The Arithmetic Operators

The arithmetic operators refer to the standard mathematical operators: addition, subtraction, multiplication, division and modulus.

Op.	Use	Description
+	x + y	adds x and y
-	x - y	subtracts y from x
*	x * y	multiplies x by y
/	x / y	divides x by y
%	x % y	computes the reminder of dividing x by y

Examples:

```
   i + 1      (x * y) % 5      b * b - 4 * a * c
```

Some remarks for integer arithmetic operators:

- The result contains only the low-order bits of the mathematical result in case of the arithmetic overflow.

Unary Operators

Java's unary operators can use either **prefix** or **postfix** notation.

Operator	Use	Description
+	+op	promotes op to int if it is a byte, short or char
-	-op	arithmetically negates op
++	++op	increments op by 1; evaluates to value of op before the incrementation
++	op++	increments op by 1; evaluates to value of op after the incrementation
--	--op	decrements op by 1; evaluates to value of op before the decrementation
--	op--	decrements op by 1; evaluates to value of op after the decrementation

Examples:

```
    -x       +(x * y)       i++       a[j--]++
```

Examples of use of ++ and --

Code:

```
int x = 5; int y;
y = x++;
```

Results:

```
x = 6      y = 5
```

Code:

```
int x = 5; int y = 11; int z;
z = --x;
x = 2 * (y++ + 3) - x;
```

Results:

```
x = 24    y = 12     z = 4
```

Relational Operators

Relational operators generate a **boolean** result.

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Examples:

```
   i + 1 < n          x == h[2*i+1]          a != b
```

Conditional Operators

Relational operators are often used with conditional operators.

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
||	op1 || op2	either op1 or op2 is both true, conditionally evaluates op2
!	!op1	op1 is false

Examples:

```
!(n >= 0)
(i < n) && (a[i++] > 0)
```

If (i>=n) then the value of i is not changed. If (i<n) then i is incremented by 1.

Bitwise Operators

The bitwise operators allow to manipulate individual bits in an integral primitive data type. Bitwise operators perform boolean algebra on the corresponding bits in the two arguments to produce the result.

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op	bitwise complement

Examples:

```
0x36 & 0x0F      0x06     (00110110 & 00001111)
0x36 | 0x80      0xB6     (00110110 | 10000000)
0x36 ^ 0x07      0x31     (00110110 ^ 00000111)
```

Shift Operators

Shift operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself.

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2

```
0x36 << 2       0xD8       (00110110  ->  11011000)

-1              -1 (decimal)
-1 >> 1         -1 (decimal)
-1 >>> 1        2147483647 (decimal)

-1              11111111111111111111111111111111 (binary)
-1 >> 1         11111111111111111111111111111111 (binary)
-1 >>> 1        01111111111111111111111111111111 (binary)
```

Ternary Operator (?:)

The ternary operator allows to avaluate expresseion in two diferrent ways depending on some condition.

The expression is of the form:

```
cond ? expr1 : expr2
```

The boolean condition cond is evaluated first. If it is **true** then expr1 is evaluated and the resulting value is the value of the whole expression. When cond evaluates to **false** then expr2 is evaluated and the resulting value is the value of the whole expression.

Example:

```
(n > 1) ? (a + b) : (a * b)
```

When (n>1) then the result is (a+b), otherwise the result is (a*b).

Assignment Operators

The basic form of assignment is

```
expr1 = expr2
```

Evaluation:

1. The left hand side (expr1) is evaluated. It must by an **lvalue** – a variable, an element of an array, a field.
2. The right hand side (expr2) is evaluated.
3. The value of the right hand side is stored into the place denoted by the left hand side.
4. The value of the whole expression is the value of the right hand side.

Examples of assignment expressions:

```
x = (z + y) * a[i]
a[i++] = x + y
```

Assignment Operators (cont.)

Examples of assignment statements:

```
x = (z + y) * a[i];
a[i++] = x + y;
```

Note that an assignment expression is not the same thing as an assignment statement.

The following construction is legal, but the resulting code is not very clear:

```
int y, x;
x = 3 * (y = 2) + 1;
```

The results are:

```
x = 7      y = 2
```

Compound Assignment Operators

There other assignment operators of the form op= where op is some binary operator:

```
*=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
```

The meaning of

```
expr1 op= expr2
```

is the same as

```
expr1 = expr1 op expr2
```

except that expr1 is evaluated only once.

For example, the statement `x *= 6;`
has the same effect as `x = x * 6;`

Notice that `a[i++] += 3;`
is not the same as `a[i++] = a[i++] + 3;`

Cast Expression

The following assinment between variables of different types is possible:

```
byte b; int i;
    .
    .
    .
i = b;
```

The following assignment is **illegal**:

```
b = i;
```

It can be assigned using the cast of the form

```
(type)expr1
```

which transforms the value of expr1 to the type type as in the following code:

```
b = (byte)i;
```

Priority of Operators

Operators ordered by priority (from lowest to highest):

Pr.	Operators
1.	()
2.	[], postfix ++ and --
3.	unary +, unary -, ~, !, cast, prefix ++ and --
4.	*, /, %
5.	+, -
6.	<<, >>, >>>
7.	<, >, <=, >=, instanceof
8.	==, !=
9.	&
10.	^
11.	|
12.	&&
13.	||
14.	?:
15.	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=

Associativity of Operators

Most binary operators are associative to the left.

For example

```
a + b + c
```

has the same meaning as

```
(a + b) + c
```

An exception are the asignment operators that are associative to the right.

For example

```
a = b = c
```

has the same meaning as

```
a = (b = c)
```

Statements

One of the basic types of statements is an **assignment statement**:

```
a = b + c;
```

Assignment statement must end with semicolon (;).

Some other types of expressions can be also used as statements:

```
i++;
sum(a, b);
```

A **declation** can be also used as a statement:

```
int i;
double x, y, z;
```

A declaration can be combined with an assignment of an initial value:

```
int i = 4;
double x = 46.3, y, z = i * 2.0;
```

Blocks

Blocks are sequences of statements enclosed between { and }.

Example:

```
{
    a = 3;
    int b = a + 1;
    a = b * 2;
}
```

The **scope** of a declation of a local variable is from the place where it is declared to the end of the enclosing block.

A block can be used in any place where a single statement can be used.

Branching Statement

The **if-else** statement is probably the most basic way to control program flow.

```
if (value > value2) {
   result = 1;
}
else if (value1 < value2) {
   result = -1;
}
else {
   result = 0;
}
```

Similarly we can use:

```
if (value > value2) result = 1;
else if (value < value2) result = -1;
else result = 0;
```

Iteration Statements

Java provides three iteration statements. The statements repeat their bodies until controlling expression evaluates to **false**.

- **while**

```
int i = 0;
while (++i < 2)
   System.out.println("i: " + i);
```

- **do-while**

```
int i = 0;
do {
   System.out.println("i: " + i);
} while (++i < 2)
```

- **for**

```
int powerOfTwo = 1;
for (int i = 0; i < 16; i++)
   powerOfTwo <<= 1;
```

Driving Iteration Statements

Inside the body of any of the iteration statements flow of the loop can be controlled using **break** and **continue** statements. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

```
int i = 0;
while (true) {
   if (i > 20)
      break;
   if (i++ % 7 == 0)
      continue;
   i += 2;
}
```

Driving Iteration Statements

The **break** and **continue** normally only alter the closest looping structures. If there are nested statements, **labeled break** and **continue** can be used to alter outer looping structures.

```
int i = 0;
outer:
while (true) {
   while (true) {
      i++;
      if (i == 1)
         break;
      if (i == 4)
         break outer;
   }
   while (true) {
      i++;
      if (i == 2)
         continue;
      if (i == 3)
         continue outer;
   }
}
```

The switch Statement

The **switch** statement is used to test an **integral expression** against one or more possible cases.

```
char ch;
boolean whitespace;

switch (ch) {
   case ' ':
   case '\n':
   case '\t':
   case '\r':
      whitespace = true;
      break;
   default:
      whitespace = false;
}
```