Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture #6

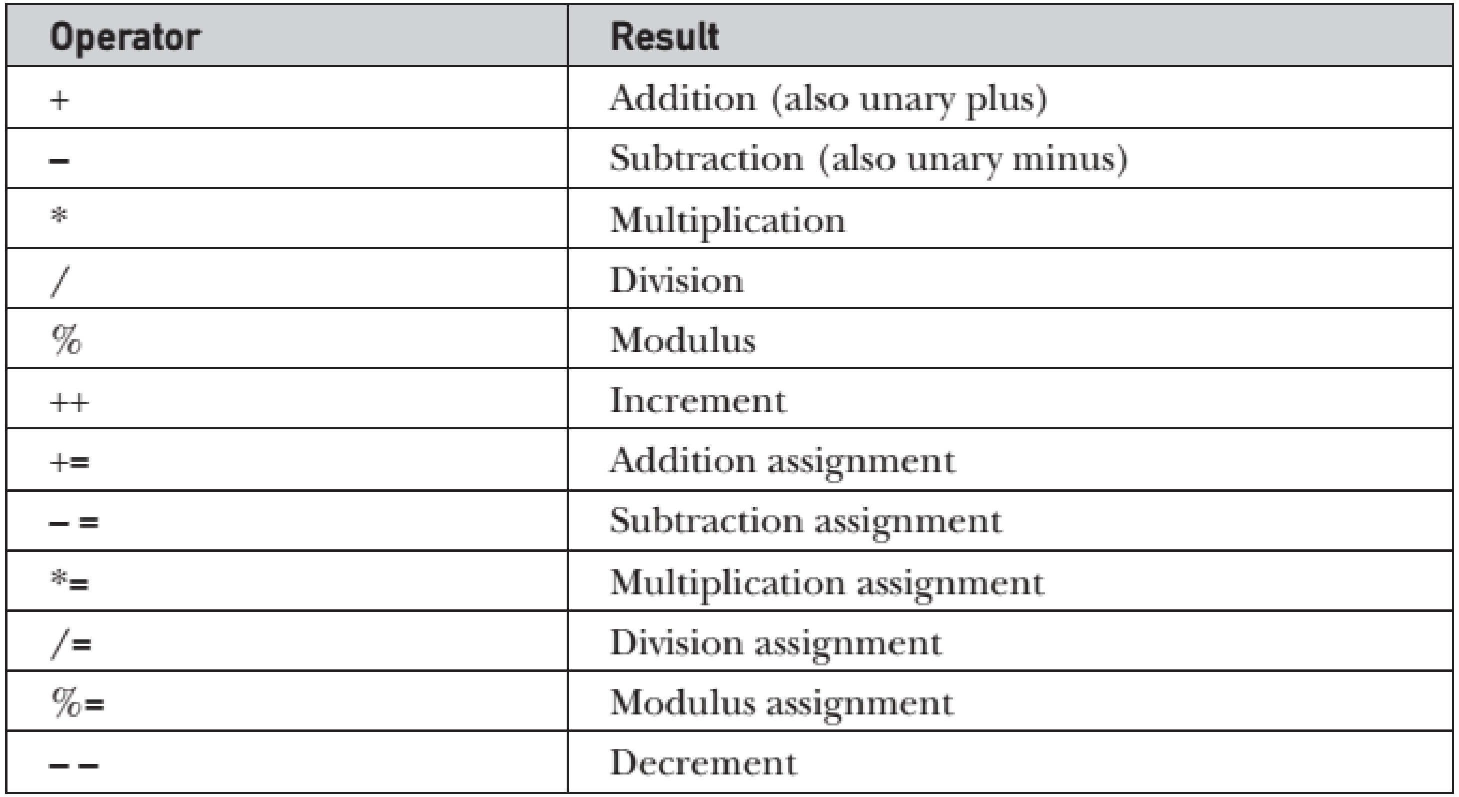
Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

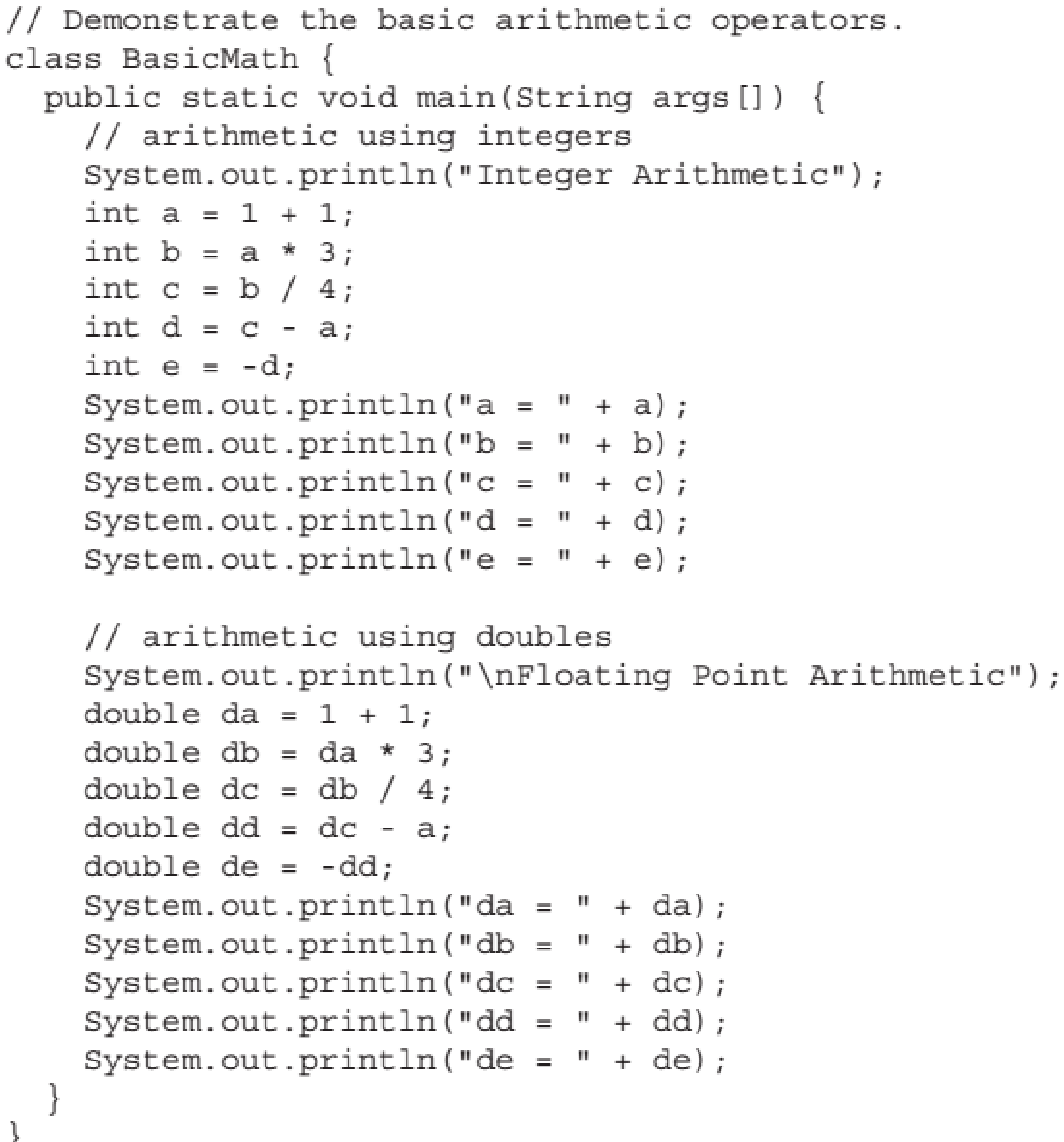
Operators

- Arithmetic Operators

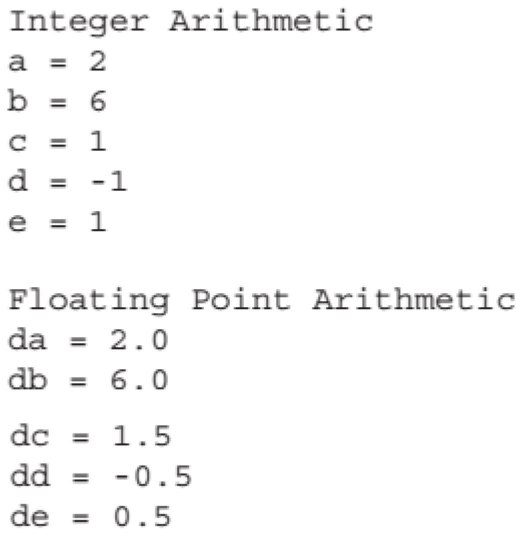


Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

Example #1

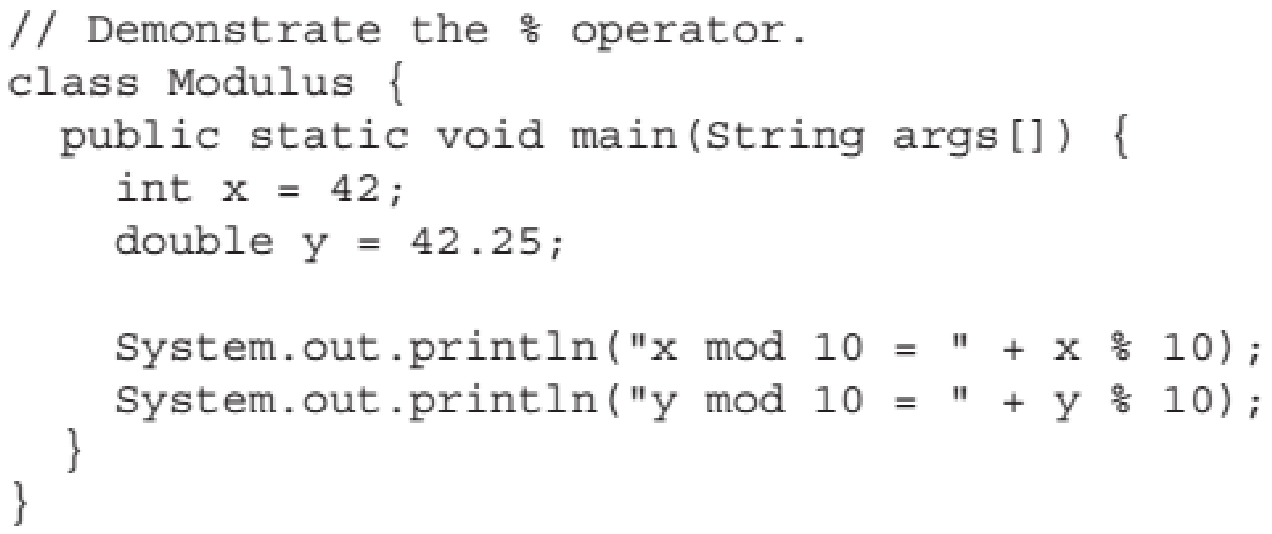


```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String args[]) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a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

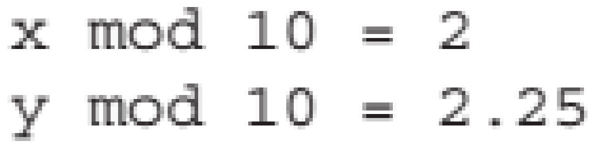


```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

Example #2
The Modulus Operator



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



```
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.
- As you probably know, statements like the following are quite common in programming:

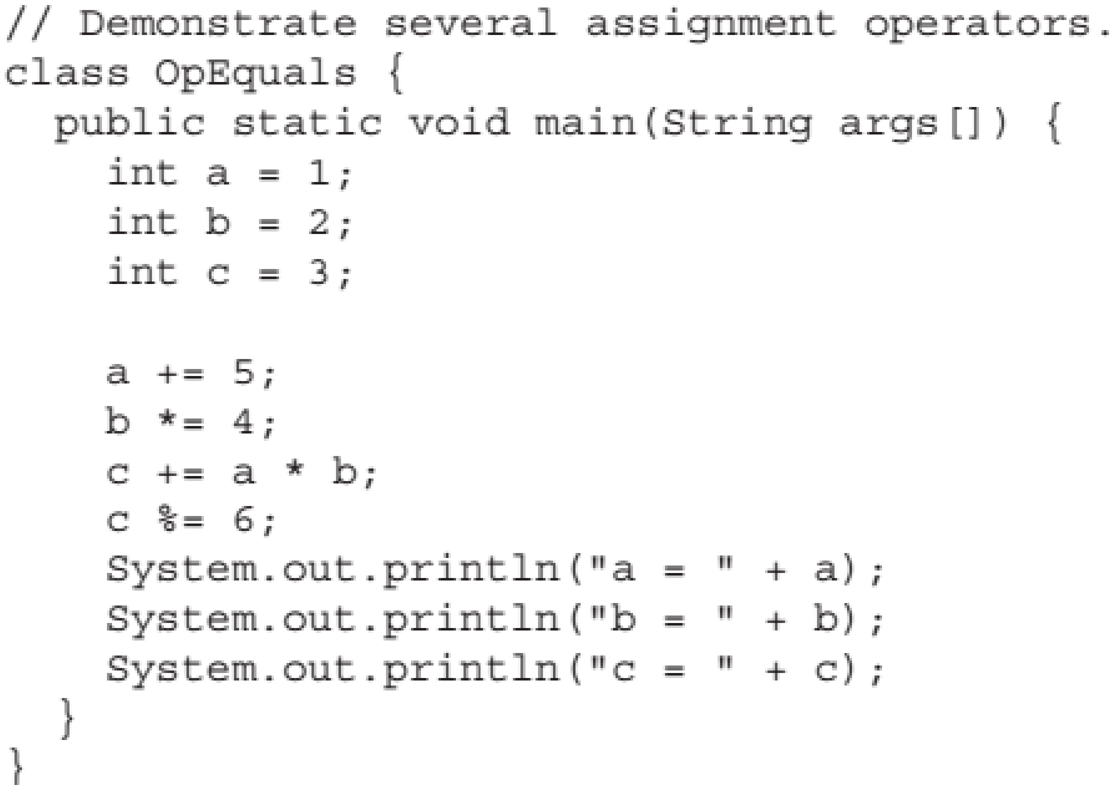
  a = a + 4;

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

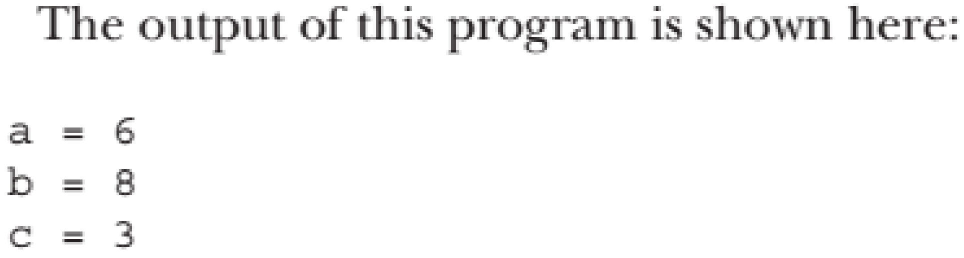
  a += 4;

  This version uses the += *compound assignment operator*.

Example #3



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

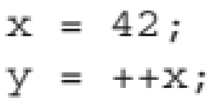


The output of this program is shown here:

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

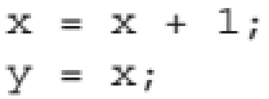
Increment and Decrement

- 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.

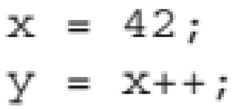


```
x = 42;
y = ++x;
```

the line **y** = **++x**; is the equivalent of these two statements:

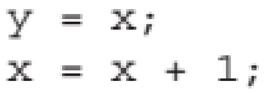


```
x = x + 1;
y = x;
```



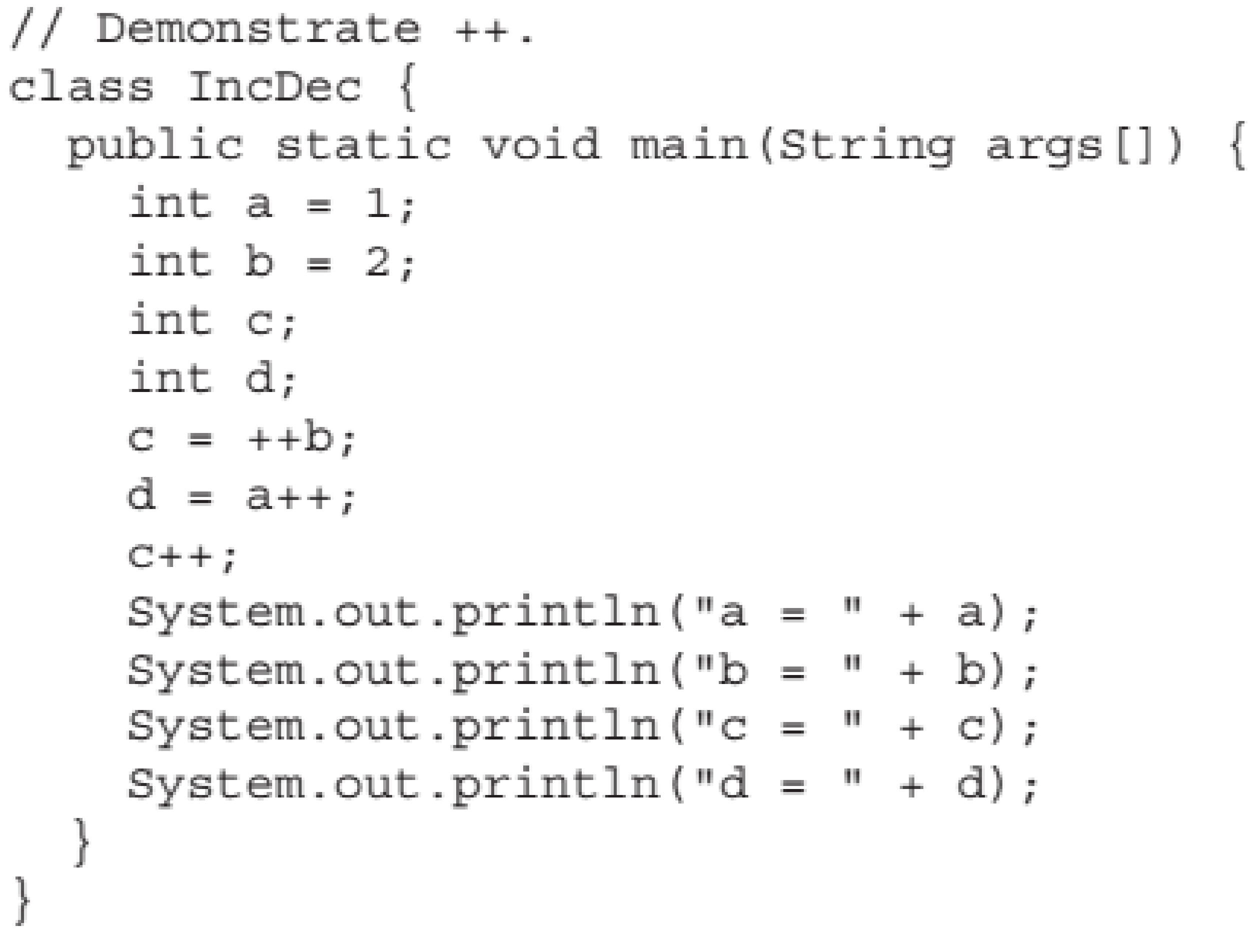
```
x = 42;
y = x++;
```

Here, the line **y = x++**; is the equivalent of these two statements:

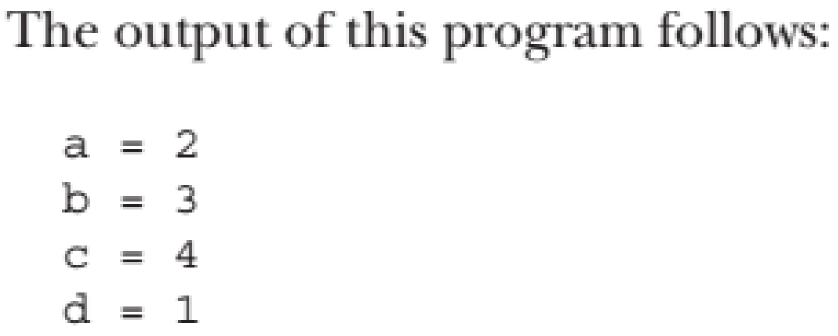


```
y = x;
x = x + 1;
```

Example #4



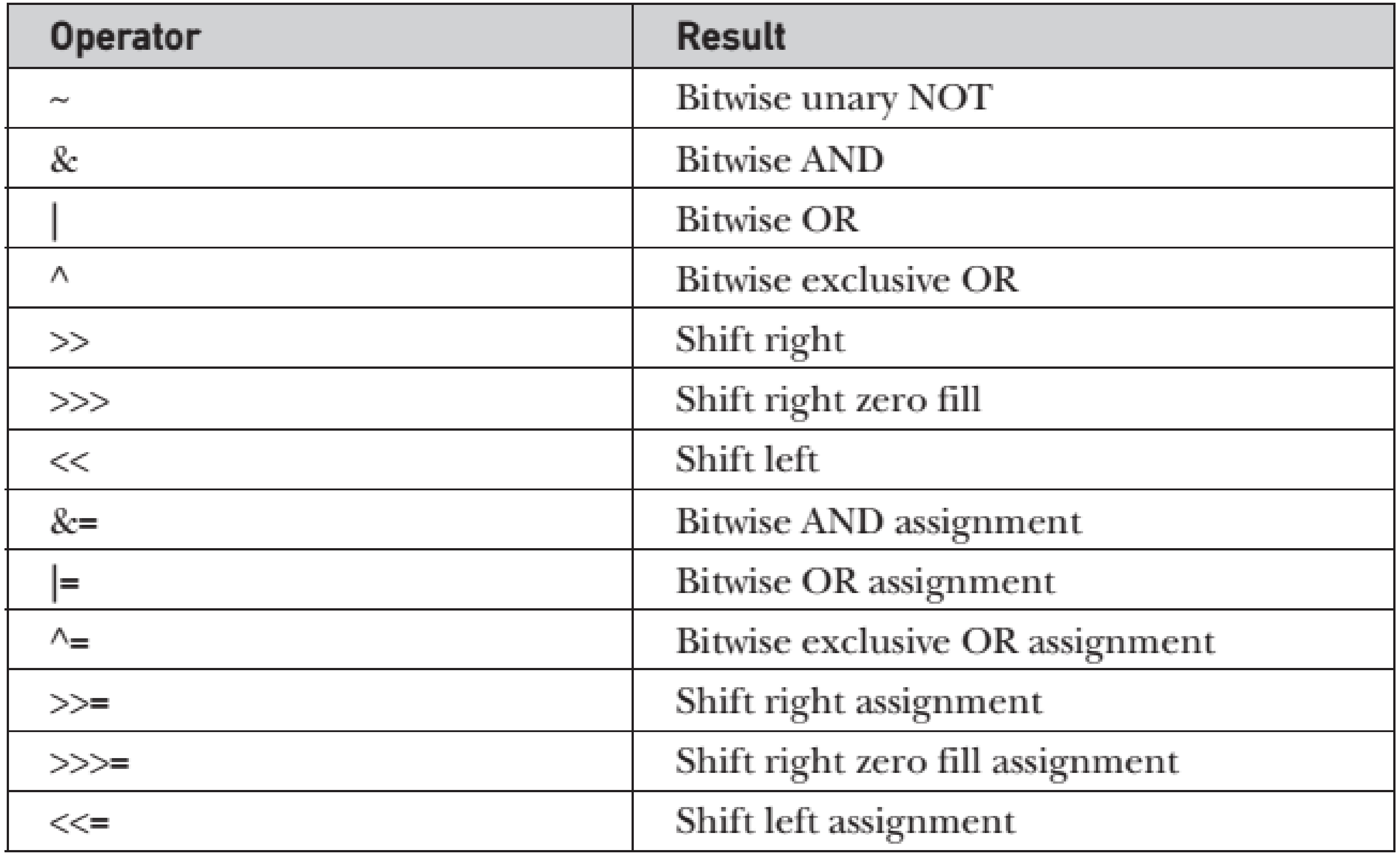
```
// Demonstrate ++.
class IncDec {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d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



The output of this program follows:

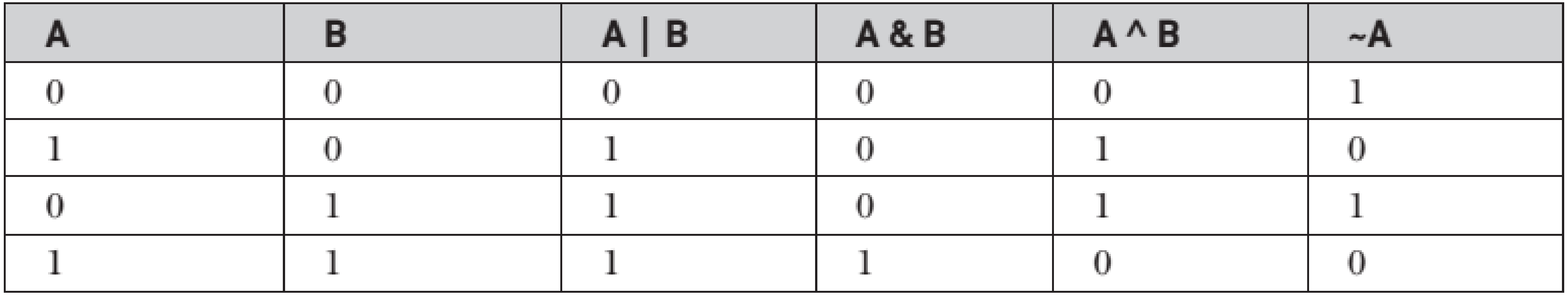
```
a = 2
b = 3
c = 4
d = 1
```

The Bitwise Operators

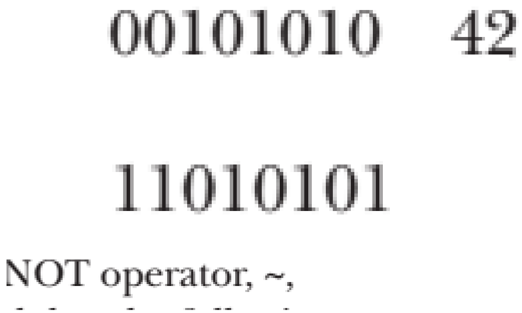


Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift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

The Bitwise Logical Operators

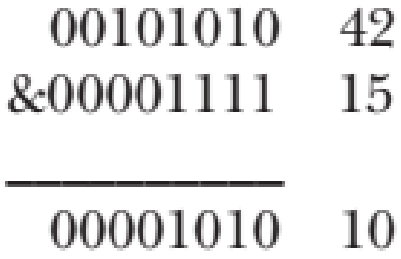


A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

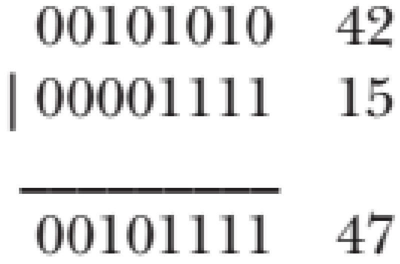


```
00101010    42

11010101
```

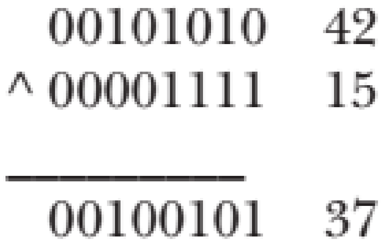
NOT operator, ~,



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

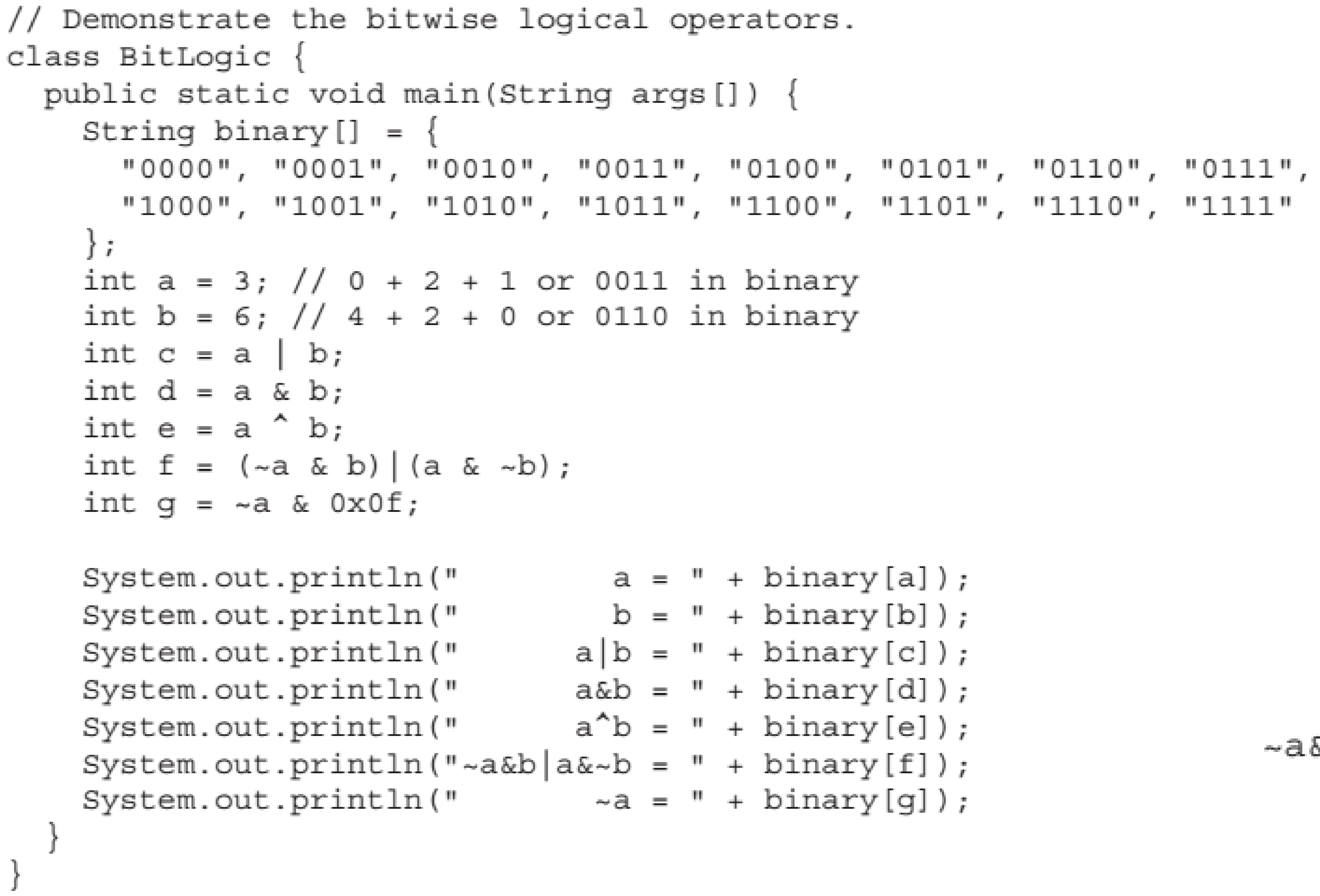


```
 00101010   42
|00001111   15
__________
 00101111   47
```

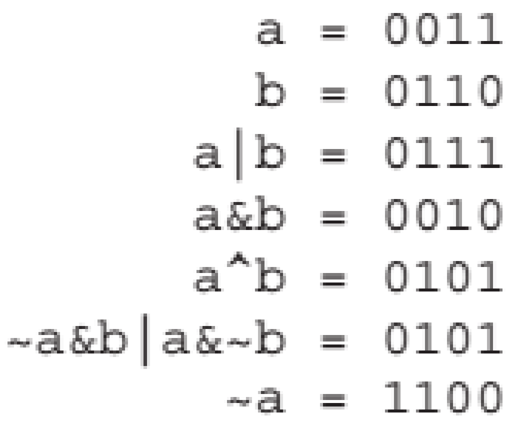


```
  00101010   42
^ 00001111   15
__________
  00100101   37
```

Example #5



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



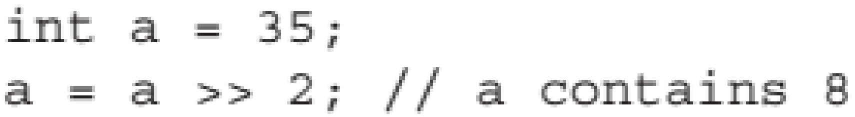
```
        a = 0011
        b = 0110
      a|b = 0111
      a&b = 0010
      a^b = 0101
~a&b|a&~b = 0101
       ~a = 1100
```

The Right Shift

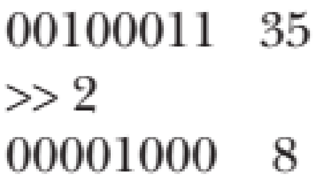
- The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here:

  value >> num

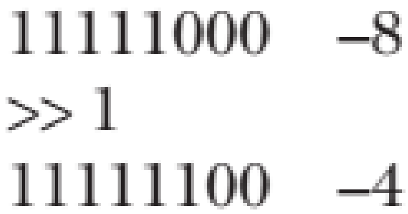
  Here, *num* specifies the number of positions to right-shift the value in *value*.
-



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

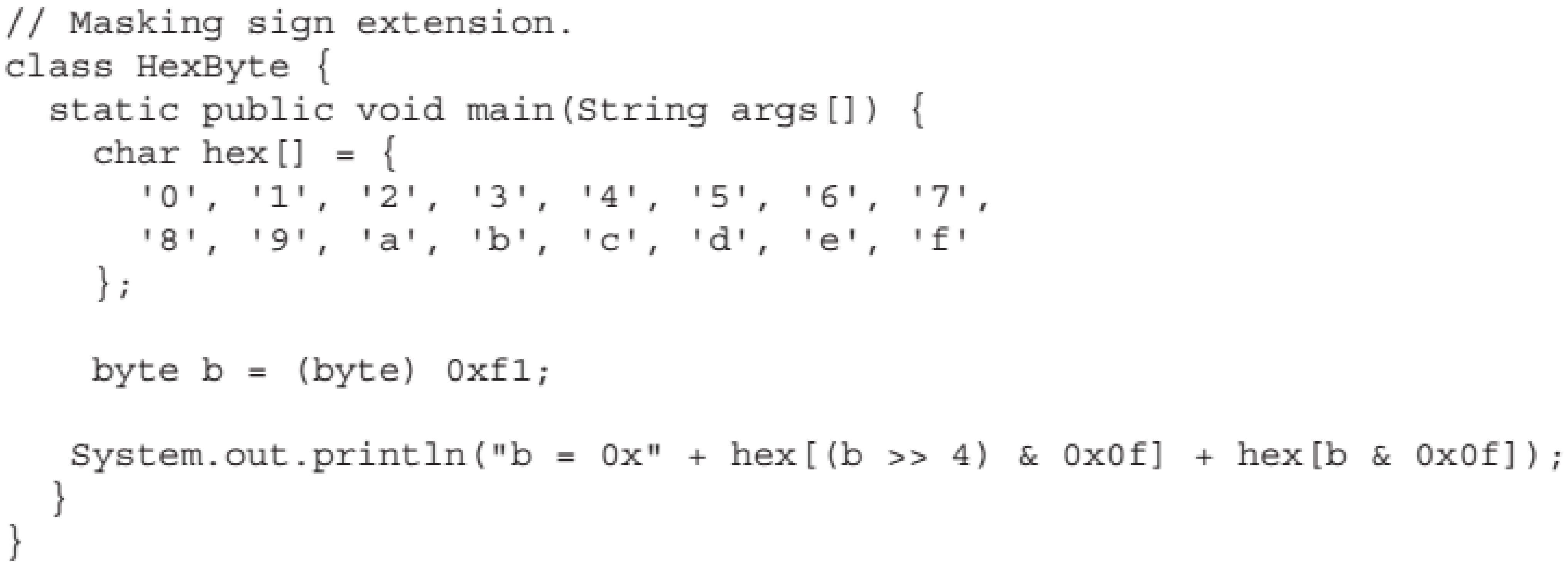


```
00100011  35
>> 2
00001000   8
```

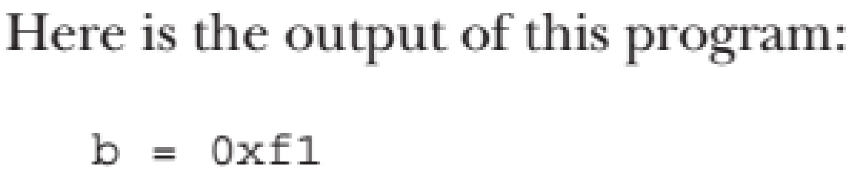


```
11111000  –8
>> 1
11111100  –4
```

Example #6



```
// Masking sign extension.
class HexByte {
  static public void main(String args[]) {
    char hex[] = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };

    byte b = (byte) 0xf1;

   System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
  }
}
```



Here is the output of this program:

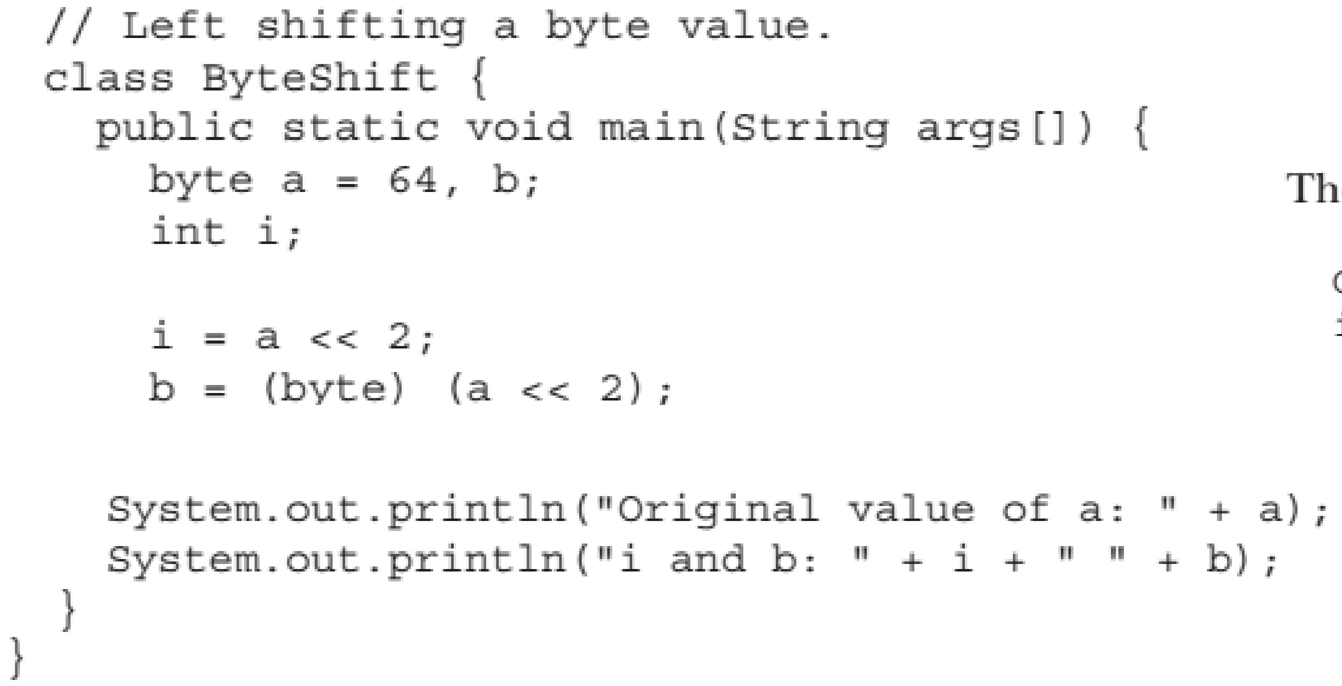
```
b = 0xf1
```

The Left Shift

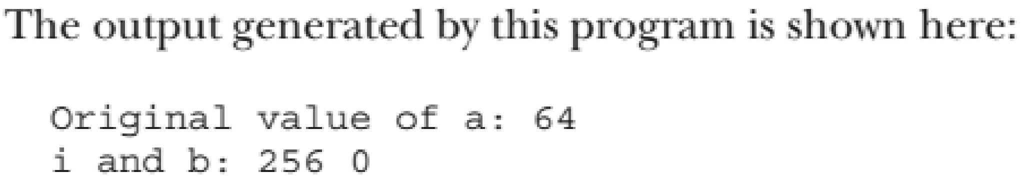
- The left shift operator, **<<,** shifts all of the bits in a value to the left a specified number of times.
- It has this general form:

  value << num

  Here, *num* specifies the number of positions to left-shift the value in *value*.



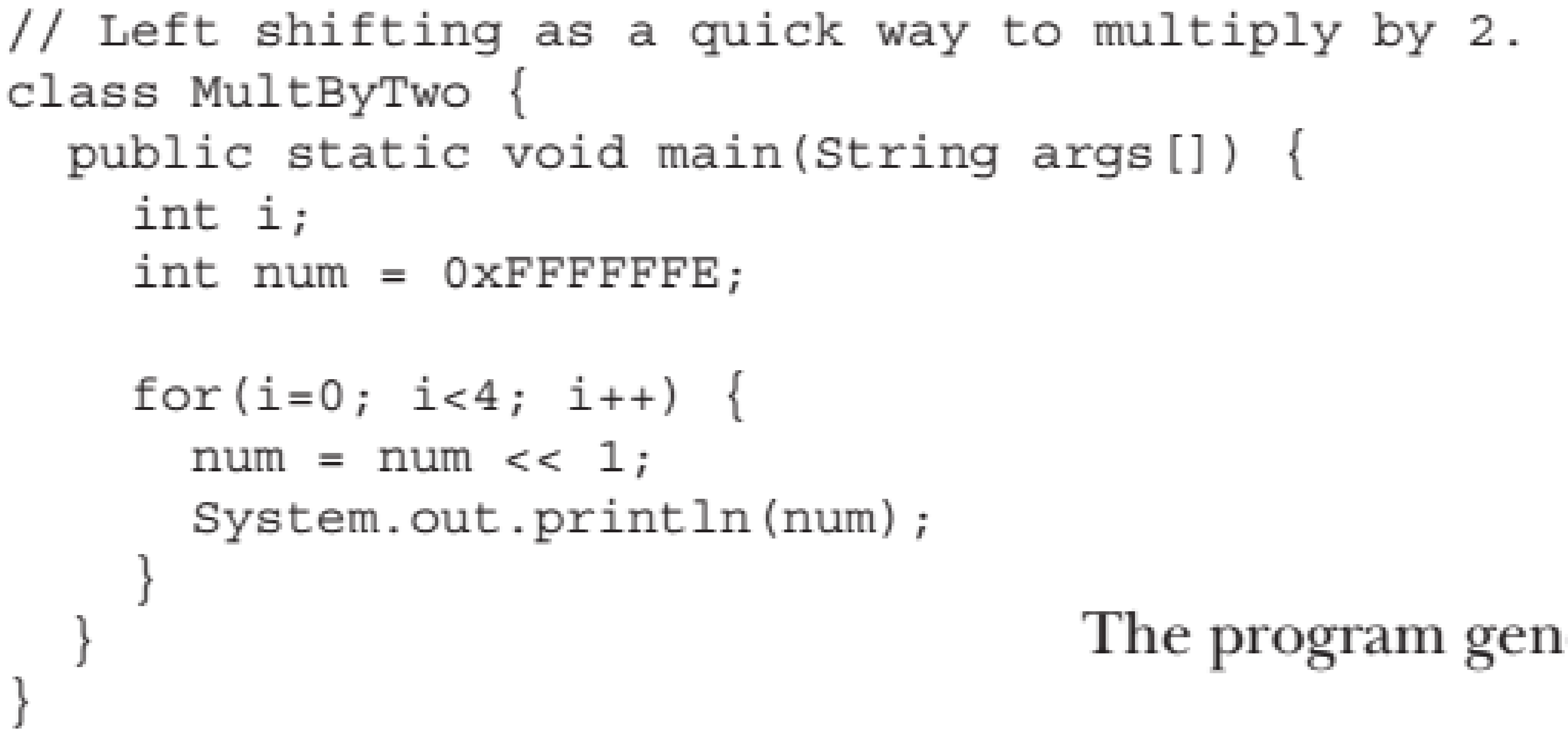
```
 // Left shifting a byte value.
 class ByteShift {
   public static void main(String args[]) {
     byte a = 64, b;
     int i;

     i = a << 2;
     b = (byte) (a << 2);


    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
  }
}
```



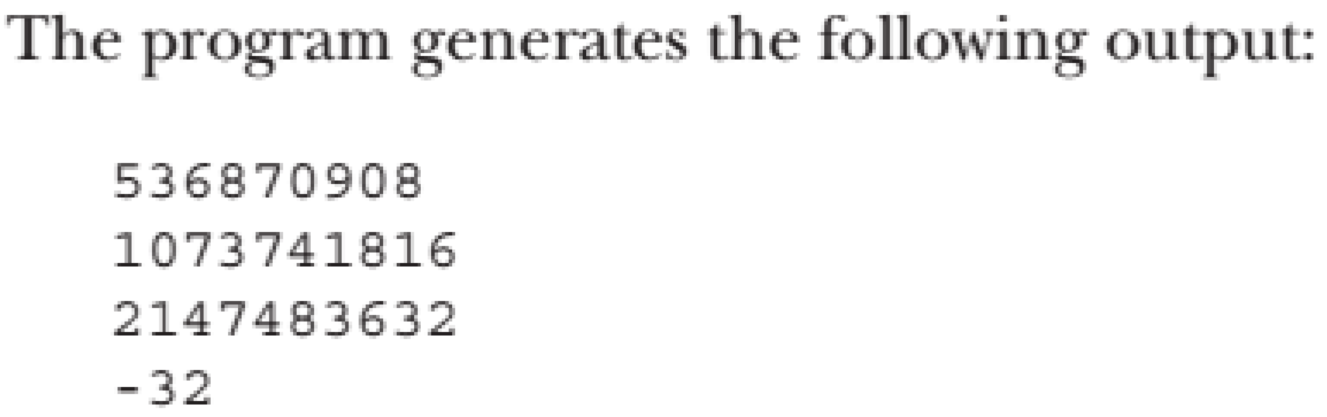
The output generated by this program is shown here:

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

Example #7



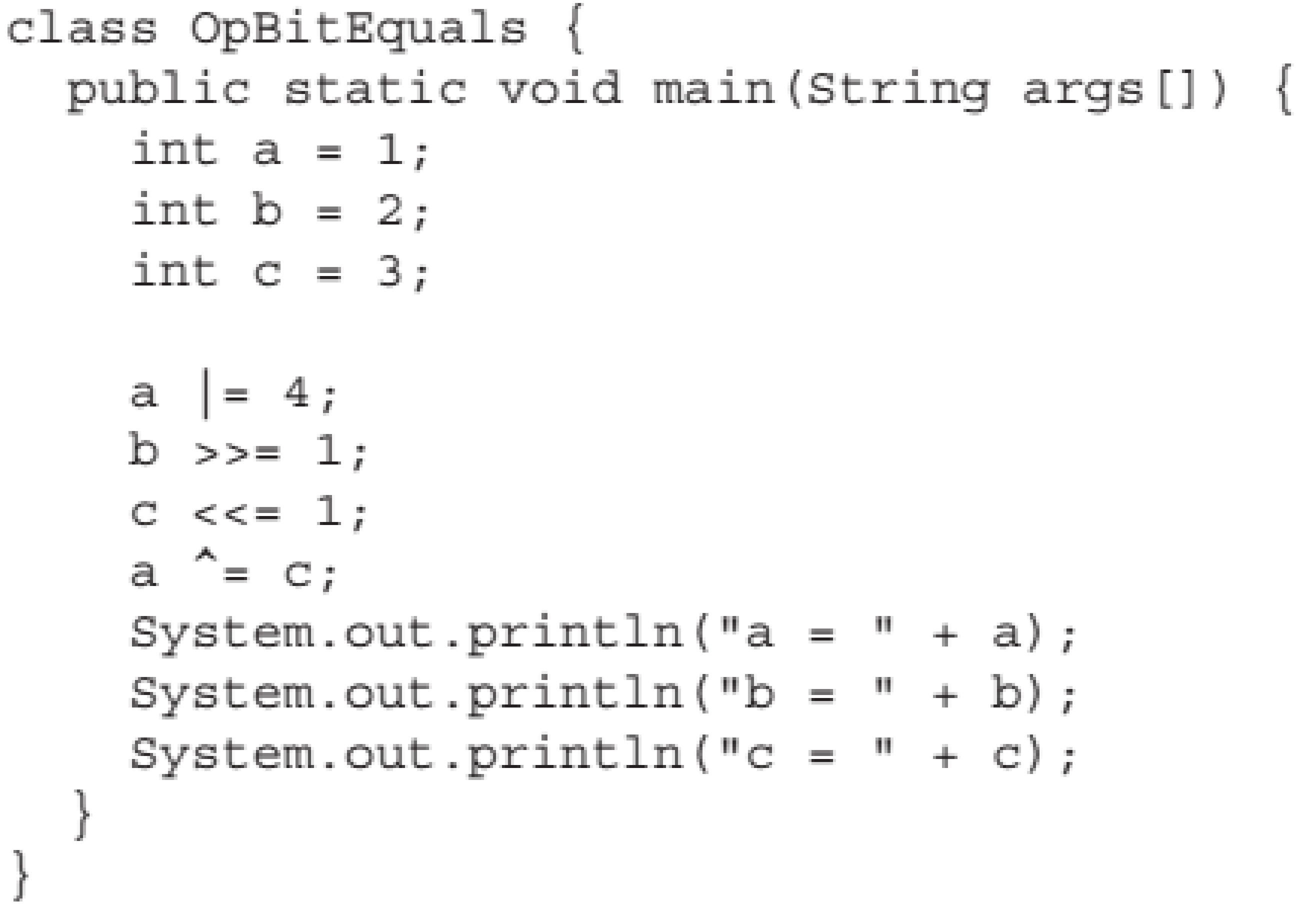
```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
  public static void main(String args[]) {
    int i;
    int num = 0xFFFFFFE;

    for(i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
}
```



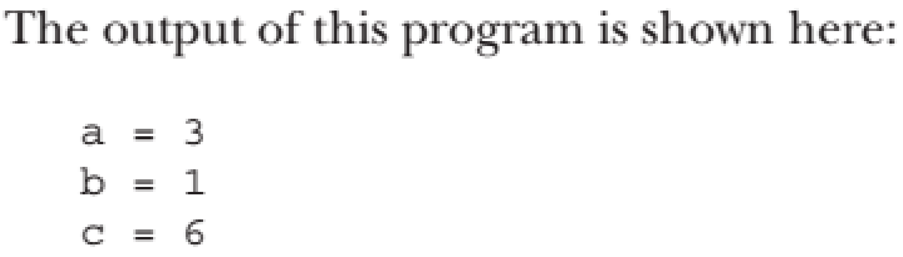
The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

Class Exercise #1



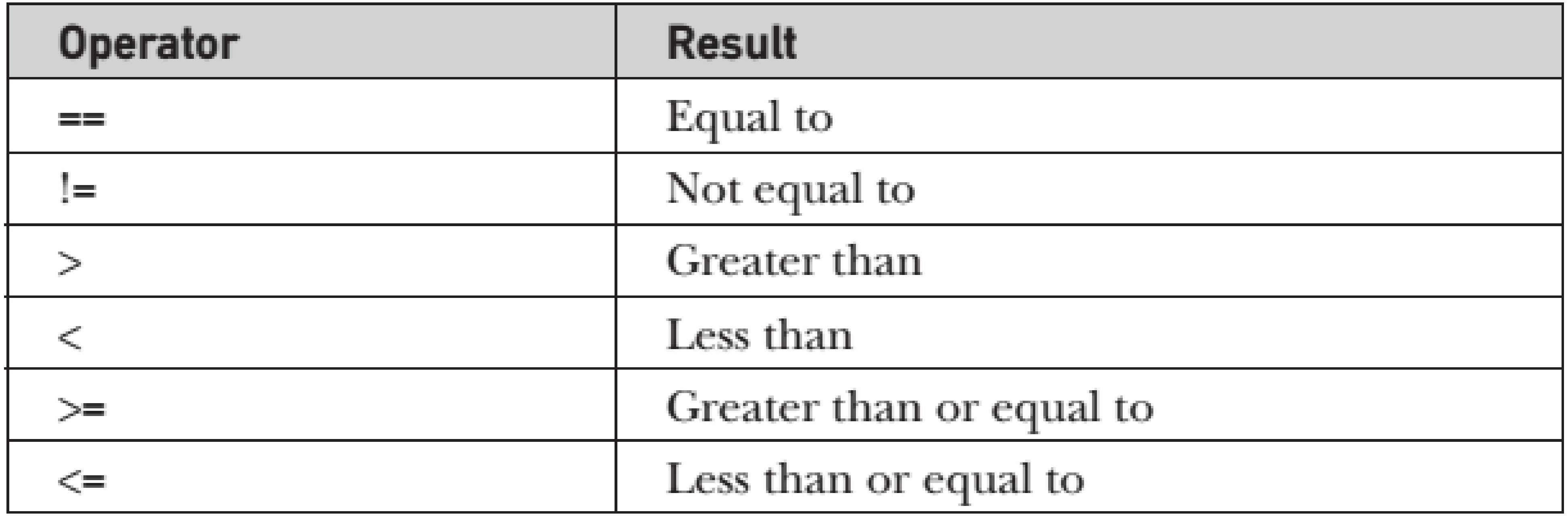
```
class OpBitEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;
    b >>= 1;
    c <<= 1;
    a ^= c;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```



The output of this program is shown here:

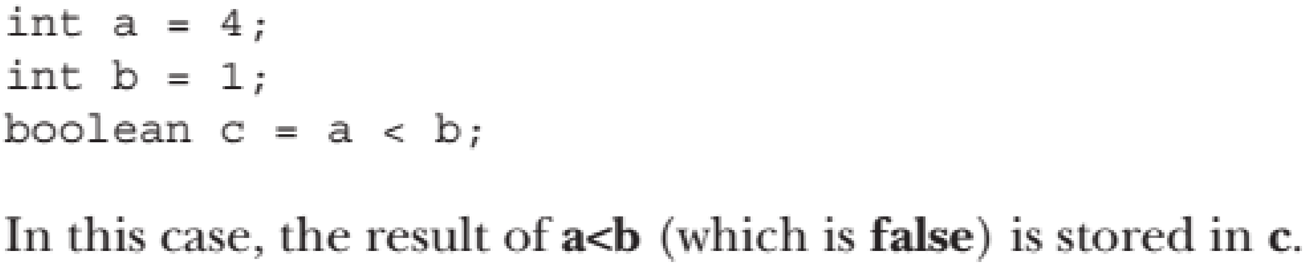
```
a = 3
b = 1
c = 6
```

Relational Operators



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

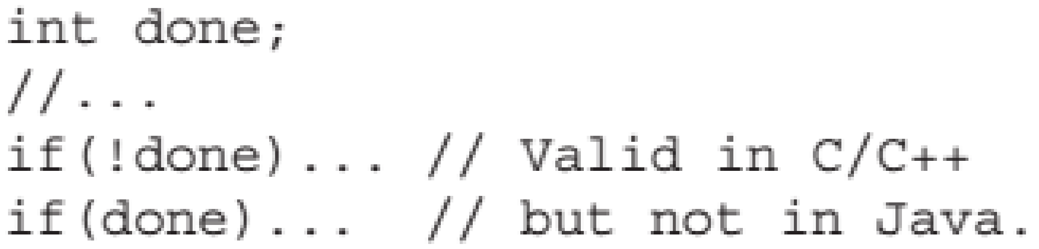
The outcome of these operations is a **boolean** value.



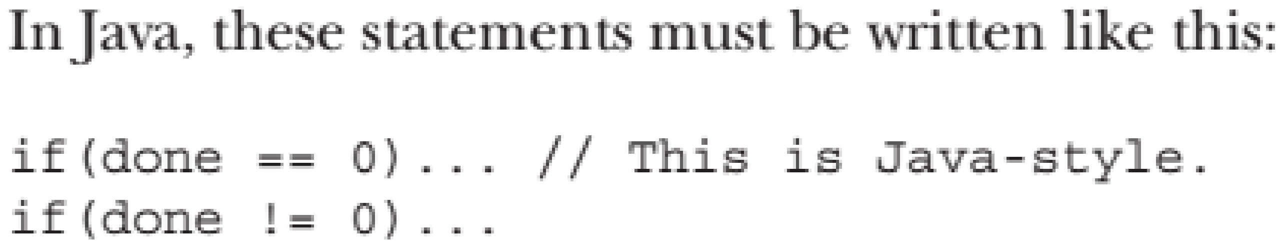
```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a<b** (which is **false**) is stored in **c**.

Relational Operators



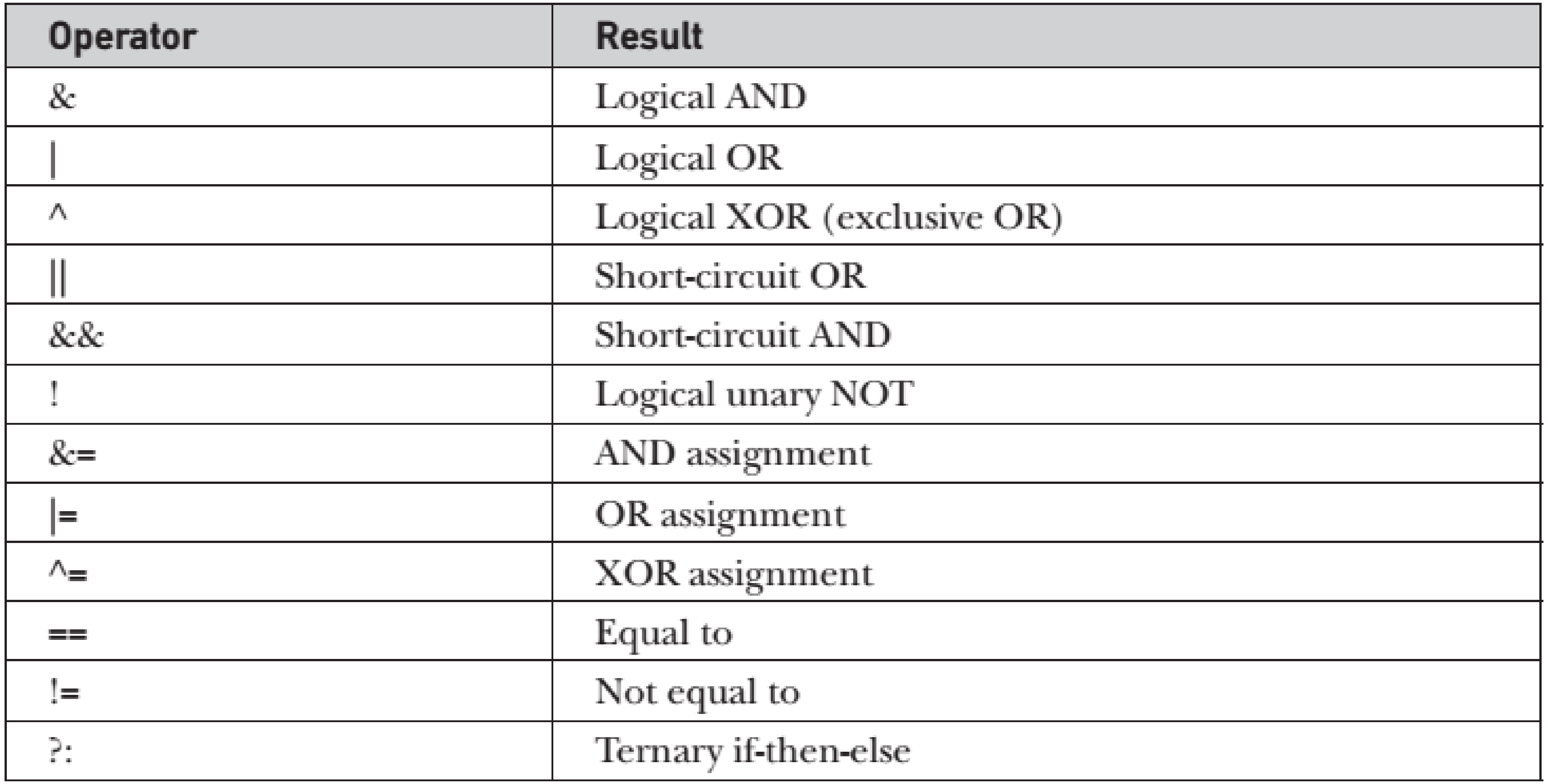
```
int done;
//...
if(!done)... // Valid in C/C++
if(done)...  // but not in Java.
```



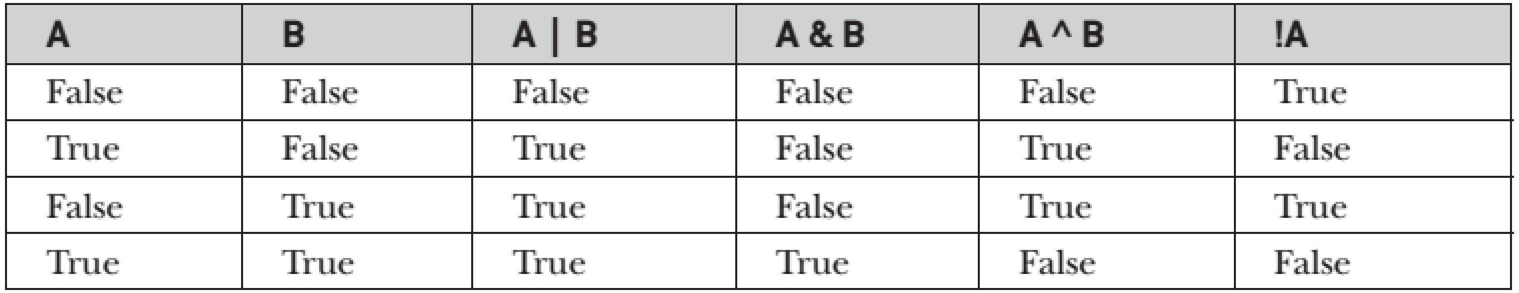
In Java, these statements must be written like this:

```
if(done == 0)... // This is Java-style.
if(done != 0)...
```

Boolean Logical Operators

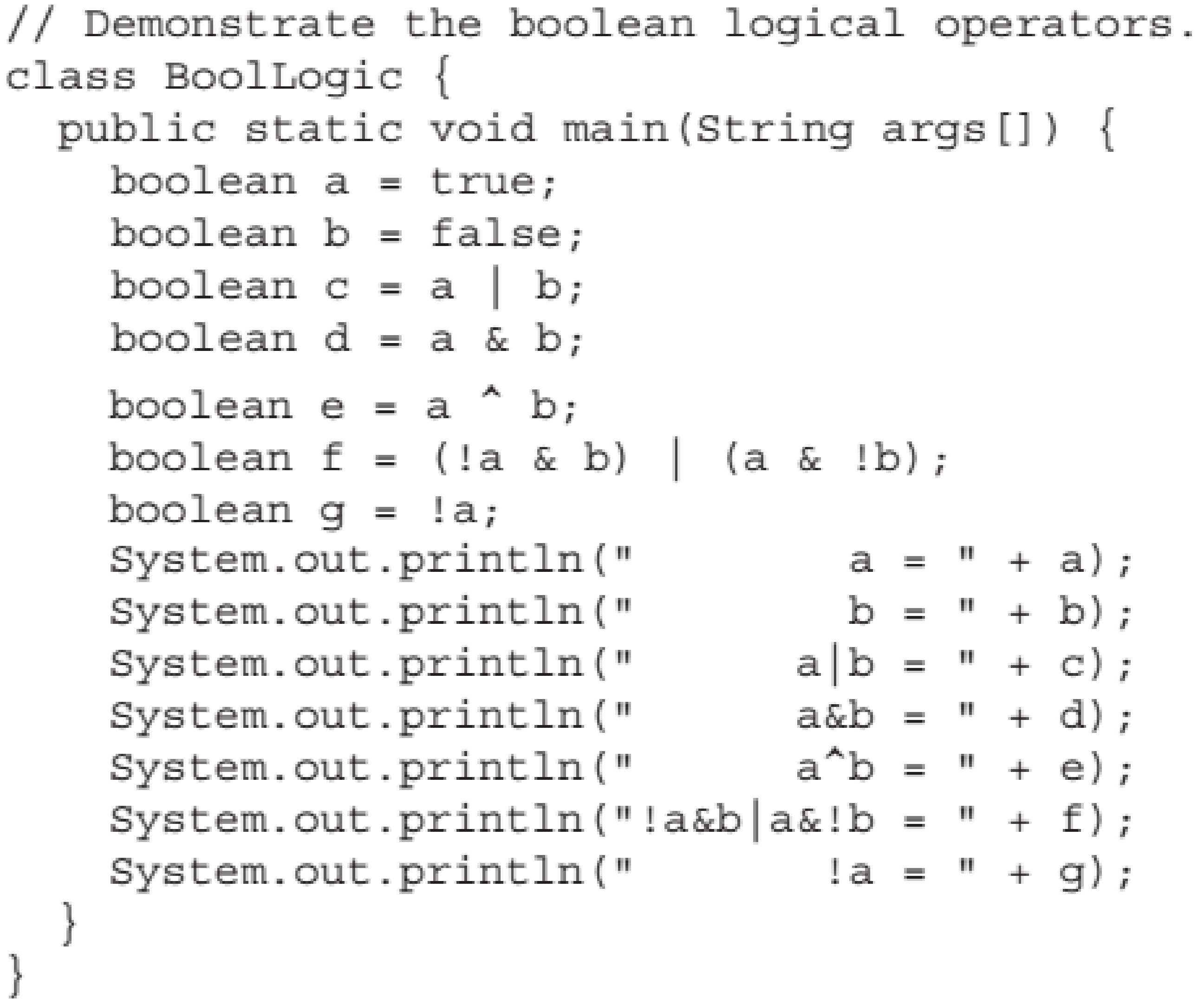


Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

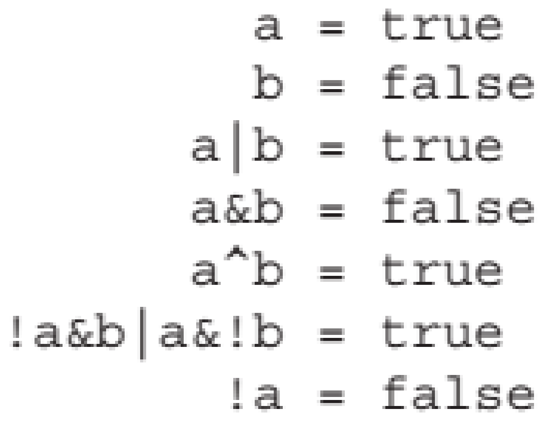


A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Example #8



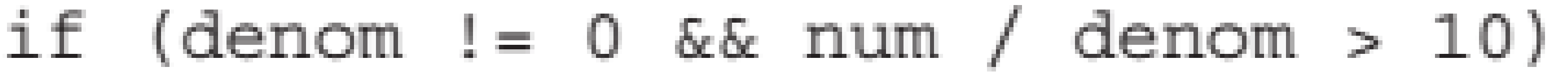
```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("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



```
        a = true
        b = false
      a|b = true
      a&b = false
      a^b = true
!a&b|a&!b = true
       !a = false
```

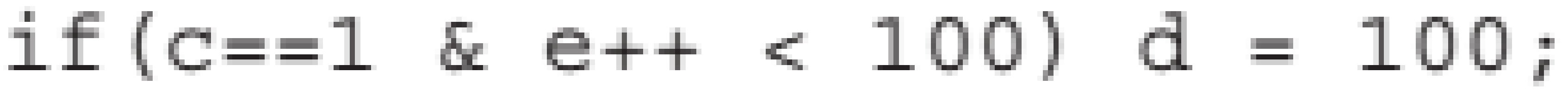
Short-Circuit Logical Operators

- 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, Java will not bother to evaluate the right hand operand when the outcome of the expression can be determined by the left operand alone.

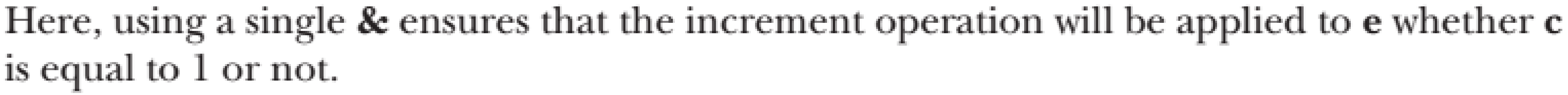


```
if (denom != 0 && num / denom > 10)
```

Class Exercise #2



```
if(c==1 & e++ < 100) d = 100;
```

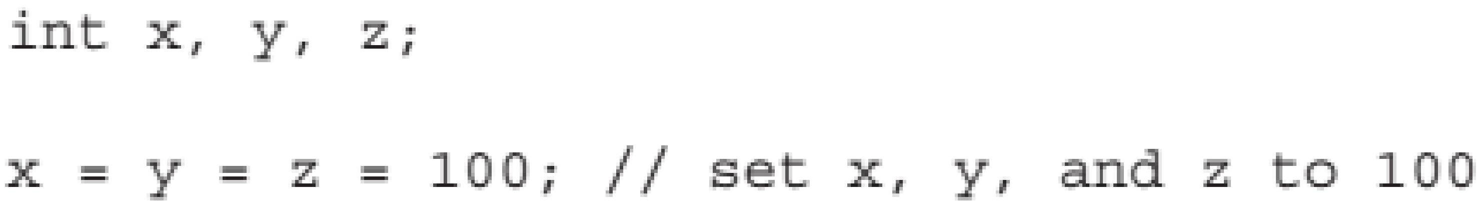


Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

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;

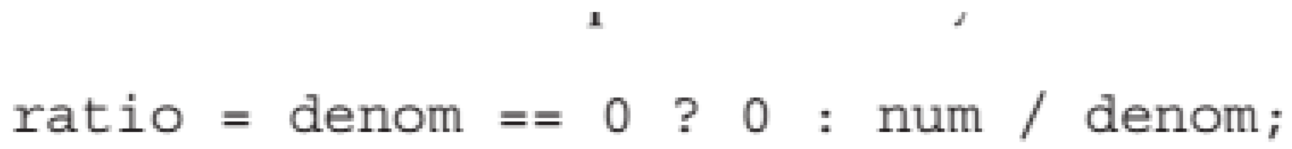
- 

```
int x, y, z;

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

The ? Operator

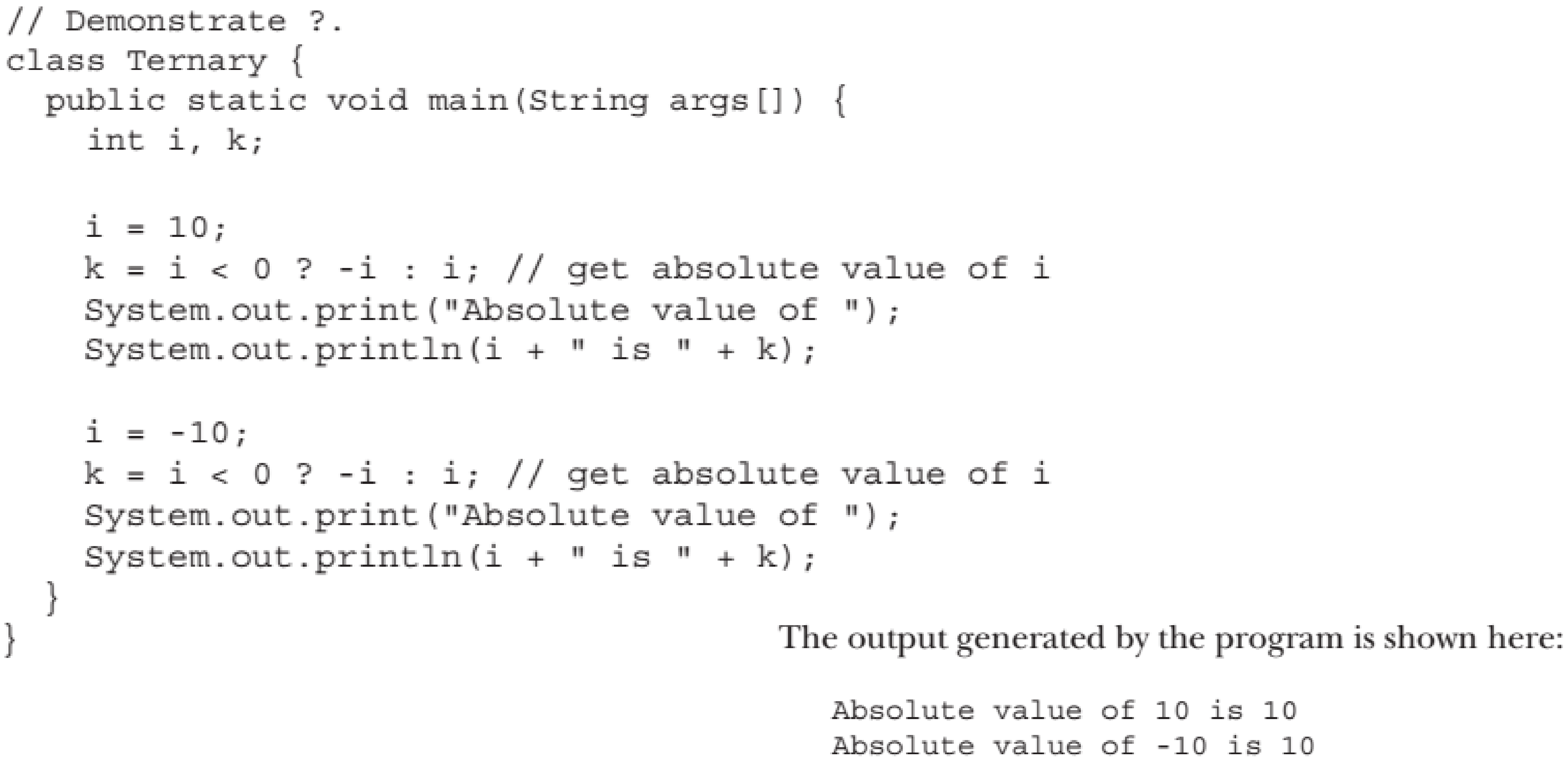
- Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements.
- The **?** has this general form:

  expression1 ? *expression2* : *expression3*

- 

```
ratio = denom == 0 ? 0 : num / denom;
```

Example #9

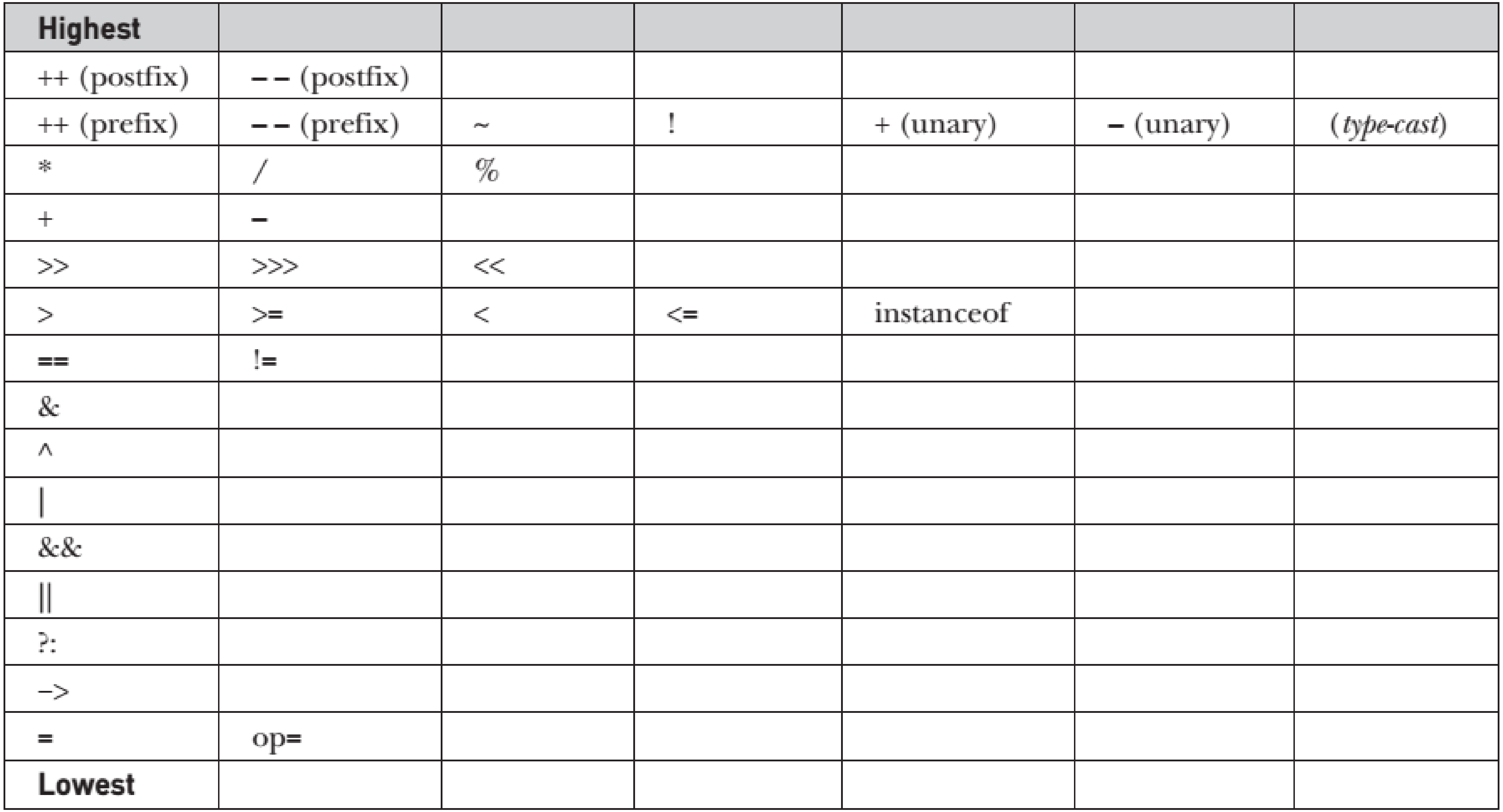


```
// Demonstrate ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;

    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
```

Operator Precedence



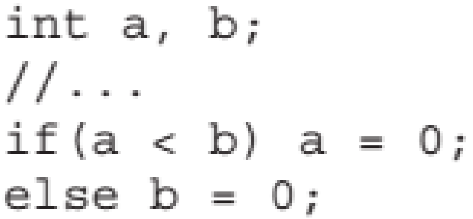
Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Control Statements

if

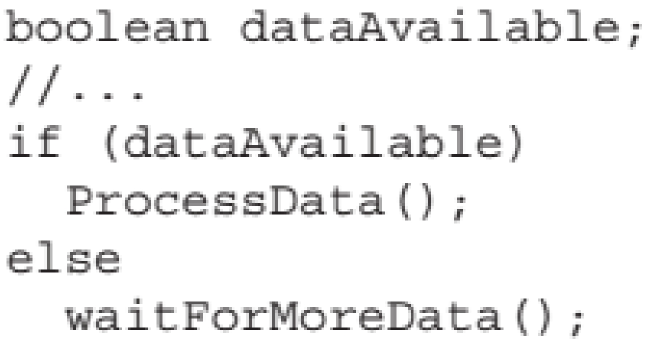
- The general form of the **if** statement:

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



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

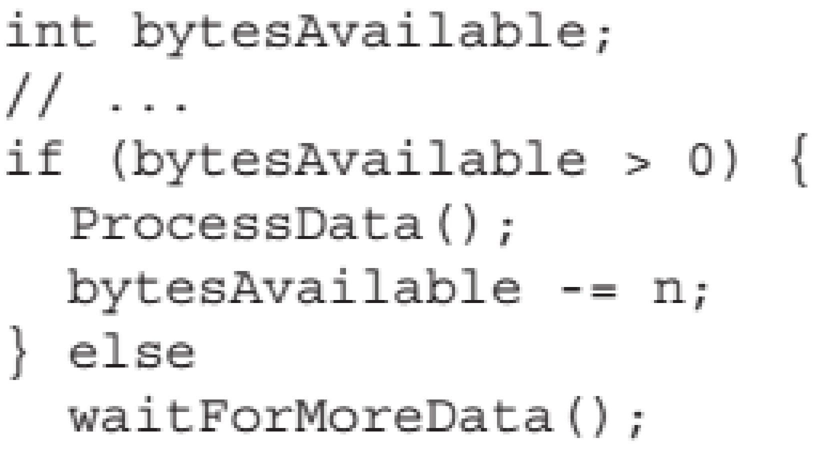
- It is possible to control the **if** using a single **Boolean** variable, as shown in this code fragment:



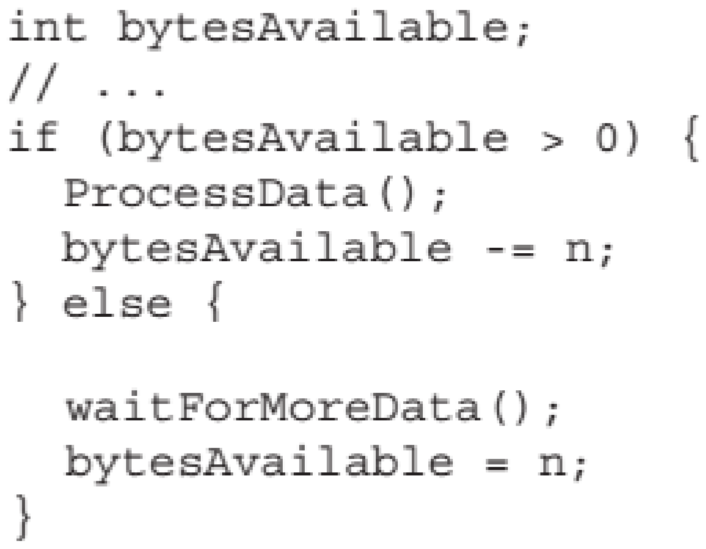
```
boolean dataAvailable;
//...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

if

- Remember, only one statement can appear directly after the **if** or the **else**.
- If you want to include more statements, you'll need to create a block, as in this fragment:



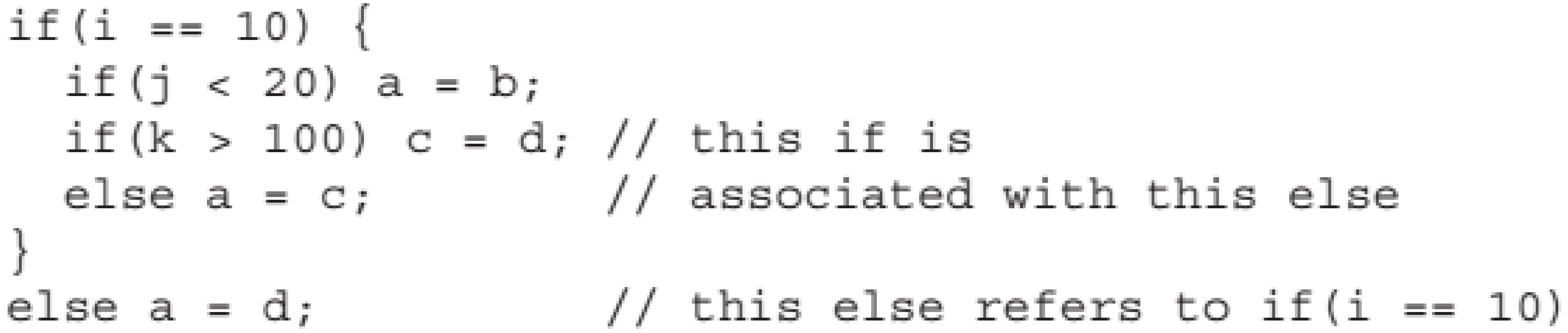
```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
```



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

  waitForMoreData();
  bytesAvailable = n;
}
```

Nested ifs

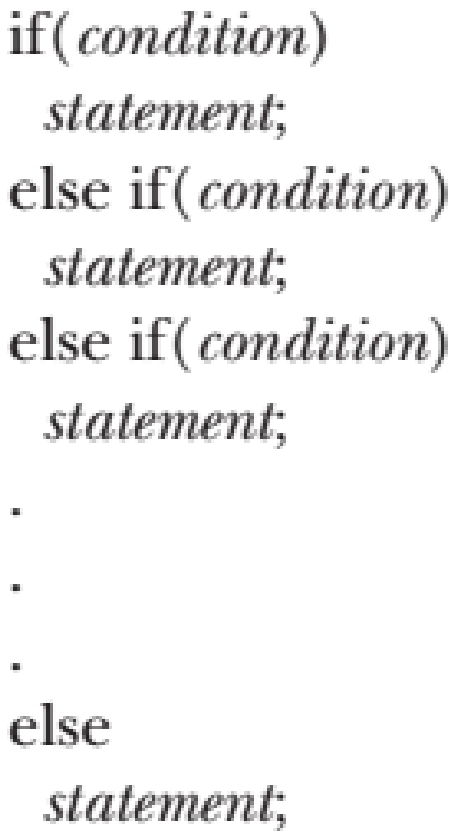
- A *nested* **if** is an **if** statement that is the target of another **if** or **else**.



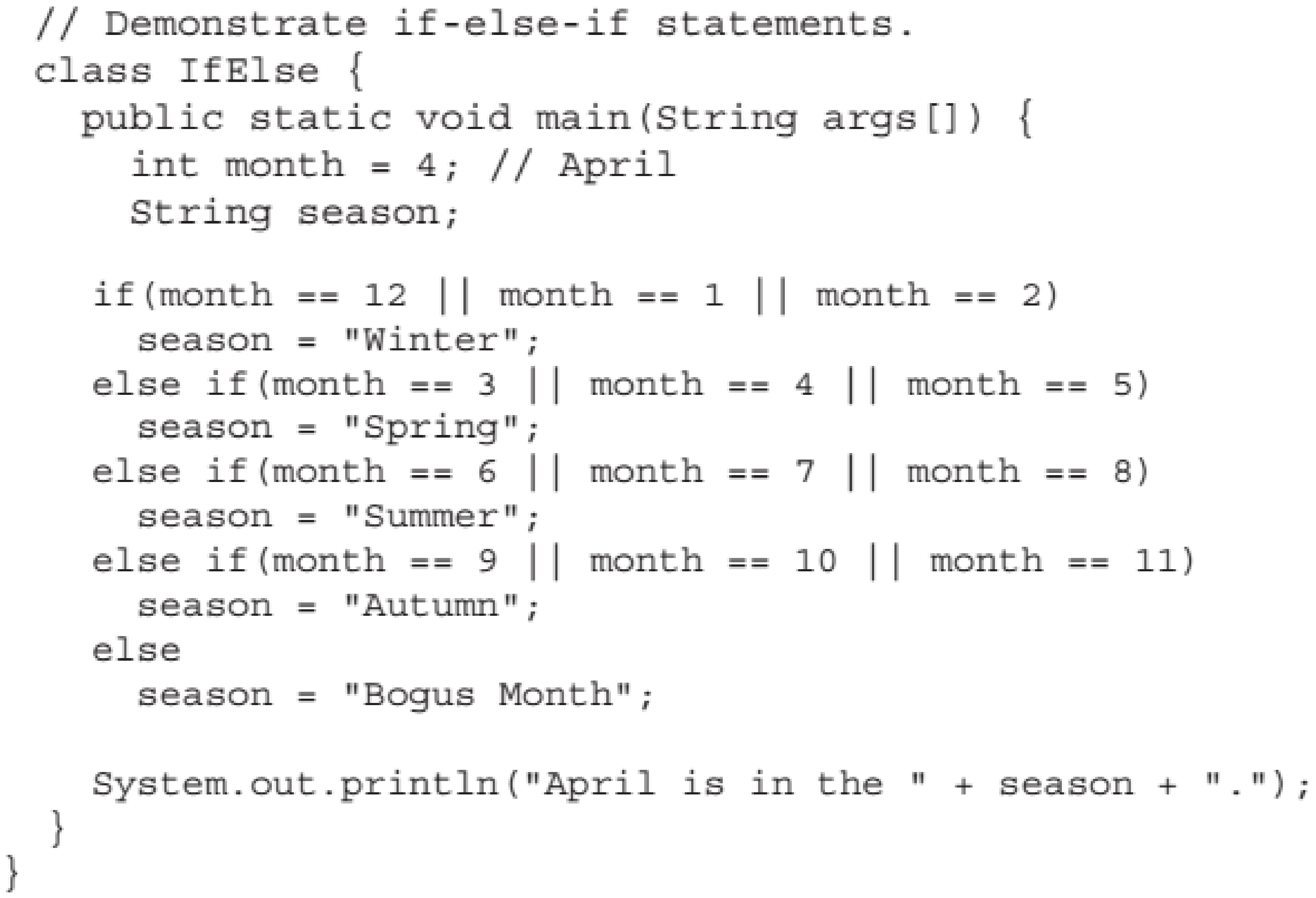
```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

The if-else-if Ladder

Example #10



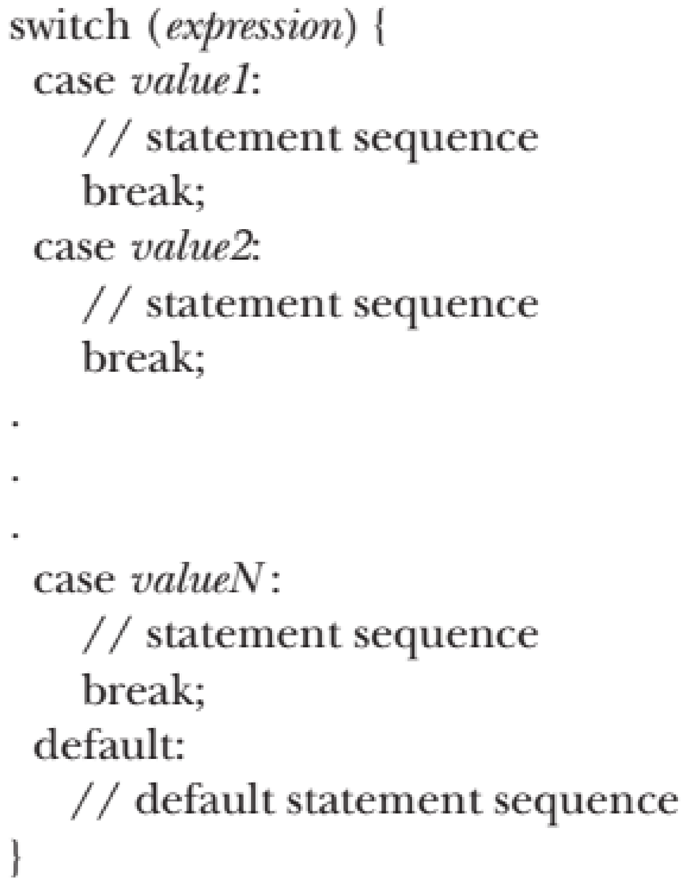
```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```



```
// Demonstrate if-else-if statements.
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 + ".");
  }
}
```

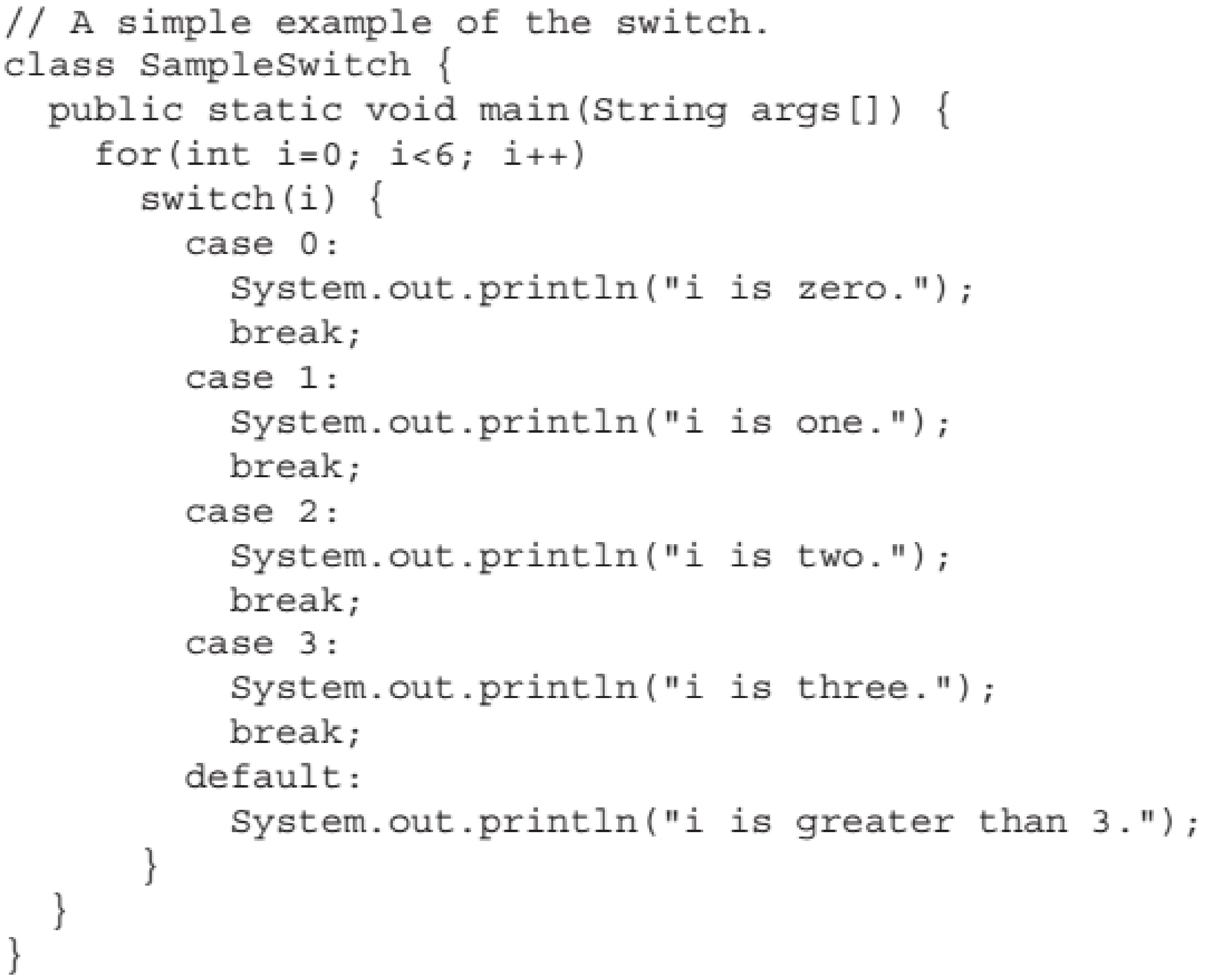
switch

- The **switch** statement is Java’s multiway branch statement.

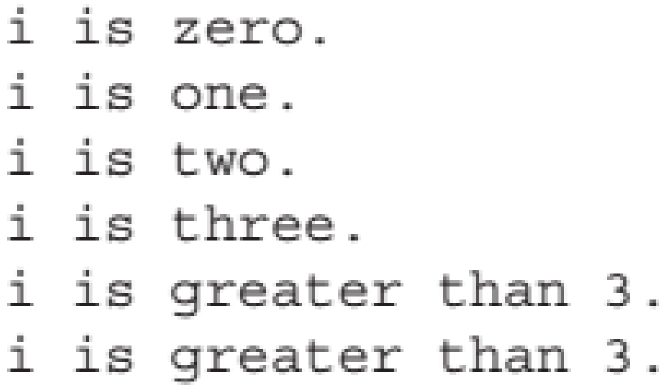


```
switch (expression) {
  case value1:
     // statement sequence
     break;
  case value2:
     // statement sequence
     break;
.
.
.
  case valueN :
     // statement sequence
     break;
  default:
    // default statement sequence
}
```

Example #11

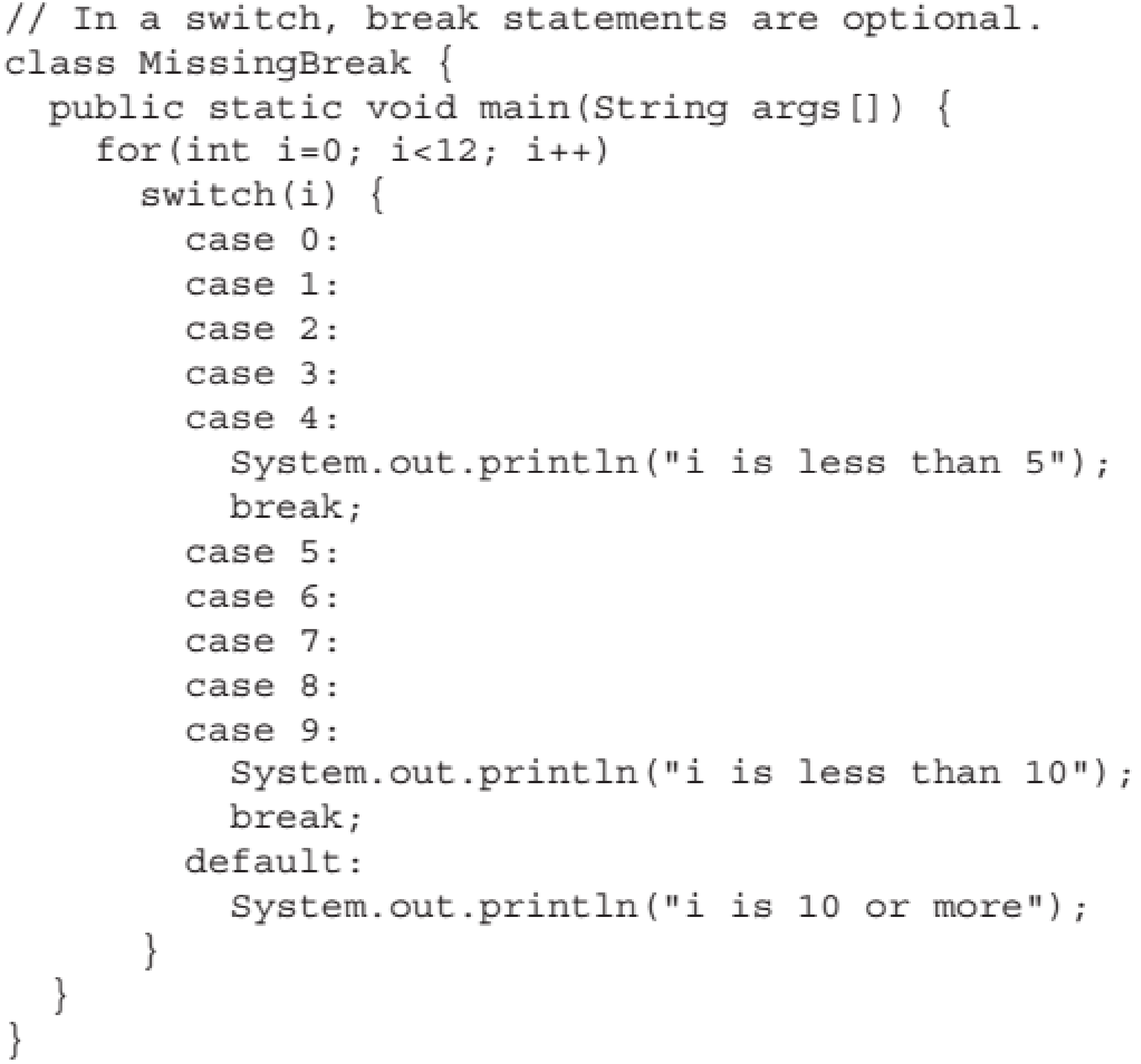


```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        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.");
      }
  }
}
```

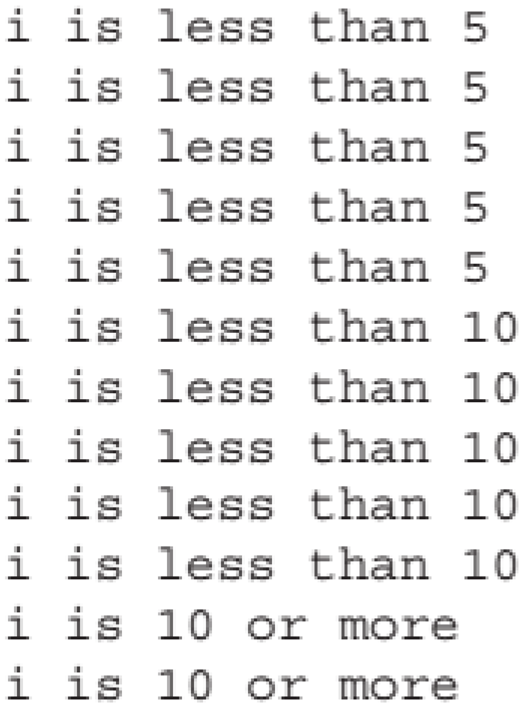


```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Class Exercise #3

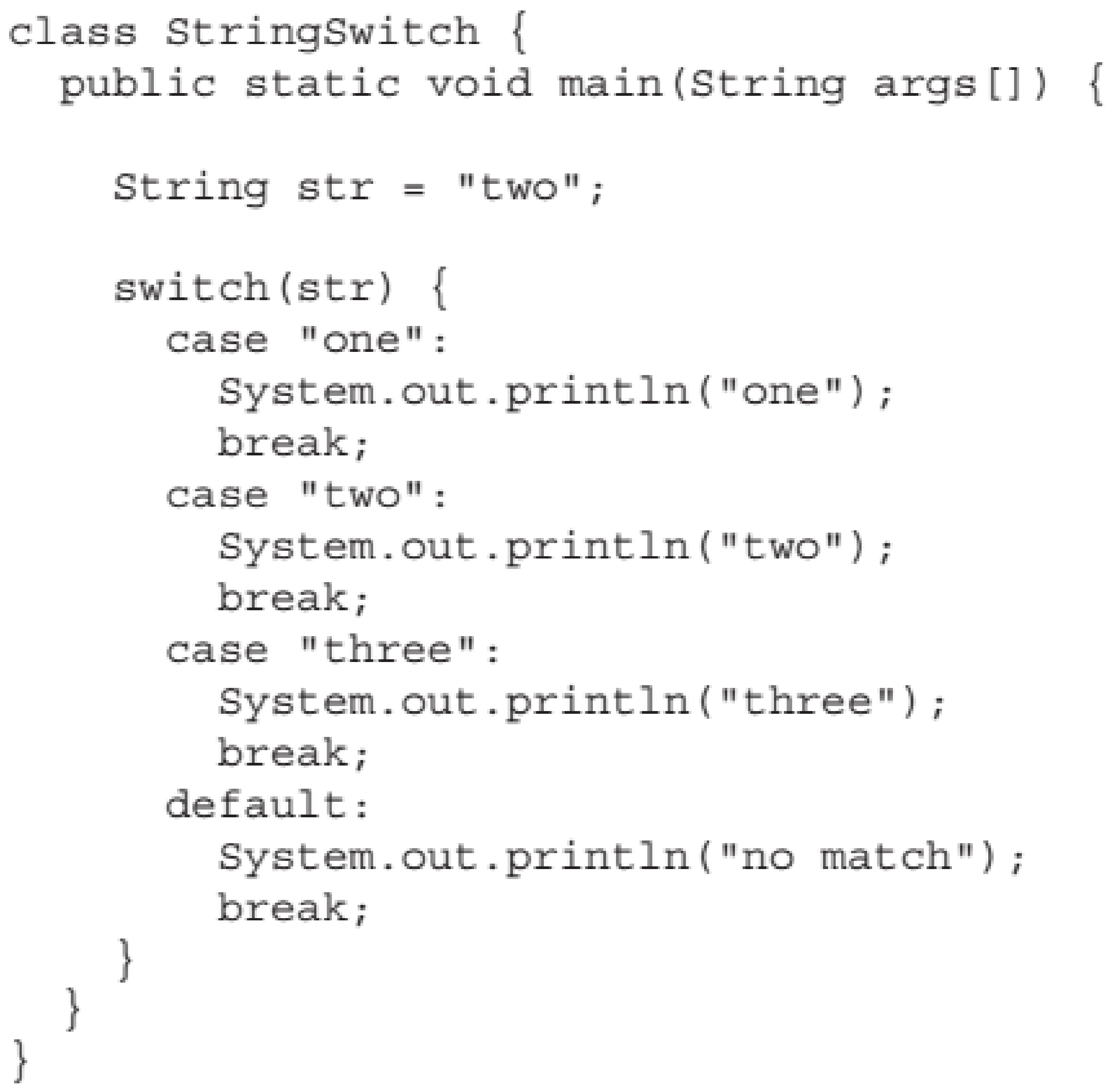


```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
         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

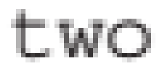


```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

Class Exercise #4

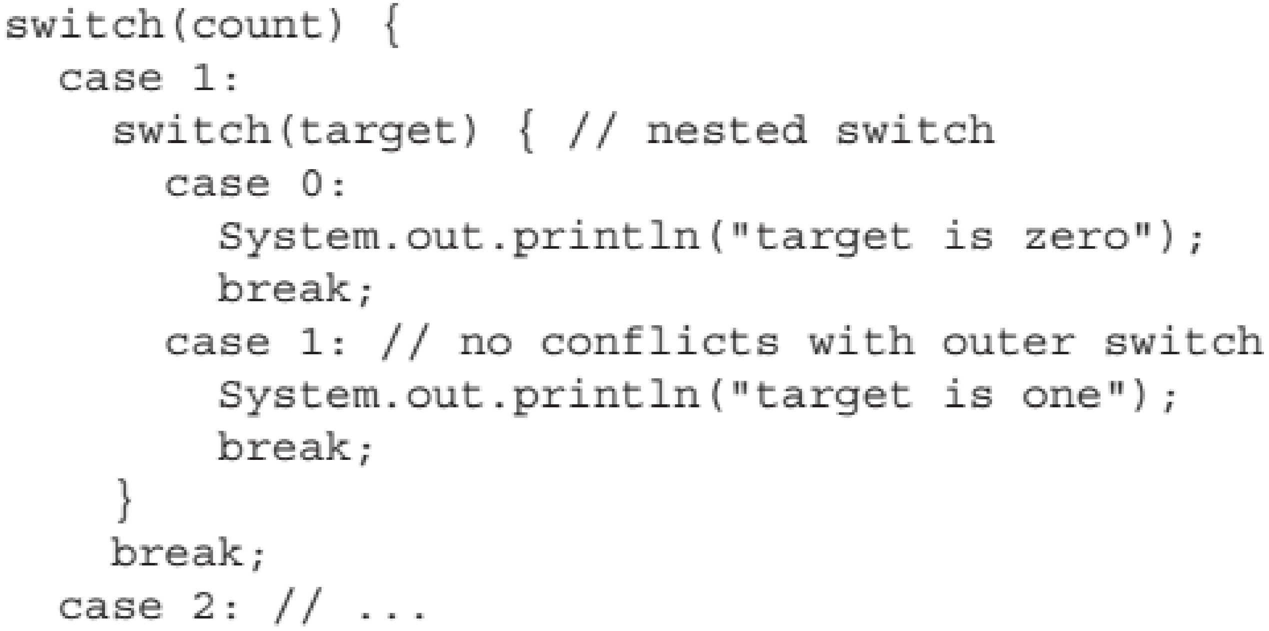


```
class StringSwitch {
  public static void main(String args[]) {

    String str = "two";

    switch(str) {
      case "one":
        System.out.println("one");
        break;
      case "two":
        System.out.println("two");
        break;
      case "three":
        System.out.println("three");
        break;
      default:
        System.out.println("no match");
        break;
    }
  }
}
```



```
two
```

Nested switch Statements



```
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: // ...
```

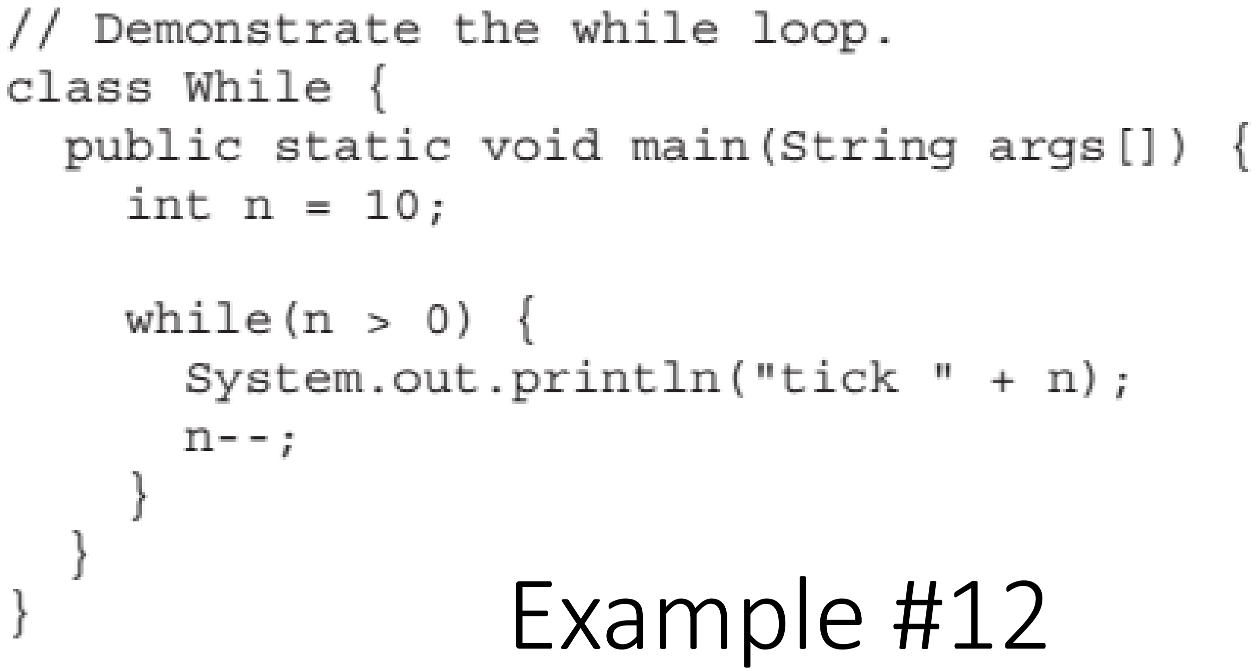
- You can use a **switch** as part of the statement sequence of an outer **switch**.
- This is called a *nested* **switch**.
- Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.

Iteration Statements

while

- The **while** loop is Java’s most fundamental loop statement.
- It repeats a statement or block while its controlling expression is true.
- General form:
  while(*condition*)

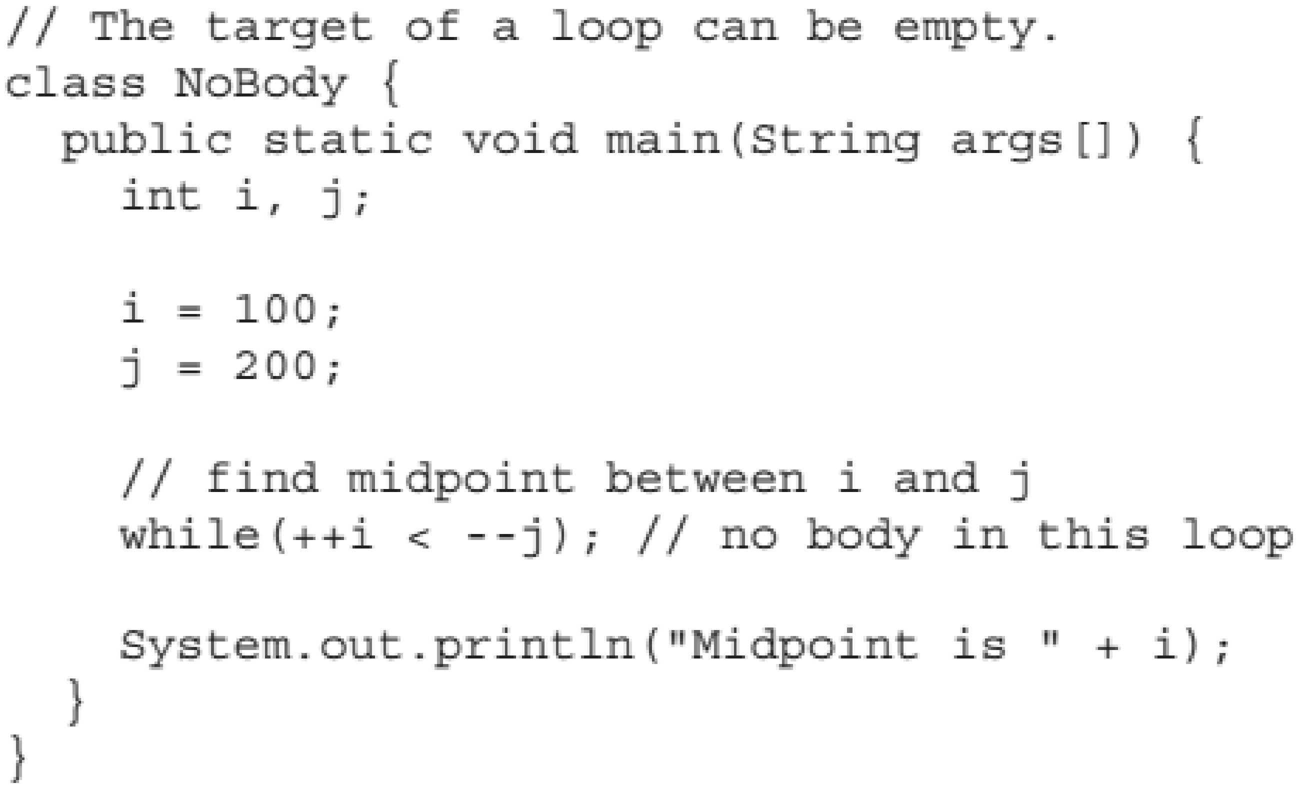
  {
          // body of loop
  }



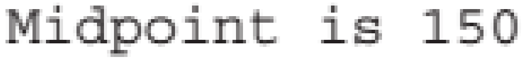
```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

Example #12

Class Exercise #5



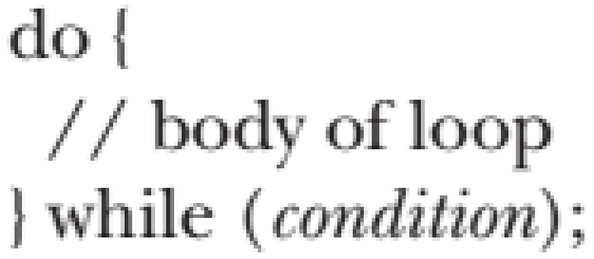
```
// The target of a loop can be empty.
class NoBody {
  public static void main(String args[]) {
    int i, j;

    i = 100;
    j = 200;

    // find midpoint between i and j
    while(++i < --j); // no body in this loop

    System.out.println("Midpoint is " + i);
  }
}
```



```
Midpoint is 150
```

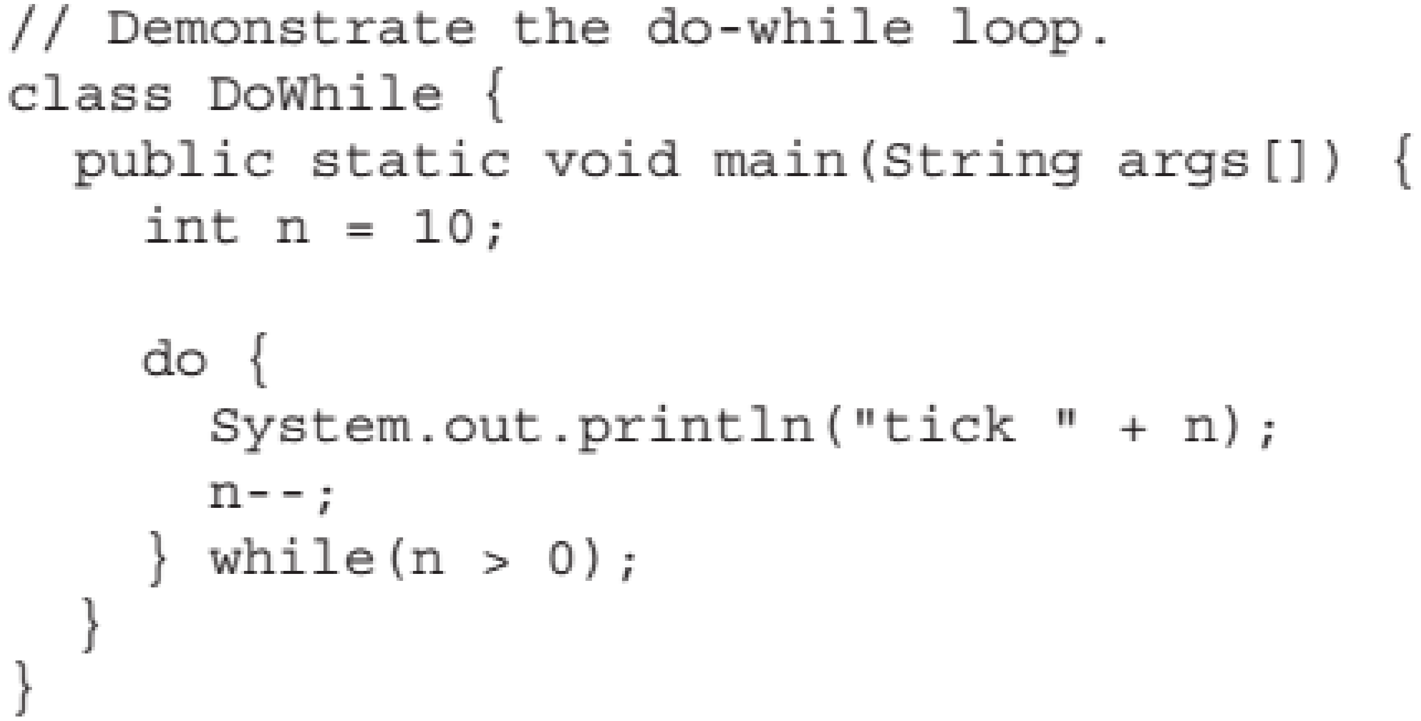
do-while

- if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all.
- The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.
- General form:

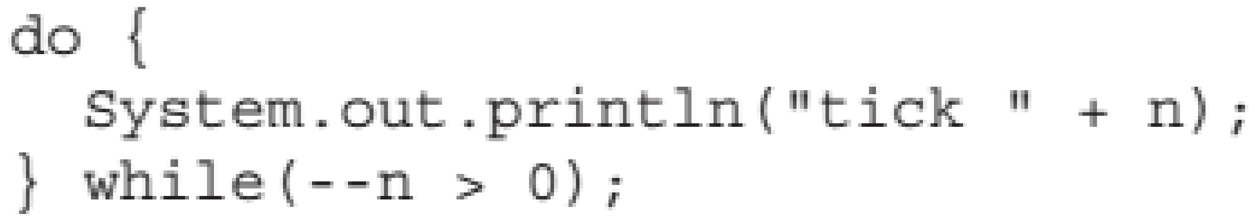


```
do {
  // body of loop
} while (condition);
```

Example #13



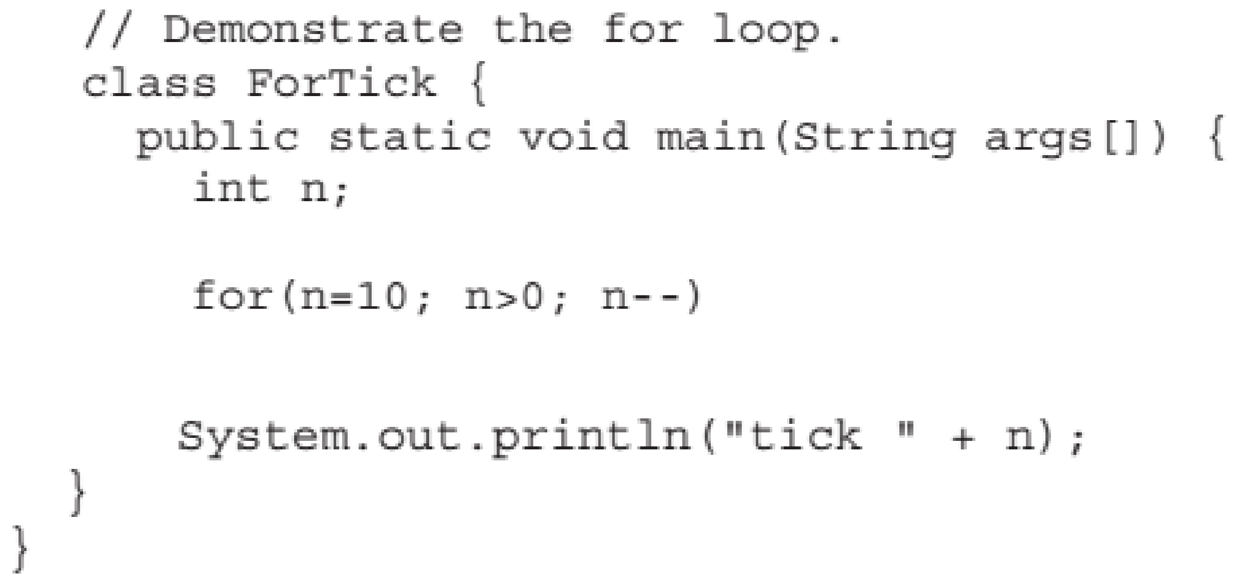
```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```



```
do {
  System.out.println("tick " + n);
} while(--n > 0);
```

for

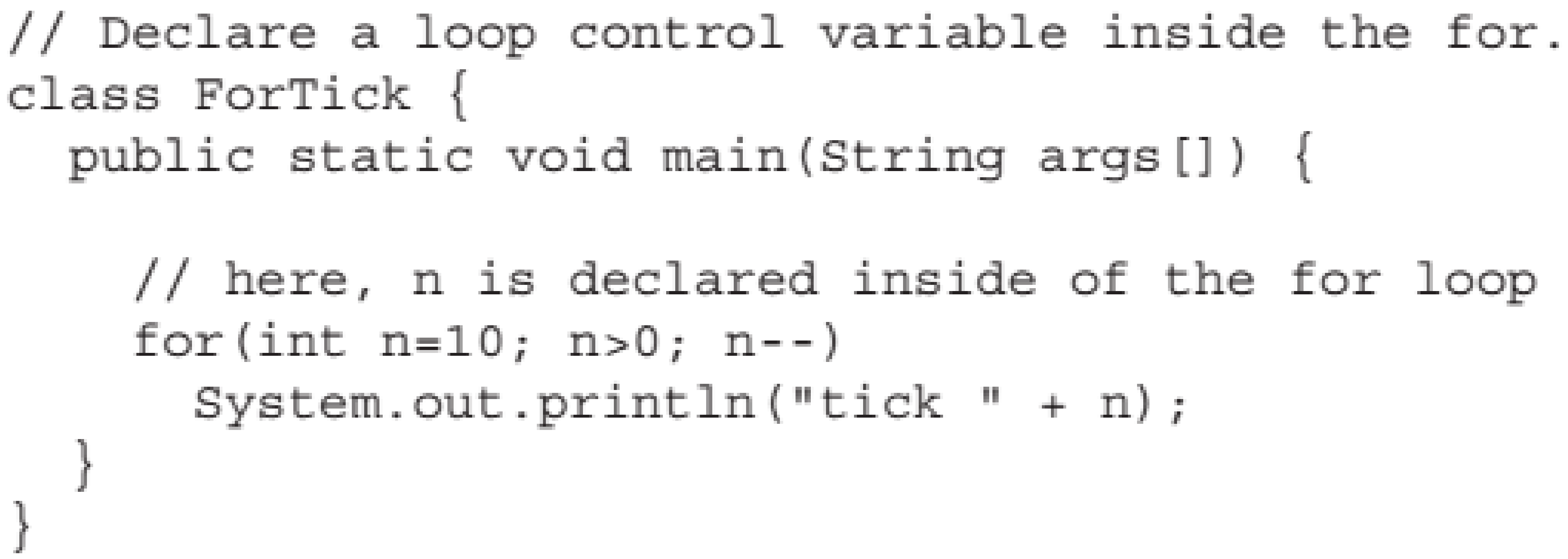
- General form of the traditional **for** statement:
  for(*initialization*; *condition*; *iteration*) {
  // body
  }



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

Example #14

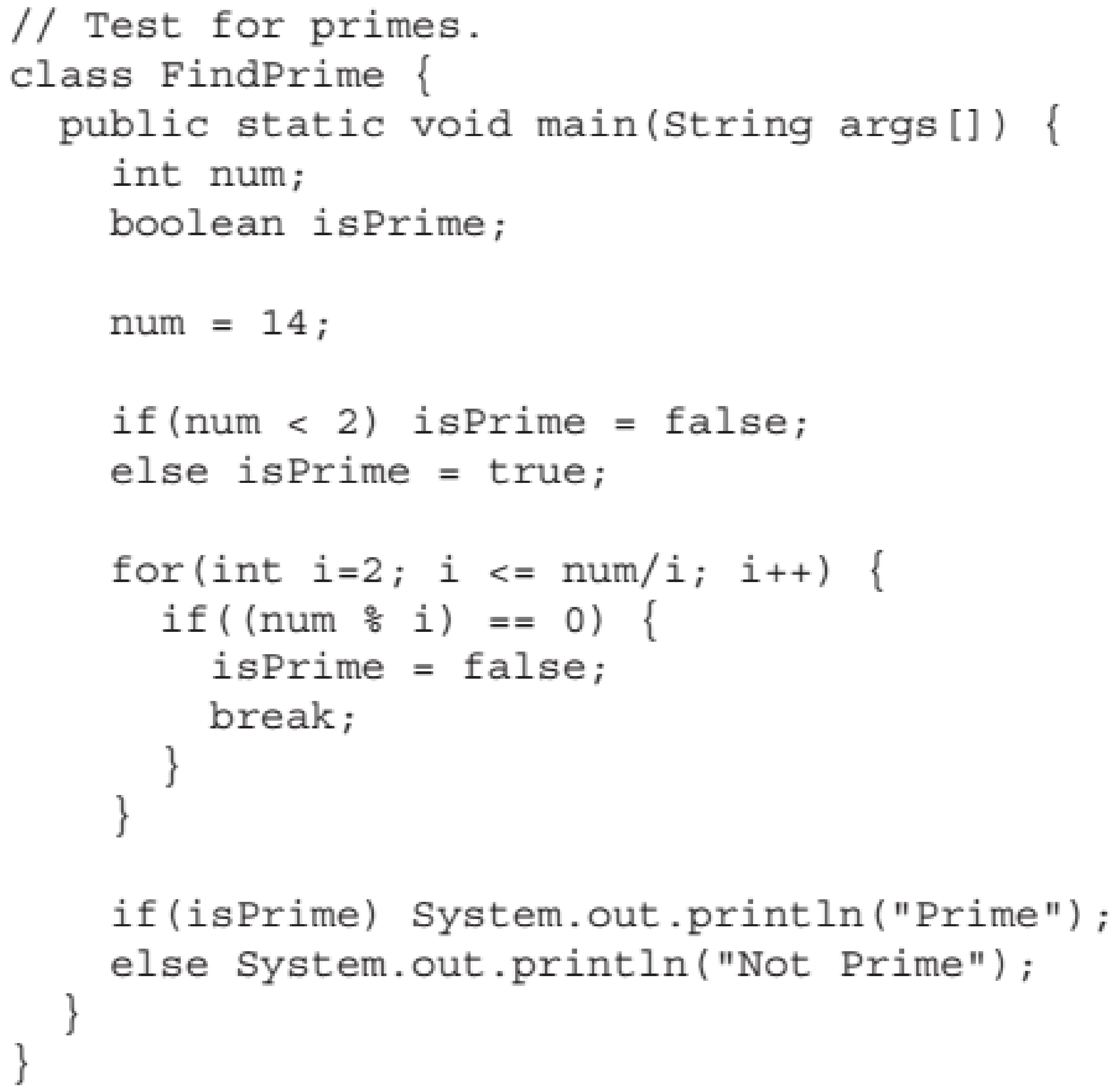
Example #15



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

Class Exercise #6

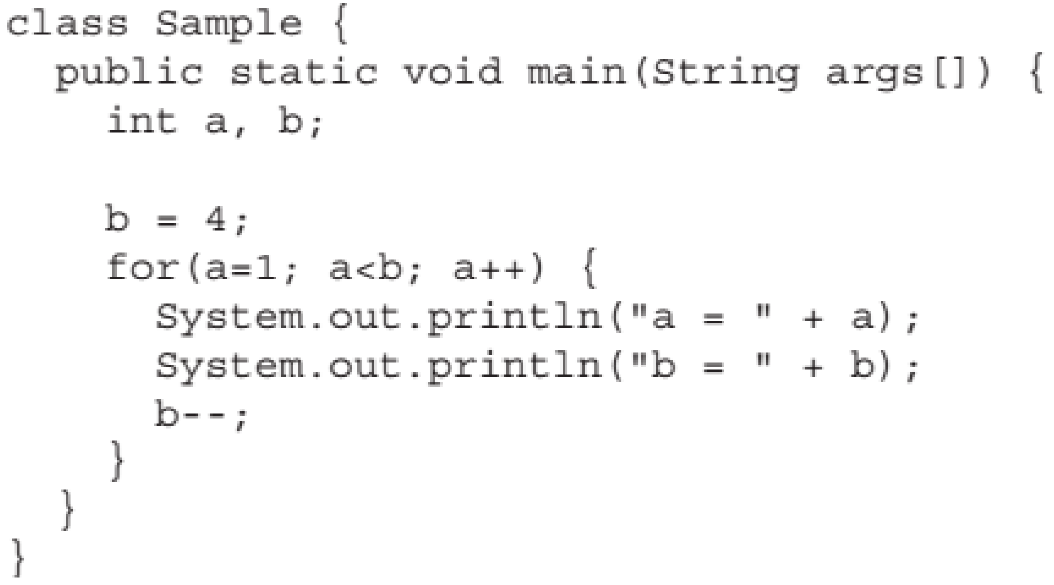
- Write a java program to find the given number is prime or not!



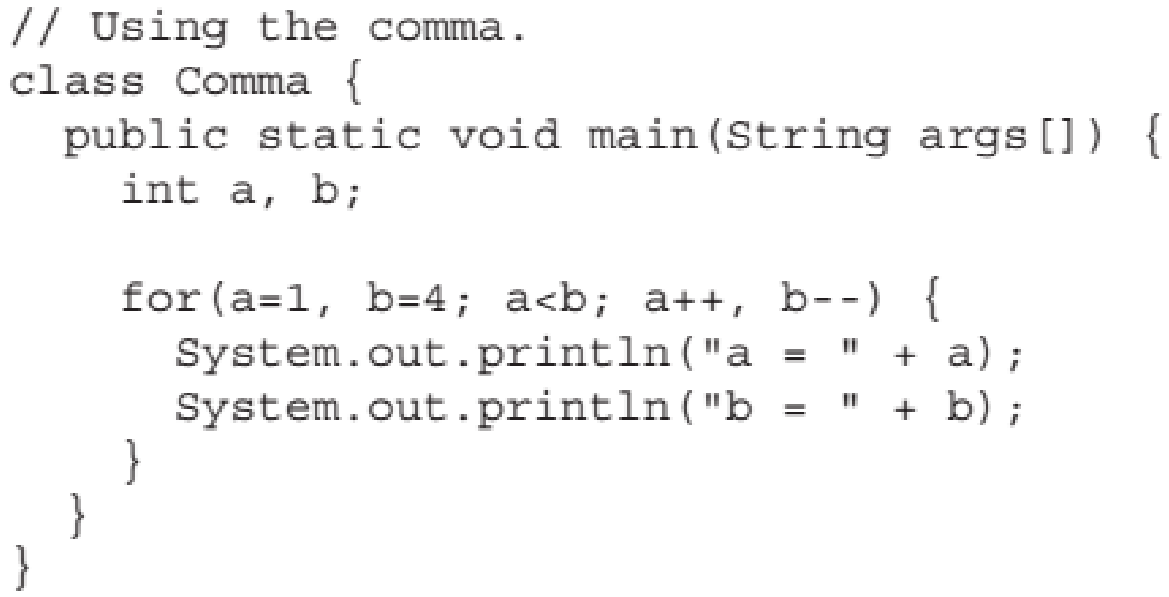
```
// Test for primes.
class FindPrime {
  public static void main(String args[]) {
    int num;
    boolean isPrime;

    num = 14;

    if(num < 2) isPrime = false;
    else isPrime = true;

    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

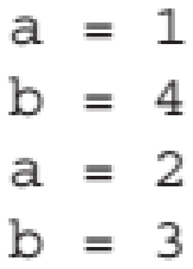
Example #16



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

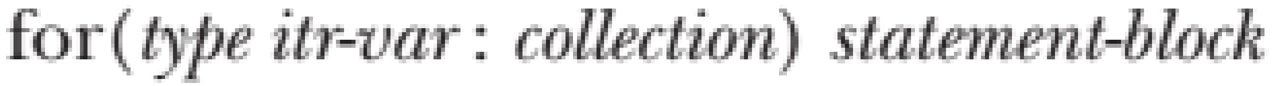


```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

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

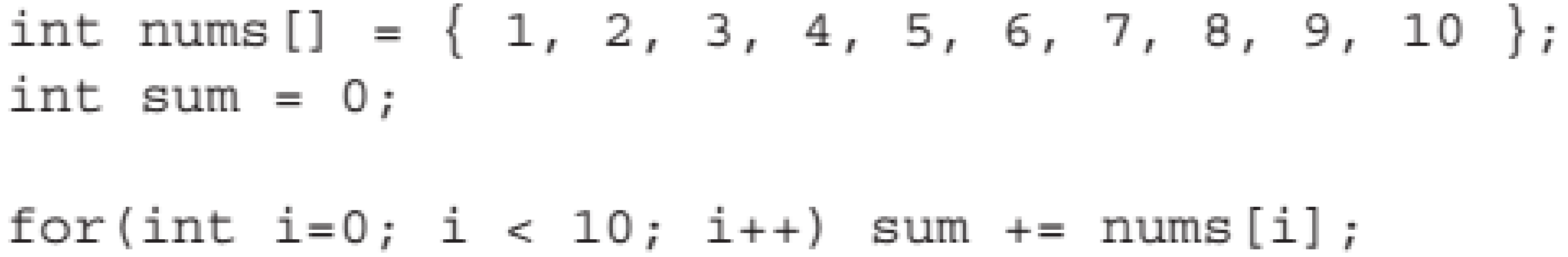


```
a = 1
b = 4
a = 2
b = 3
```

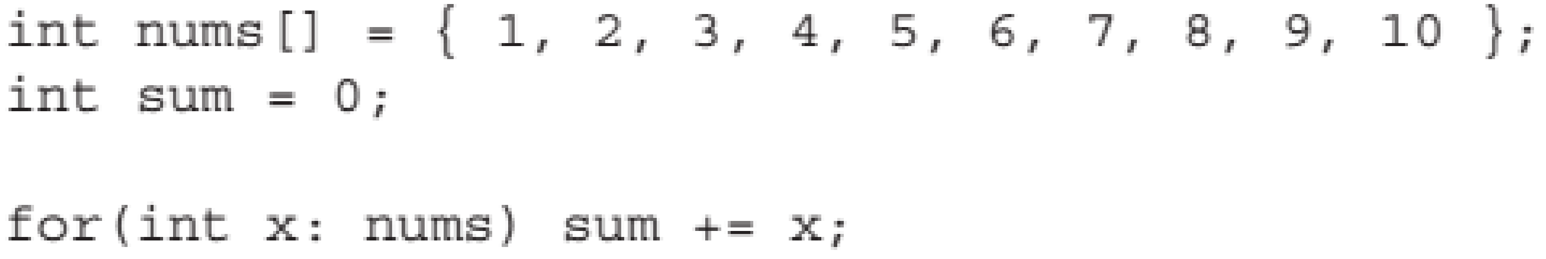
The For-Each Version of the for Loop



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

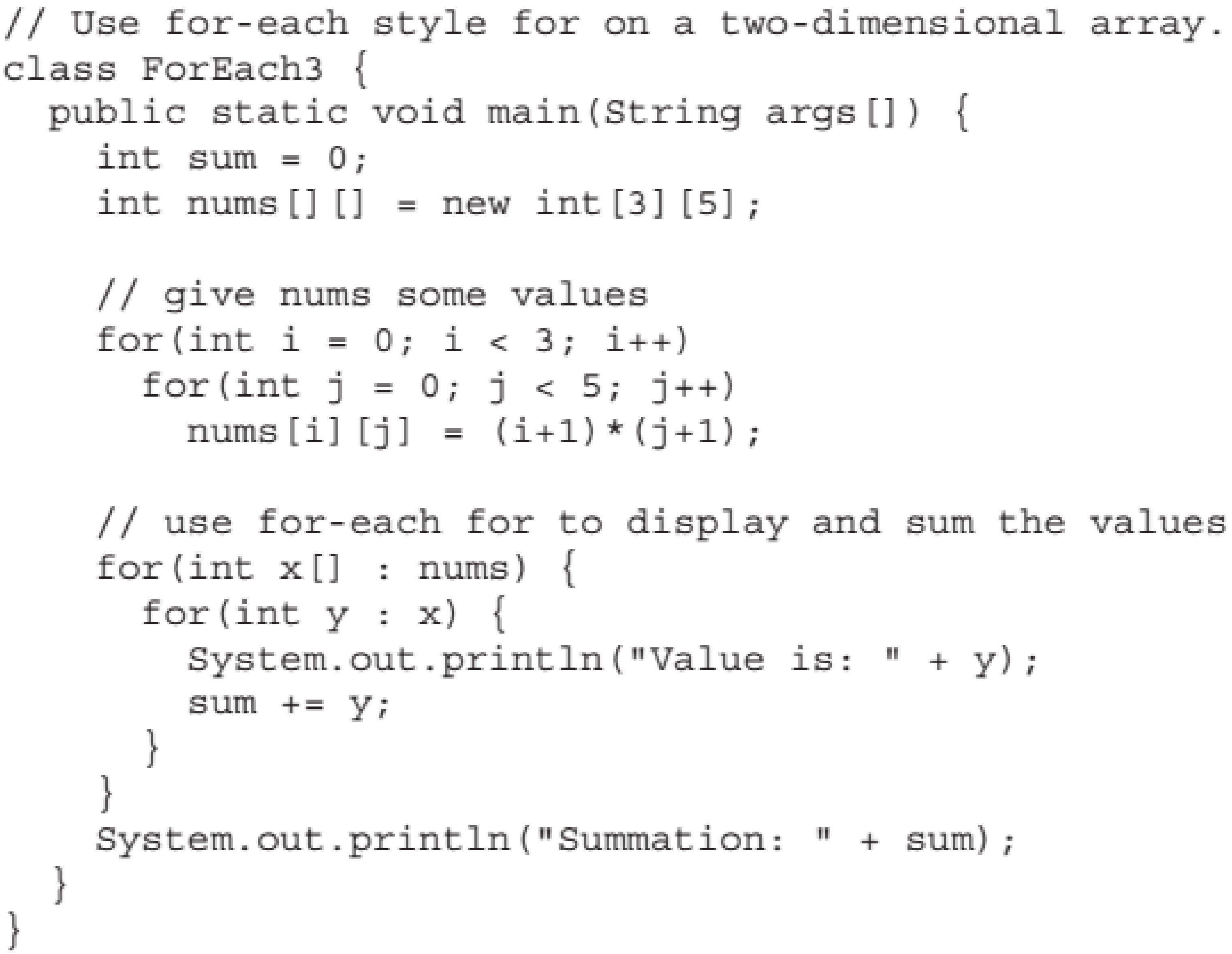


```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

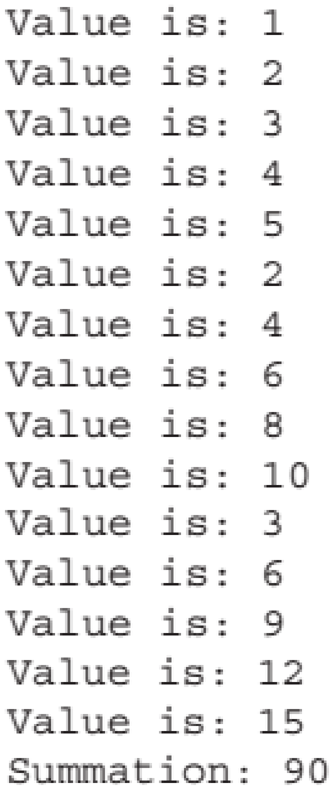


```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

Exercise #7



```
// 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);

    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

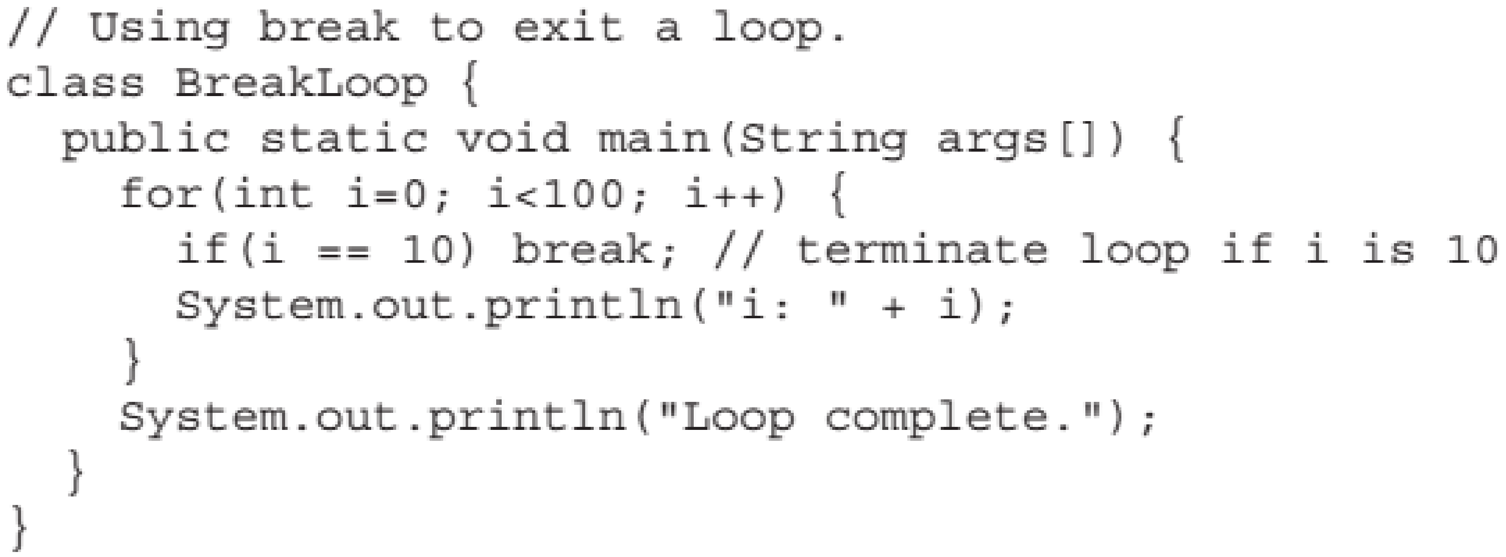


```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

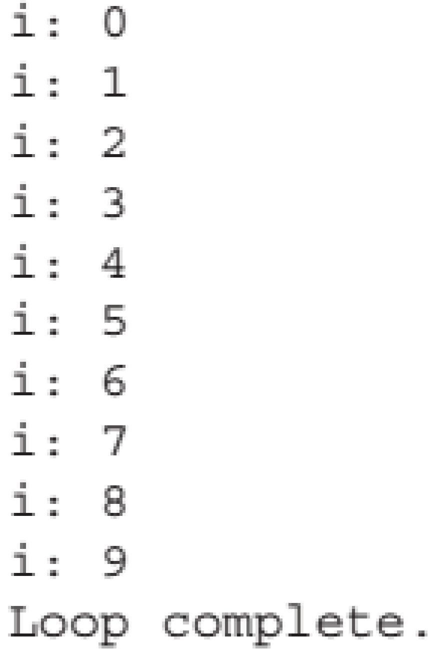
Jump Statements

- Java supports three jump statements: **break**, **continue**, and **return**.
- 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.

Example #17

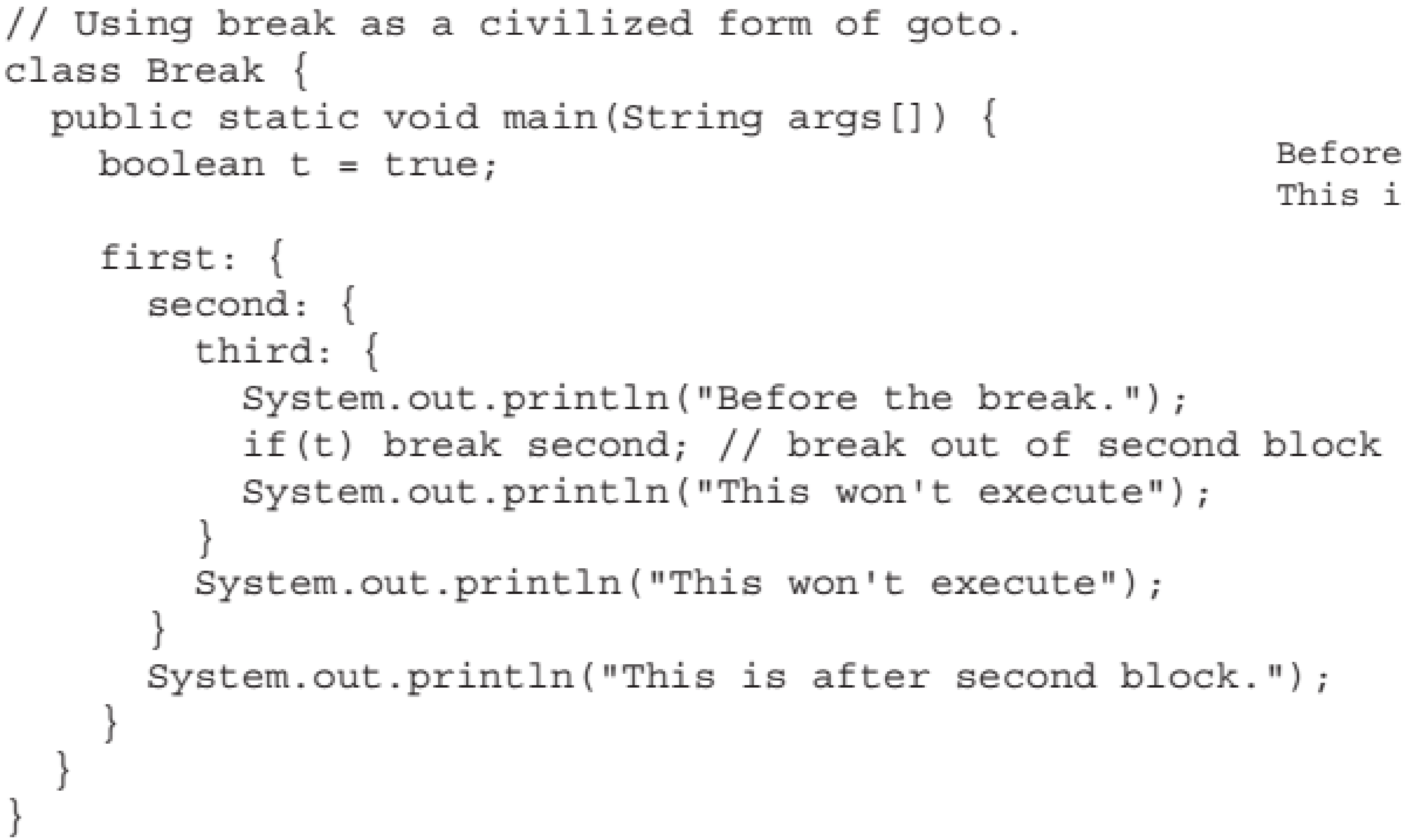


```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

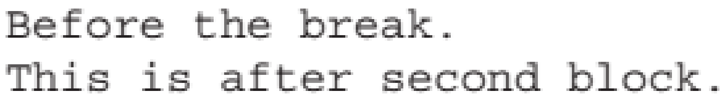


```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Example #18 : Civilized form of goto



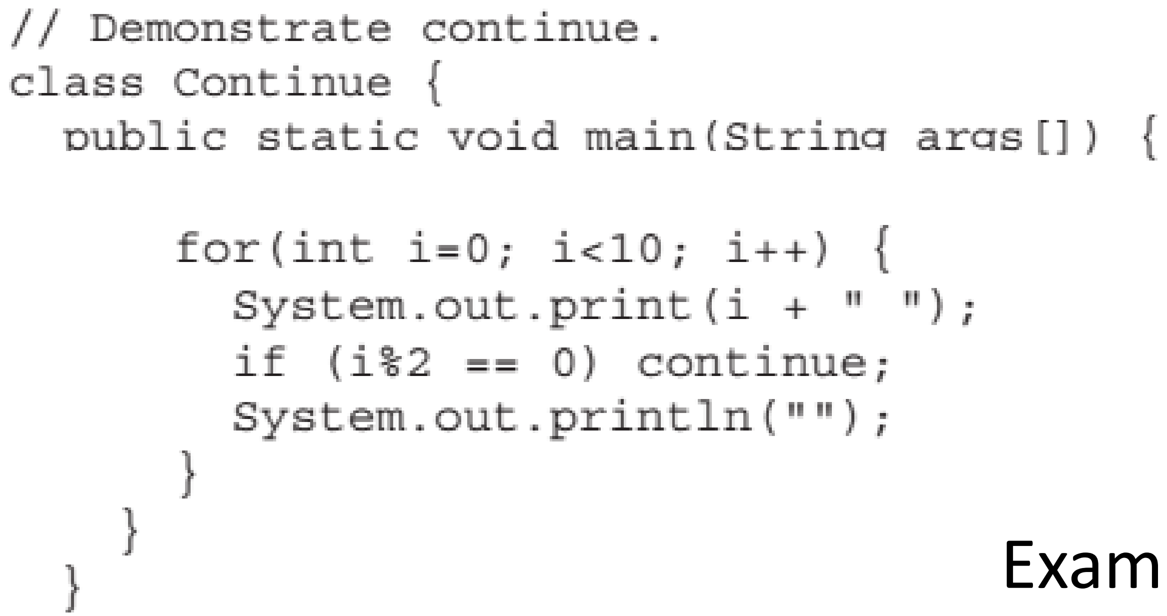
```
// Using break as a civilized form of goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    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.");
    }
  }
}
```



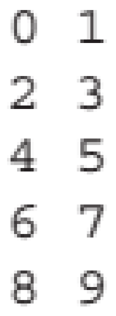
```
Before the break.
This is after second block.
```

Using continue

- If one 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.



```java
// 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("");
      }
    }
  }
```



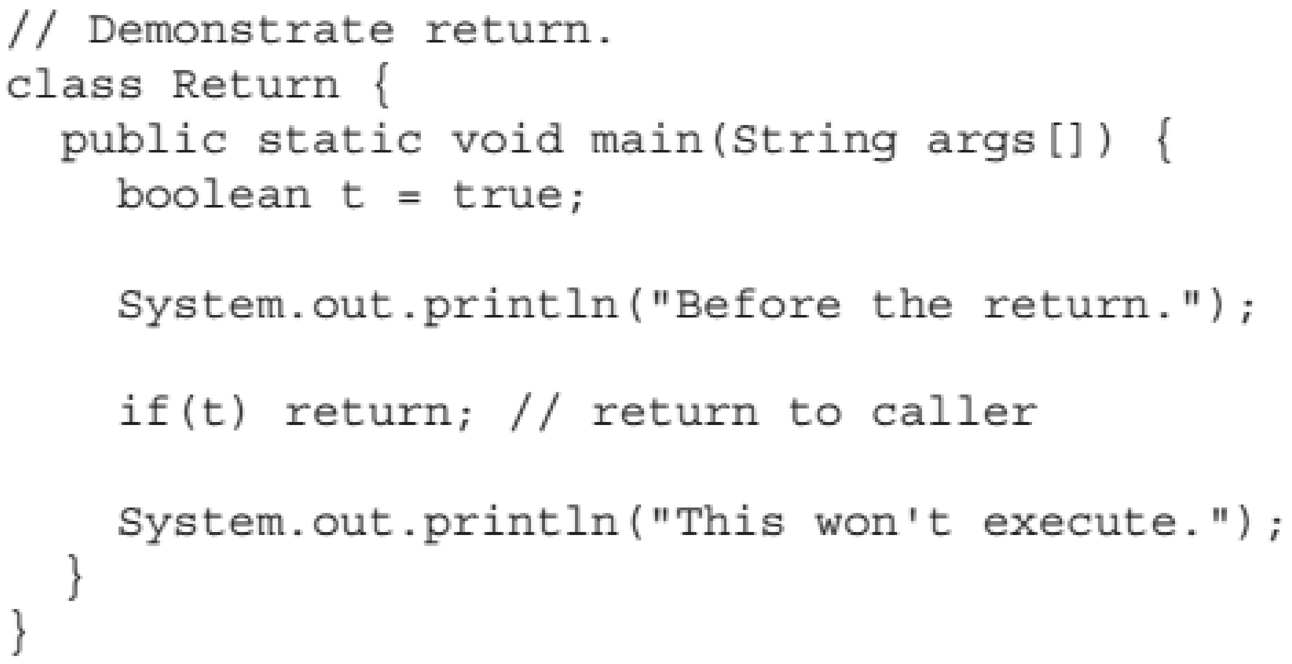
```
0 1
2 3
4 5
6 7
8 9
```

Example #19

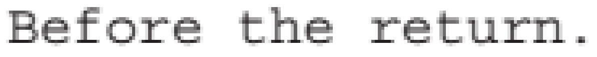
return

- The **return** statement is used to explicitly return from a method.

Example #20



```
// 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.");
  }
}
```



```
Before the return.
```

Tutorial #2

1. Write a Java program that takes a year from user and print whether that year is a leap year or not.
2. Write a Java program to prove that Euclid’s algorithm computes the greatest common divisor of two positive given integers.
3. Write a Java program to find the k largest elements in a given array. Elements in the array can be in any order
   - *Expected Output:*
     ```
     Original Array:
     [1, 4, 17, 7, 25, 3, 100]
     3 largest elements of the said array are:
     100 25 17
     ```
4. Write a Java program to move every zero to the right side of a given array of integers.
   - Original array: [0, 3, 4, 0, 1, 2, 5, 0]
     Result: [3, 4, 1, 2, 5, 0, 0, 0]
5. Write a Java program to test whether there are two integers x and y such that x^2 + y^2 is equal to a given positive number.