Reading Material

Operators and Statements

Haaris Infotech

WEEK 2

OPERATORS AND STATEMENTS

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: assignment, arithmetic, relational, and logical. Java also defines some additional operators that handle certain special situations.

OPERATORS

Assignment Operators

The assignment statements have the following syntax:

<variable> = <expression>

The destination variable and the source expression must be type compatible. The destination variable must also have been declared. Since variable can store either primitive data values or object references, <expression> evaluates to either a primitive data value or an object reference. When assigning a value to a primitive, size matters. Be sure you know when implicit casting will occur, when explicit casting is necessary, and when truncation might occur.

Assigning primitive value

```
int a, b;
a = 2;     // 2 is assigned to variable a
b = 5;     // 5 is assigned to variable b
```

Assigning references

```
Home home1 = new Home();  // new object created

Home home2 = home1;       // assigning the reference of home1
in home2
```

In the above example home1 and home2 points to the same reference. Any change made to attribute in home2, same change will be reflected in the attribute of home1 also. But interesting thing is that if you assign null value to home2 it will be applied only on home2 and not on home1. home1 will still be holding the reference and home2 will be nullified.

Compound Assignment Operators

+=, -=, *=, and /=

Examples

```
int i;
i += 5; // it is equal to i = i + 5
int a;
a -= 5; // it is equal to i = i - 5
int b;
b *= 5  // it is equal to i = i * 5
int c;
c /= 5  // it is equal to i = i / 5
```

Arithmetic Operators

The arithmetic operators are used to construct mathematical expressions as in algrebra. Their operands are of numeric type.

+ addition	// i = i + 1
- subtraction	// i = i - 1
* multiplication	// i = i * 1
/ division	// i = i / 1. This operator only returns the quotient.

% remainder operator // i = i % 1. This operator returns the remainder

// value after division.

++ increment operator // i++ . It increments the value by one.

-- decrement operator // i-- . It decrements the value by one.

If you have any doubt about the usage of division operator and remainder operator then please try the below code.

Code for Division operator:

```
publicDivisionExample {

publicstatic void main(Stringargs[]) {

    int a = 6;
System.out.println(a / 4);

   }
}
```

Code for remainder operator

```
publicRemainderOpertor{

publicstaticvoid main(Stringargs[]){

int a = 6;
System.out.println(a % 4);
}

}
```

In Java, + operator is also used for String concatenation. Like an example given below

```
publicStringConcatenation {

publicstatic void main(String[] args) {

StringfirstName = "FirstName";

StringlastName = "LastName";

String name = firstName + " " + lastName;

System.out.println("Name is " + name);

  }

}
```

Relational Operators

Java has six relational operators

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

Less than sign is used to check whether the value of the first variable is less than the value of the second variable.

```
publicLessThanExample {

publicstatic void main(Stringargs[]) {

    int a = 5;
    int b = 10;

    if(a < b) {

System.out.println("a is less than b");
    }
  }
}
```

Less than or equal to sign is used to check whether the value of the first variable is less than or equal to the value of the second variable.

```
publicLessThanEqualExample {

publicstatic void main(Stringargs[]) {

    int a = 10;
    int b = 10;

    if(a <= b) {

System.out.println("a is less than or equal to b");
    }
  }
}
```

Greater than or equal to sign is used to check whether the value of the first variable is greater than or equal to the value of the second variable.

```
publicGreaterThanEqualExample {

publicstatic void main(Stringargs[]) {

    int a = 10;
    int b = 10;
    if(a => b) {

System.out.println("a is greater than or equal to b");
    }
  }
}
```

Equal to sign is used to check whether the value of the first variable is equal to the value of the second variable.

```
publicEqualToExample {

publicstatic void main(Stringargs[]) {

    int a = 10;
    int b = 10;

    if(a == b) {

System.out.println("a is equal to b");
    }
  }
}
```

Not equal to sign is used to check whether the value of the first variable is not equal to the value of the second variable.

```
publicNotEqualExample {

publicstatic void main(Stringargs[]) {

    int a = 10;
    int b = 5;

    if(a != b) {

System.out.println("a is not equal to b");
    }
  }
}
```

Logical Operators

These logical operators work only on boolean operands. Their return values are always boolean.

?: Ternary if-then-else

& Logical AND

&& Short-circuit AND

| Logical OR

|| Short-circuit OR

^ Logical XOR

== Equal to

!= Not equal to

! Logical unary NOT

Now we will see the usage of these operators.

<u>?: Ternary if-then-else</u>

```
publicclassTernaryOperatorExample {

     publicstatic void main(String[] args) {

          char ans = 'y';
          // (?) condition is checked
         // (:) value is returned based on the condition.
If
          // it is true then the first expression is
returned
          // otherwise the second one.
          String value = ans == 'y' ? "IT'S YES" : "OH
NO!!!";
          System.out.println(value);
     }
}
```

& Logical AND

```
publicclassANDOperatorExample {

     publicstatic void main(String[] args) {

          char ans = 'y';
          int count = 1;

          if(ans == 'y' & count == 0) {
               System.out.println("Count is Zero.");
          }

                if(ans == 'y' & count == 1) {
                    System.out.println("Count is One.");
                }

                 if(ans == 'y' & count == 2) {
                    System.out.println("Count is Two.");
                }
            }
        }
```

&& Short-circuit AND

```
        publicclassShortCircuitANDOperatorExample{

            publicstaticvoid main(String[]args){

                int hour = 9;
                int minute = 30;

                // short circuit AND operator isdiffernt from
                // normal AND operator. If happens like this
        that
                // the when first condition is checked and if
        it
                // is true then only it checks for the second
        condition
                // that is on the other side of the &&.
                if(hour == getHour()&& minute == getMinute()){
```

```
                System.out.println("It is 10:30");
            }


        }

        publicstaticintgetHour(){
            System.out.println("Return Hour");
            return10;
        }

        publicstaticintgetMinute(){
            System.out.println("Return Minute");
            return30;
        }
    }
```

Logical OR

```
    publicclassORExample {

        publicstatic void main(String[] args) {

            int x = 10;
            if(x == 5 | x ==10) {
                System.out.println("Value of x can be
    divided by 10");
```

```
            }
        }
    }
```

Short-circuit OR

```
    publicclassShortCircuitORExample {

        publicstatic void main(String[] args) {

            int hour = 10;
            int minute = 30;

            // if the first condition is true then it will
    not
            // check for the second condition. But if the
    first
            // condition is false then it will check for
    second
            // condition. In case of OR either one of the
    conditions
            // should be true.
            if(hour == getHour() || minute == getMinute())
    {
                System.out.println("Correct time of
    display this message");
            }
        }

        publicstatic int getHour() {
            System.out.println("returning hour");

            return 10;

        }


        publicstatic int getMinute() {

            System.out.println("returning minute");

            return 30;

        }

    }
```

Logical XOR

```
    publicclassXORExample {


        publicstatic void main(String[] args) {


            int a = 1;

            int b = 2;


            int c = 3;

            int d = 4;


            // XOR will return true only if the conditions

            // are mutually exclusive.

            System.out.println(a == 1 ^ b == 2); // both
    the conditions are true

            System.out.println(c == 4 ^ d == 3); // both
    the conditions are false
```

```
            System.out.println(c == 4 ^ d == 4); // only
second condition is true

     }

}
```

Equal to

```
publicclassEqualToExample {


     publicstatic void main(String[] args) {


          char ans = 'y';


          if(ans == 'y') {

               System.out.println("It is YES.");

          }

     }

}
```

Not equal to

```
publicclassNotEqualToExample {


     publicstatic void main(String[] args) {


          int value = 4;

          if (value != 10) {

               System.out.println("Value is not equal to
10");

          }
```

```
        }
    }
```

Logical unary NOT

```
    publicclassUnaryNotExample {

        publicstatic void main(String[] args) {

            boolean flag = false;

            // Here the value of the flag is false. But if
    block
            // is only executed when the condition is true.
    So
            // in cases we can use Unary Not (!). This
    negates the
            // boolean value of the flag. If the value of
    flag is
            // false and used with Unary Not then it will
    return
            // true and vice-versa.
            if(!flag) {
                System.out.println("Hello world.");
            }
        }
    }
```

STATEMENTS

The control statements provided in Java are basically used to cause the proper flow of execution in the advanced and branched Java programs. These control statements can be categorized into the following:

Selection Statements

Selection statements, as self-explanatory helps the programmer to choose different paths of execution based upon the outcome of an expression or the state of a variable.

```
if(isOpen) {  //isOpen is a boolean variable

System.out.println("The Door is OPEN");

} else {

System.out.println("The Door is CLOSE);

}
<>
```

Iteration Statements

Iteration statements as the term states, is the repetition of the block of code in a program.

```
int i = 1;

 while(i< 5) {


System.out.println("The current value of variable i is " + i);

i = i + 1;

}
```

Jump Statements

Jump statements allow your program to execute in a nonlinear fashion.

```
// break is a jump statement


  int count = 5;
  while(true) {
    if(count == 5) {


System.out.println("We will use break statement " +
                              "to jump out of the while
block");
break;
    }
    // code for some process.
}
```

CONTROL STRUCTURES

Control structures are programming blocks that can change the path we take through those instructions.

There are three kinds of control structures:

- Conditional Branches, which we use for choosing between two or more paths. There are three types in Java: if/else/else if, ternary operator and switch.
- Loops that are used to iterate through multiple values/objects and repeatedly run specific code blocks. The basic loop types in Java are for, while and do while.

- Branching Statements, which are used to alter the flow of control in loops. There are two types in Java: break and continue.

These are used to choose the path for execution. There are some types of control statements:

if statement: It is a simple decision-making statement. It is used to decide whether the statement or block of statements will be executed or not. Block of statements will be executed if the given condition is true otherwise the block of the statement will be skipped.

```
if(condition) {

// If condition is true then block of statements will be
executed

}
```

nested if statement: Nested if statements mean an if statement inside an if statement. The inner block of if statement will be executed only if the outer block condition is true.

if-else statement: In if statement the block of statements will be executed if the given condition is true otherwise block of the statement will be skipped. But we also want to execute the same block of code if the condition is false. Here we come on the if-else statement. An if-else statement, there are two blocks one is if block and another is else blocked. If a certain condition is true, then if block will be executed otherwise else block will be executed.

```
if (condition) {

// It will be print if block condition is true.

}

else
```

```
{
// It will be print if block condition is false.
}
```

if-else if statement/ ladder if statements: If we want to execute the different codes based on different conditions then we can use if-else-if. It is also known as if-else if ladder. This statement is always be executed from the top down. During the execution of conditions if any condition founds true, then the statement associated with that if it is executed, and the rest of the code will be skipped. If none of the conditions is true, then the final else statement will be executed.

```
if(condition) {
// If condition is true then this block of statements will
be executed
}
else if(condition) {
// If condition is true then this block of statements will
be executed
} . . .
Else
 {
// If none of condition is true, then this block of
statements will be executed
 }
```

Switch statement: The switch statement is like the if-else-if ladder statement. To reduce the code complexity of the if-else-if ladder switch statement comes. In a switch, the statement executes one statement from multiple statements based on condition. In the switch statements, we have a number of choices and we can perform a different task for each choice.

```
switch(variable/expression)
{
case value1 : // code inside the case value1
break; //
optional case value2 : // code inside the case value2
break; //
optional . . . default : // code inside the default case .
 }
```

These are used to iterate the instruction for multiple times.

for loop: If a user wants to execute the block of code several times. It means the number of iterations is fixed. JAVA provides a concise way of writing the loop structure.

```
      for(initialization; condition; increment/decrement)
                                {
                       // Body of for loop
                                }
```

A while loop allows code to be executed repeatedly until a condition is satisfied. If the number of iterations is not fixed, it is recommended to use a while loop. It is also known as an entry control loop.

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

do-while loop: It is like while loop with the only difference that it checks for the condition after the execution of the body of the loop. It is also known as Exit Control Loop.

```
do

{

// Body of loop

} While(condition);
```

1. Enhanced for loop: This is mainly used to traverse collection of elements including arrays.

Syntax

for(declaration : expression) { // Statements }

- Declaration – The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- Expression – This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

These are used to alter the flow of control in loops.

1. break: If a break statement is encountered inside a loop, the loop is immediately terminated, and the control of the program goes to the next statement following the loop. It is used along with if statement in loops. If the condition is true, then the break statement terminates the loop immediately.

break;

1. continue: The continue statement is used in a loop control structure. If you want to skip some code and need to jump to the next iteration of the loop immediately. Then you can use a continue statement. It can be used for loop or while loop.

continue;

Arrays

An array is a data structure that defines an indexed collection of a fixed number of homogeneous data elements. In java arrays are objects that store multiple variables of the same type. Arrays can hold either primitives or object references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements. A position in the array is indicated by a non-negative integer value called index. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot be increased to accommodate more elements.

Declaring an Array

Arrays are declared by stating the type of elements the array will hold, which can be an object or primitive followed by square brackets to the left or right of the identifier.

```
<element type>[] <array name>;
or
<element type><array name>[];
```

Arrays can be single dimensional, two dimensional or multidimensional. Declaring a single dimensional array is very simple. The most common way to construct an array is to use the new keyword that allocates memory to any object. Remember arrays are treated as objects in Java. The following is an example of constructing a single dimensional array of type long:

```
long[] empSalaries = new long[10];
```

As you can see that the above statement will allocate memory to the array variable empSalaries and the size of the array would be 10. In java index begin from 0. That means in order to access the first element of an array you need to do something like

```
empSalaries[0];
publicSingleDimensionalArray {

  long[] empSalaries = new long[] {10000, 20000, 30000,
40000};
publicstatic void main(String[] args) {
System.out.println("Value of the first element in the array: "
+ empSalaries[0];
  }
}
```

An array can be initialized in many ways:

1) `long[] empSalaries = {10000L, 20000L, 30000L, 40000L};`

// L means long value. It is required to distinguish between integer value and

// long value. By default it is integer always.

2) `long[] empSalaries = new long[] {10000L, 20000L, 30000L, 40000L};`

3) `long[] empSalaries = new long[4];`

```
empSalaries[0] = 10000L;
empSalaries[1] = 20000L;
empSalaries[2] = 30000L;
empSalaries[4] = 40000L;
```

Let us check out some invalid array declarations

```
String names[] = new String[];  // We have to mention the size
of
                                // the array when using new
operator.


int ars[] = new int{1,2,3,4,5}; // square([]) brackets are
missing.
int ars[] = new int[5]{1,2,3,4,5}; // size of the array should
not be mentioned
```

```
                                        // when declaring an array
like this.
int ars[4]; // it is invalid because memory can be allocated
only using new operator.
int ars[];
ars = {1,2,3,4,5};
```

Multidimensional Array

In Java multidimensional arrays are actually arrays of arrays. These as you might expect, look and act like regular multidimensional arrays. Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. Let us take an example of two-dimensional array which is a form of multidimensional array.

```
int matrix = new int[3][2];
```

This very much resembles to a 3 x 2 matrix. Where 3 represents rows and 2 represents columns.

Valid array declarations

```
Integer ars[][] = new Integer[4][3];  // Integer is a wrapper
class for int.
int ars[][] = new int[][]{{1,2,3}, {2,4,6}, {3,4,7}, {5,6,7}};
```

It will be like a 4 x 3 matrix

```
 _     _
| 1 2 3|
| 2 4 6|
| 3  4  7|

| 5  6  7|

+-      -+
```

Iterate through a multidimensional array

```
Public classMultiDimensionalArrayExample {

Public static void main(String[] args) {

    int ars[][] = new int[][]{{1,2,3}, {2,4,6}, {3,4,7},
{5,6,7}};

    for(int i = 0; i< 4; i++) {
      for(int j = 0; j < 3; j++) {
System.out.println(ars[i][j]);
      }
    }
  }
```