type.
- You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String args[]) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

Output of the Program

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0

dc = 1.5
dd = -0.5
de = 0.5
```

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 example program demonstrates the %:

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

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators :

- Java provides special operators that can be used to combine an arithmetic operation with an assignment.
- The syntax is

  var op= expression;

- This version uses the += compound assignment operator.
- Ex : a = a % 2; which can be expressed as

  a %= 2;

- The compound assignment operators provide two benefits.
  - First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms.
  - Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

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--;
  ```

- These operators are unique in that they can appear both in *postfix form, where they* follow the operand as just shown, and *prefix form, where they precede the operand.*
- Ex: x = 42;

  y = ++x; In this case, y is set to 43.

  x = 42;

  y = x++; In this case, y is set to 42.

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

The output of this program follows:

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

Relational Operators

- The relational operators determine the relationship that one operand has to the other.
- Specifically, they determine equality and ordering.
- 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.

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

- Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=.
- Only numeric types can be compared using the ordering operators.

```
Class RelationOp {
    public static void main(String args[]) {
        int a = 4;
        int b = 1;
        boolean c = a < b;
        System.out.println(“c = " + c);
        if(a <= b)
                System.out.println(“a is smaller”);
        Else
                System.out.println(“b is smaller);
    }
}
```

Boolean Logical Operators

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

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

- The following table shows the effect of each logical operation:

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

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a = " + a);                    a = true
    System.out.println("        b = " + b);                    b = false
    System.out.println("      a|b = " + c);                  a|b = true
    System.out.println("      a&b = " + d);                  a&b = false
    System.out.println("      a^b = " + e);                  a^b = true
    System.out.println("!a&b|a&!b = " + f);            a&b|a&!b = true
    System.out.println("       !a = " + g);                   !a = false
  }
}
```

Short-Circuit Logical Operators

- Java will not bother to evaluate the right-hand operand when the outcome of the expression can determined by the left operand alone.
- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.
- For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
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.

The Assignment Operator

- The assignment operator is the single equal sign, =.
- 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 ?.
- 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.

```
// Demonstrate ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

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

- Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value.

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

Decimal to Binary Conversion

	128	64	32	16	8	4	2	1
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
4	0	0	0	0	0	1	0	0
11	0	0	0	0	1	0	1	1
26	0	0	0	1	1	0	1	0
97	0	1	1	0	0	0	0	1

- **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. Here is an example:

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

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    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 = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

```
        a = 0011
        b = 0110
      a|b = 0111
      a&b = 0010
      a^b = 0101
~a&b|a&~b = 0101
       ~a = 1100
```

- **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.

```
// Left shifting a byte value.
class ByteShift {
    public static void main(String args[]) {
        byte a = 64, b;
        int i;
        i = a << 2;     // 01000000
        b = (byte) (a << 2);
        System.out.println("Original value of a: " + a);
        System.out.println("i and b: " + i + " " + b);
    }
}
```

Output: 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.
- The >> operator automatically fills the high-order bit with its previous contents each time a shift occurs.

- The following code fragment shifts the value 32 to the right by two positions, resulting in **a being set to 8:**

Ex 1: int a = 32;

a =a >> 2;

00100000 32>>2

00001000 8

Ex 2 : int a = -8

a=a>>1

11111000 -8>>1

11111100 -4

- **The Unsigned Right Shift :**
- The >>> operator automatically fills the high-order bit with Zero each time a shift occurs.
- Ex : int a = -8

  a=a>>>1

  11111000 -8>>>1

  01111100 124

Bitwise Operator Compound Assignments

- All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.
- Ex : a = a >> 4;

  a >>= 4;
- Ex : a = a | b;

  a |= b;

```
class OpBitEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Operator Precedence

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)

- 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

Control Statements

Introduction

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

Java's Selection Statements

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

if statement

- 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.

- consider the following:

```
int a, b;
// ...
if(a < b)
        a = 0;
else
        b = 0;
```

- Some programmers find it convenient to include the curly braces when using the **if,** even when there is only one statement in each clause.

```
int bytesAvailable;
// ...
if (bytesAvailable > 0)
{
        ProcessData();
        bytesAvailable -= n;
}
else
        waitForMoreData();
        bytesAvailable = n;
```

Program to check the eligibility of voting

```
class Vote
{
    public static void main(String args[])
    {
        int age= 20;
        if(age >= 18)
            System.out.println(“Person is Eligible to vote”);
        else
            System.out.println(“Person is not Eligible to vote”);
    }
}
```

Nested ifs

- A nested **if** is an if statement that is the target of another if or else. Nested ifs are very common in programming.
- When you nest ifs, 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;
```

Program to find largest of 3 numbers using nested if statement

```
class Vote
{
  public static void main(String args[])
  {
    int a=10,b=20,c=15;
    if(a > b)
    {
       if(a > c)
          System.out.println(a + “is greater ”);
      else
          System.out.println(c + “is greater ”);
    }
else
 {
 if( b > c)
    System.out.println(b + “is greater ”);
 else
    System.out.println(c + “is greater ”);
 }
}
}
```

Output :
20 is greater

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.
- 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.

program that uses an if-else-if ladder to determine which season a particular month is in.

```
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 + ".");
}
}
```

Output :
April is in the Spring.

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.
- 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.

A simple example of the switch

```
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 cases without break statements between them.

```
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");
}
}
}
```

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
```

An improved version of the season 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 = "Summer"; break;
case 6:
case 7:
case 8:
season = “Spring";break;
case 9:
case 10:
case 11:
season = "Autumn";break;
default:
season = "Bogus Month";
}
System.out.println("April is in the " + season + ".");
}
}
```

Output:
April is in the Summer

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.

```
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 only compared 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 ifs.

Iteration Statements

- A loop repeatedly executes the same set of instructions until a termination condition is met.
- iteration statements are **for, while, and do-while.**

While loop

- 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.

Demonstrate the while loop

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

Output :

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

- Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

```
int a = 10, b = 20;
while(a > b)
   System.out.println("This will not be displayed");
```

- The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

While loop without body

```
class NoBody {
  public static void main(String args[]) {
    int i, j;
    i = 100;
    j = 200;
    // find midpoint between i and j
    while(++i < --j)
          ;    // no body in this loop
    System.out.println("Midpoint is " + i);
  }
}
```

Output
Midpoint is 150

do-while loop

- 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.

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

- Output :

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

- 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.

// Using a do-while to process a menu selection

```
class Menu {
public static void main(String args[])
throws java.io.IOException {
char choice;
do {
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.println(" 3. while");
System.out.println(" 4. do-while");
System.out.println(" 5. for\n");
System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');

System.out.println("\n");
switch(choice) {
case '1':
System.out.println("The if:\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" // ...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\n");
System.out.println("while(condition) {
System.out.println (“statement; }");
break;
case '4’:
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while (condition);");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
}}}
```

- Output

Help on:
1. if
2. switch
3. while
4. do-while
5. For

Choose one:
4
The do-while:
do {
statement;
} while (condition);

For loop

- Here is the general form of the traditional for statement:

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

- If only one statement is being repeated, there is no need for the curly braces.
- The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.
- Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Demonstrate the for loop.

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

Program to Test the Prime Number

```
class FindPrime {
public static void main(String args[])
{
int num;
boolean isPrime = true;
num = 14;
for(int i=2; i <= num/i; i++) {
   if((num % i) == 0) {
      isPrime = false;
      break;
}
}
if(isPrime)
      System.out.println("Prime");
else System.out.println("Not
      Prime");
}
}
```

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.

```
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--;
}
}
}
```

```
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);
}
}
}
```

Output:

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

Some for Loop Variations

- One of the most common variations involves the conditional expression.
- Specifically, this expression does not need to test the loop control variable against some target value.
- In fact, the condition controlling the for can be any Boolean expression.
- For example, consider the following fragment:

```
boolean done = false;
for(int i=1; !done; i++) {
// ...
if(interrupted()) done = true;
}
```

- Here is another interesting for loop variation.
- Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// 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++;
}
}
}
```

- Here is one more for loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the for empty.
- For example:

```
for( ; ; ) {
// …
}
```

The For-Each Version of the for Loop

- The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- The for-each style of for is also referred to as the enhanced for loop.
- The general form of the **for-each** version of the for is shown here:

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

- Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

- The for-each style for automates the preceding loop.
- Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array.
- Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

Example of 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);
}
}
```

- Output

```
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
```

- There is one important point to understand about the for-each style loop.
- Its iteration variable is “read-only” as it relates to the underlying array.
- An assignment to the iteration variable has no effect on the underlying array.

The for-each loop is essentially read-only.

```
class NoChange {
public static void main(String
    args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10 };
for(int x : nums) {
System.out.print(x + " ");
x = x * 10; // no effect on nums
}
System.out.println();
for(int x : nums)
System.out.print(x + " ");
System.out.println();
}
}
```

- The output, shown here, proves this point:

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

Iterating Over Multidimensional Arrays

- In Java, multidimensional arrays consist of arrays of arrays.
- This is important when iterating over a multidimensional array, because each iteration obtains the next array, not an individual element.
- Furthermore, the iteration variable in the for loop must be compatible with the type of array being obtained.

Use for-each style for on a two-dimensional array.

```
class ForEach3 {
public static void main(String args[]) {
int sum = 0;
int nums[][] = new int[3][5];
// give nums some values
for(int i = 0; i < 3; i++)
for(int j=0; j < 5; j++)
nums[i][j] = (i+1)*(j+1);
for(int x[] : nums) {
for(int y : x) {
System.out.println("Value is: " + y);
sum += y;
}
}
System.out.println("Summation: " +
    sum);
}
}
```

- Output :

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90

Search an array using for-each style for.

```
class Search {
public static void main(String args[]) {
int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
int val = 5;
boolean found = false;
// use for-each style for to search nums for val
for(int x : nums) {
if(x == val) {
found = true;
break;
}
}
if(found)
System.out.println("Value found!");
}
}
```

Nested Loops

- Java allows loops to be nested. That is, one loop may be inside another.
- For example, here is a program that nests **for loops:**

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<5; i++) {
for(j=i; j<5; j++)
System.out.print(“*");
System.out.println();
}
}
}
```

Jump Statements

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

Using break

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

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.

- // Using break to exit a loop.

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
if(i == 6) break; // terminate loop if i is 6
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

Using break as a Form of Goto

- The break statement can also be employed by itself to provide a “civilized” form of the goto statement.
- Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner.
- There are, however, a few places where the goto is a valuable and legitimate construct for flow control.
- 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:

  break label;
- 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 block.
- 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.
- Doing so causes execution to resume at the end of the labeled block

```
// Using break as a civilized form of goto.
class Break {
public static void main(String args[]) {
boolean t = true;
first: {
second: {
third: {
System.out.println("Before the break.");
if(t) break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}
}
}
```

Using continue

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

- // 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("");
}
}
}
```