Chapter 2.2

Operators in Java

An operator, in Java, is a special symbol performing specific operations on one, two or three operands and then returning a result.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators. Assume integer variable A holds 10 and variable B holds 20, then

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
-(Subtraction)	Subtracts right-hand operand from left- hand operand	A – B will give -10
*(Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B—gives 19

Relational Operators

There are following relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int,short, char, and byte. Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b =13; now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators. Assume integer variable A holds 60 and variable B holds 13 the

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 11
>>> (zero fill rightshift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Logical Operators

The following table lists the logical operators. Assume Boolean Variables A holds true and variable B holds false then

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B)

Assignment Operators

Following are the assignment operators supported by Java language

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+ =	Add AND assignment operator. It adds right operand to the left operand and assign the result to left.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C /A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C%A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >>2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C&2
^=	bitwise exclusive OR and assignment operator	C ^= 2 issame as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C | = 2 issame as C = C | 2

Conditional Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as

```
variable x = (expression) ? value if true : value if false
```

Control Statements

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 our 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 our 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

The if statement is Java's conditional branch statement. It can be used to route programexecution through two different paths. Here is the general form of the if statement:

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

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

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

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

Nested if

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;
        else a=c;
}
else a=d;
```

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)
  statements;
else if(condition)
  statements;
else if(condition)
  statements;
……
……
else
 statements;
```

Switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

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

Iteration Statement

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. A loop statement allows us to execute a statement or group of statements multiple times.

for loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A for loop is useful when you know how many times a task is to be repeated. The syntax of a for loop is:

```
for(initialization; Boolean_expression; update) {
        // Statements
}
```

while loop

A while loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true. The syntax of a while loop is:

```
while(Boolean_expression) {
        // Statements
}
```

Here, key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

do while loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed toexecute at least one time. Following is the syntax:

```
do {
        // Statements
}while(Boolean_expressions);
```

Nested Loop

Like all other programming languages, Java allows loops to be nested. That is, one loop maybe inside another. For example, here is a program that nests for loops:

```
class StarPattern {
   public static void main(String[] args) {
      for(int i=1;i<=5;i++) {
         for(int j=1;j<=i;j++) {
            System.out.print("* ");
         }
         System.out.print("\n");
      }
   }
}
```

The output produced by this program is shown here:

```
*

* *

* * *

* * * *

* * * * *
```

Jump Statements

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of our 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, by passing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

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

This program generates the following output:

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

As we can see, although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when i equals 10.

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 iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

In a while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
// Demonstrate Continue
class Continue {
   public static void main(String[] args) {
      for(int i=0;i<10;i++) {
         System.out.print(i+" ");
         if(i%2==0) continue;
         System.out.println("");
      }
   }
}
```

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

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

Using return

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

Example:

```
class A {
   int a,b,sum;
   public int add() {
      a=10;
      b=15;
      sum=a+b;
      return sum;
   }
}
class B {
   public static void main(String[] args) {
```

```
        A obj=new A();
        int res=obj.add();
        System.out.print("Sum of two numbers is:"+res);
    }
}
```

Output:

Sum of two numbers is:25

Differences between For, While & Do While Loop

For Loop	While Loop	Do While Loop
Initialization is done inside the loop statement.	Initialization is done outside the loop statement.	Initialization is done outside the loop statement.
Condition is checked before each iteration.	Condition is checked before each iteration.	Condition is checked after each iteration.
The loop may not execute even once.	The loop may not execute even one.	The loop executes at least once.

Syntax:
for(initialization;condition;updating)
{
// statements;
}
	Syntax:
	while(condition)
	{
	// statements;
	}
		Syntax:
		do
		{
		// statements;
		}
		while(condition);

It is also called counter controlled loop as loop is controlled by a counter value, at each iteration counter value will increase or decrease.	It does not need a counter value for its execution.	It does not need a conter value for its execution.
Increment / Decrement operation is specified within the loop statement.	Increment / Decrement operation is done inside the loop.	Increment / Decrement operation is done inside the loop.
It is Entry – Controlled Loop.	It is Entry – Controlled Loop.	It is Exit – Controlled Loop.
More compact and often easier to read when iteration count is known.	Can be easier to read for loops with complex conditions.	Can be easier to read when ensuring at least one execution.