



UNIVERSITY WITH A PURPOSE



Object Oriented Programming



Type Conversion and Casting

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place.

There are no automatic conversions from the numeric types to **char** or **boolean**.

Casting Incompatible Types

To create a conversion between two incompatible types, you must use a cast.

A *cast* is simply an explicit type conversion. It has this general form:

(target-type) value

```
// Demonstrate casts.  
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
  
        System.out.println("\nConversion of int to byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
  
        System.out.println("\nConversion of double to int.");  
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
  
        System.out.println("\nConversion of double to byte.");  
        b = (byte) d;  
        System.out.println("d and b " + d + " " + b);  
    }  
}
```

This program generates the following output:

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

Arrays

An *array* is a group of like-typed variables that are referred to by a common name.

Arrays of any type can be created and may have one or more dimensions.

A specific element in an array is accessed by its index.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables.

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

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

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

Array Initialization

```
// Demonstrate a one-dimensional array.  
class Array {  
    public static void main(String args[]) {  
        int month_days[];  
        month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        month_days[3] = 30;  
        month_days[4] = 31;  
        month_days[5] = 30;  
        month_days[6] = 31;  
        month_days[7] = 31;  
        month_days[8] = 30;  
        month_days[9] = 31;  
        month_days[10] = 30;  
        month_days[11] = 31;  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}  
  
// An improved version of the previous program.  
class AutoArray {  
    public static void main(String args[]) {  
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  
                            30, 31 };  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays.

```
int twoD[][] = new int[4][5];  
  
// Demonstrate a two-dimensional array.  
class TwoDArray {  
    public static void main(String args[]) {  
        int twoD[][] = new int[4][5];  
        int i, j, k = 0;  
  
        for(i=0; i<4; i++)  
            for(j=0; j<5; j++) {  
                twoD[i][j] = k;  
                k++;  
  
            }  
  
        for(i=0; i<4; i++) {  
            for(j=0; j<5; j++)  
                System.out.print(twoD[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

This program generates the following output:

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

```

// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];

        int i, j, k = 0;

        for(i=0; i<4; i++) {
            for(j=0; j<i+1; j++) {
                twoD[i][j] = k;
                k++;
            }

            for(i=0; i<4; i++) {
                for(j=0; j<i+1; j++)
                    System.out.print(twoD[i][j] + " ");
                System.out.println();
            }
        }
    }
}

```

The array created by this program looks like this:

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

This program generates the following output:

```

0
1 2
3 4 5
6 7 8 9

```

```

// Initialize a two-dimensional array.
class Matrix {
    public static void main(String args[]) {
        double m[][] = {
            { 0*0, 1*0, 2*0, 3*0 },
            { 0*1, 1*1, 2*1, 3*1 },
            { 0*2, 1*2, 2*2, 3*2 },
            { 0*3, 1*3, 2*3, 3*3 }
        };
        int i, j;

        for(i=0; i<4; i++) {
            for(j=0; j<4; j++)
                System.out.print(m[i][j] + " ");
            System.out.println();
        }
    }
}

```

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

```

0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0

```

```

// Demonstrate a three-dimensional array.
class ThreeDMatrix {
    public static void main(String args[]) {
        int threeD[][][] = new int[3][4][5];
        int i, j, k;

        for(i=0; i<3; i++)
            for(j=0; j<4; j++)
                for(k=0; k<5; k++)
                    threeD[i][j][k] = i * j * k;

        for(i=0; i<3; i++) {
            for(j=0; j<4; j++) {
                for(k=0; k<5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println();
            }
            System.out.println();
        }
    }
}

```

This program generates the following output:

0 0 0 0 0	0 0 0 0 0
0 0 0 0 0	0 2 4 6 8
0 0 0 0 0	0 4 8 12 16
0 0 0 0 0	0 6 12 18 24
0 0 0 0 0	
0 1 2 3 4	
0 2 4 6 8	
0 3 6 9 12	

Alternative Array Declaration Syntax

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

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

For example, the following two declarations are equivalent:

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

The following declarations are also equivalent:

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

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

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

This is similar to

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

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

- 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**.

The Basic Arithmetic Operators

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

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

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.

```
// 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

- There are compound assignment operators for all of the arithmetic, binary operators.
- Any statement of the form `var = var op expression;` can be rewritten as `var op= expression;`

```
// 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

```
// 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
<code>~</code>	Bitwise unary NOT
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise exclusive OR
<code>>></code>	Shift right
<code>>>></code>	Shift right zero fill
<code><<</code>	Shift left
<code>&=</code>	Bitwise AND assignment
<code> =</code>	Bitwise OR assignment
<code>^=</code>	Bitwise exclusive OR assignment
<code>>>=</code>	Shift right assignment
<code>>>>=</code>	Shift right zero fill assignment
<code><<=</code>	Shift left assignment

- 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.

The Bitwise Logical Operators

A	B	A B	A & B	A ^ B	$\sim A$
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
// 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

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

The outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shift past bit position 31. Each left shift has the effect of doubling the original 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
```

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*

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

Each time you shift a value to the right, it divides that value by two—and discards any remainder.

Bitwise Operator Compound Assignments

```
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);
    }
}
```

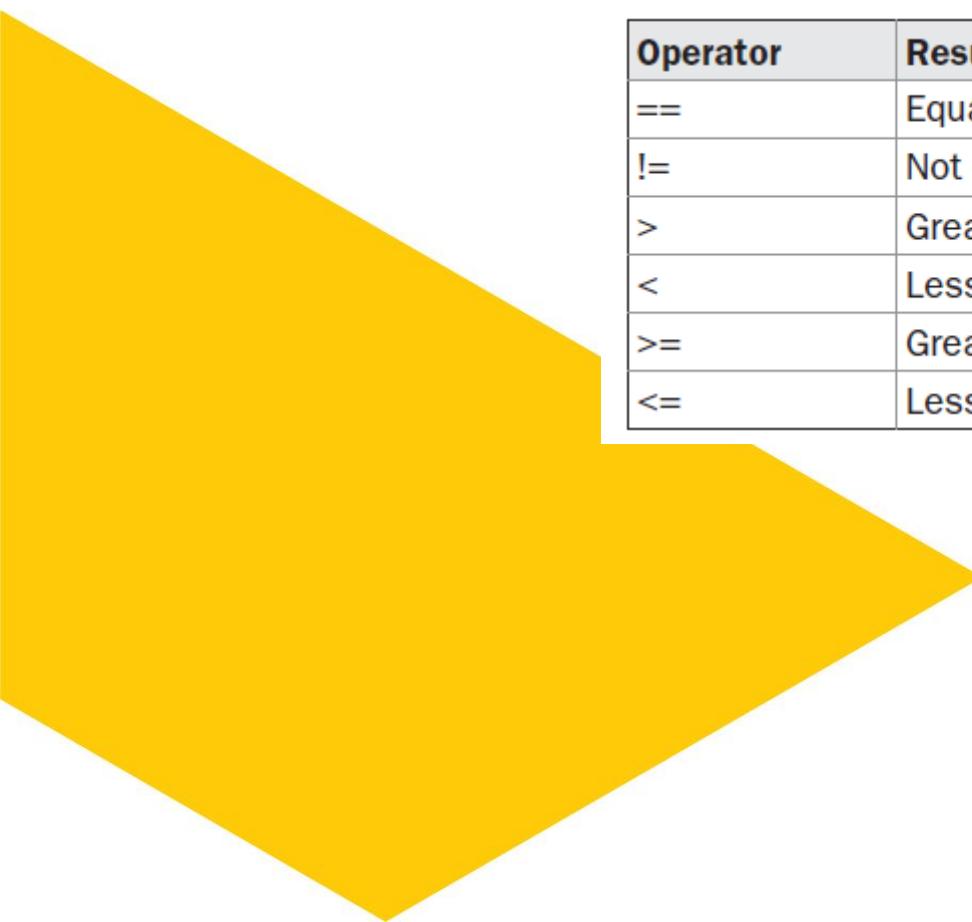
The output of this program is shown here:

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

Relational Operators

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

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



```
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

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

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);  
        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

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 ? 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`

- Here, `expression1` can be any expression that evaluates to a **boolean** value.
- If `expression1` is **true**, then `expression2` is evaluated; otherwise, `expression3` is evaluated.

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

Control Statements

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

- The **if** statement is Java's conditional branch statement.

- 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.

Nested ifs

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

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**.

```
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

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

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.

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

```
// 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:

```
i is zero.  
i is one.  
i is two.  
i is three.  
i is greater than 3.  
i is greater than 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");  
            }  
    }  
}
```

This program generates the following output:

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

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**.

These statements create what we commonly call *loops*.

While

- 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--;  
        }  
    }  
}
```

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
```

- 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.

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

This program finds the midpoint between i and j. It generates the following output:

```
Midpoint is 150
```

do-while

- The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

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

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

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

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
```

for

- 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.
- 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.
- 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.

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

The For-Each Version of the for Loop

Java adds the for-each capability by enhancing the **for** statement.

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:

```
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.

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.

```
// 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

```
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
```

```
// Use break with a for-each style for.  
class ForEach2 {  
    public static void main(String args[]) {  
        int sum = 0;  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
        // use for to display and sum the values  
        for(int x : nums) {  
            System.out.println("Value is: " + x);  
            sum += x;  
            if(x == 5) break; // stop the loop when 5 is obtained  
        }  
        System.out.println("Summation of first 5 elements: " + sum);  
    }  
}
```

This is the output produced:

```
Value is: 1  
Value is: 2  
Value is: 3  
Value is: 4  
Value is: 5  
Summation of first 5 elements: 15
```

Jump Statements

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

These statements transfer control to another part of your program.

Using break

- 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<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;`
- label* is the name of a label that identifies a block of code.

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

Running this program generates the following output:

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

Using continue

- A **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.

```
// 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
```

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

```
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 **return** statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.
- At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method.
- Thus, the **return** statement immediately terminates the method in which it is executed.

```
// 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.

References

Gosling, J., Holmes, D. C., & Arnold, K. (2005). The Java programming language.