

A group of four students are gathered around a table in a library, looking at a laptop screen. The background is filled with bookshelves. The image has a semi-transparent blue overlay on the left side and a semi-transparent red overlay at the bottom.

Working with Java Data Types

Operators

Operators

- In an expression such as `int x = 3 + 4;` the `+` is the *operator* and 3 and 4 are called *operands*. The `+` is a *binary* operator as it has 2 operands.
- Java operators are typically not overloaded i.e. an overloaded operator can appear in different contexts. There are 2 notable exceptions:
 - `+` can be used to append *String*'s together in addition to adding primitives
 - `&`, `|` can be used with both boolean and integral operands but the bit-twiddling variant (e.g. *int*'s on both sides of the operator) has not been examined since Java 1.4.



Order of Precedence

- Expressions are evaluated left to right by default.
- You can change this sequence, or *precedence* by adding parentheses.
- $*$, $/$, and $\%$ have a higher precedence than the $+$ or $-$ operators.
- $=$ (assignment) has the lowest precedence and as a result is performed last.



Order of Precedence

Operator	Symbols/Example
postfix	expr++, expr--
prefix	++expr, --expr
other unary operators	+, -, !, (type)
mult/div/modulus	*, /, %
addition/subtraction	+, -
relational operators	<, >, <=, >=, instanceof
equal/not equal	==, !=
logical operators	&, , ^
short-circuit operators	&&,
ternary operator	booleanExpr ? expr1 : expr2
assignment operators	=, +=, -=, *=, /=



Prefix/Postfix Operators

- Unary operators as they have only one operand.
- ++ operator; increment a variable by exactly one
- -- operator; decrement a variable by exactly one
- The operator is placed either before (prefix) or after (postfix) a variable to change its value and this can change the outcome of an expression.



Prefix/Postfix Operators

```
int x=3, y=4;  
x++;      // x is 4  
--y;      // y is 3  
System.out.println(x + " " + y); // 4 3  
System.out.println(x++ + " " + --y); // 4 2  
System.out.println(x + " " + y); // 5 2
```

Other Unary Operators

```
// unary operators (one operand)
int x = +6; // positive is the default
int y = -x;
System.out.println(x + " " + y); // 6 -6

int z = (int)3.45;
System.out.println(z); // 3

boolean b = true;
System.out.println(!b); // false
```

Arithmetic Operators

- + addition
- - subtraction
- * multiplication
- / division (integer division truncates)
- % modulus/remainder operator

```
int x=10, y=3;  
int div = x/y; // integer division truncates  
int mod = x%y; // keep remainder only  
System.out.println(div + " " + mod); // 3 1  
System.out.println(0 % 3); // 0
```


Arithmetic Operators - precedence

- `*`, `/`, and `%` have a higher precedence than the `+` or `-` operators.
- `=` (assignment) has the lowest precedence and as a result is performed last.

```
int res = 3 + 2 * 4;  
System.out.println(res); // 11  
res = (3 + 2) * 4;  
System.out.println(res); // 20  
res = 6 + 4 - 2;  
System.out.println(res); // 8  
res = 10 / 4 * 6;  
System.out.println(res); // 12
```

Arithmetic Operators

- Any maths operation involving *int*-types or smaller, results in an *int*.

```
byte b1 = 2, b2 = 3; // compiler "knows" that the int literals
                     // are in range => ok
byte b3 = 128;       // 128 is not in range (-128..+127)

byte b4 = b1 + b2;    // must cast int to byte
byte b6 = (byte) (b1 + b2); // note the brackets
```

String Concatenation Operator

- The + operator is used.
- If both operands are numbers, the + operator is the addition operator. However, if **either** operand is a String, the + operator becomes a String concatenation operator.

```
int a=3, b=2;
int res = a+b;
System.out.println(res); // 5
String s="abc";
String s1 = a + s;
String s2 = s + a;
System.out.println(s1 + " " + s2); // 3abc abc3

System.out.println("Output is "+ a + b); // Output is 32
System.out.println("Output is "+ (a + b)); // Output is 5
```


Relational Operators

- There are 6 relational operators:
 - `==`, `!=`, `>`, `>=`, `<`, `<=`
- Relational operators always result in a *boolean* value (*true/false*).

```
System.out.println(5.0 == 5); // true i.e. 5.0 == 5.0 (promotion)
System.out.println(5.1 == 5); // false i.e. 5.1 == 5.0 (promotion)
System.out.println(5.0 == 5L); // true i.e. 5.0 == 5.0 (promotion)
```

Equality/Inequality Operators

- == is the “equal to” operator
- != is the “not equal to” operator

```
// assignment is the = and "equal to" is the ==  
int x=8, y=9;  
System.out.println(x == y); // false  
System.out.println(x != y); // true
```

Logical Operators

- Logical AND i.e. &&

- boolean expressions as operands
- short-circuits i.e. && evaluates the left side of the expression first and if it resolves to *false*, the right side of the expression is not evaluated because && knows the complete expression cannot be *true*, i.e. $F \ \&\& \ T == F$

```
boolean b1 = false, b2 = true;
boolean res = b1 && (b2=false); // F &&
System.out.println(b1 + " " + b2 + " " + res); // false true false
```


Logical Operators

- Logical OR i.e. ||

- boolean expressions as operands

- short-circuits i.e. || evaluates the left side of the expression first and if it resolves to *true*, the right side of the expression is not evaluated because || knows immediately that the complete expression is *true*, i.e. $T \parallel F == T$

```
boolean b1 = false, b2 = true;  
boolean res = b2 || (b1=true); // T ||  
System.out.println(b1 + " " + b2 + " " + res); // false true true
```

Logical Operators

- Bitwise AND i.e. &
 - boolean expressions as operands (and integral operands)
 - does NOT short-circuit

```
boolean b1 = false, b2 = true;  
boolean res = b1 & (b2=false); // F & F  
System.out.println(b1 + " " + b2 + " " + res); // false false false
```

Logical Operators

- Bitwise OR i.e. |
 - boolean expressions as operands (and integral operands)
 - does NOT short-circuit

```
boolean b1 = false, b2 = true;  
boolean res = b2 | (b1=true); // T | T  
System.out.println(b1 + " " + b2 + " " + res); // true true true
```


Logical Operators

- Bitwise XOR i.e. ^
 - boolean expressions as operands (and integral operands)
 - one or the other but not both
 - does NOT short-circuit

```
boolean b1 = (5 > 1) ^ (10 < 20); // T ^ T == F
boolean b2 = (5 > 10) ^ (10 < 20); // F ^ T == T
System.out.println(b1 + " " + b2); // false true
```

Bitwise Operators (integral operands)

```
byte b1 = 6 & 8;    // int operands, both must be on
                    //      110
                    //    & 1000
                    //    ====
                    //    0000
byte b2 = 7 | 9;    // int operands, one or the other or both
                    //      111
                    //    & 1001
                    //    ====
                    //    1111 (15)
byte b3 = 5 ^ 4;    // int operands, one or the other but not both
                    //      101
                    //    & 100
                    //    ====
                    //    001
System.out.println(b1 + " " + b2 + " " + b3); // 0 15 1
```

Ternary Operator

- Ternary operator `_? _ : _`
 - 3 operands
 - used to evaluate *boolean* expressions, just like an *if* statement, except that, instead of executing a block of code if the test is true, a conditional operator will assign a value to a variable.
 - the goal of the conditional operator is to decide which of two values to assign to a variable




```
// ? and : are used; parentheses are optional
// x = (boolean expr) ? value to assign if true : value to assign if false
public class TernaryOp {
    public static void main(String[] args){
        int numPets=3;
        String status = (numPets<4) ? "Pet limit not exceeded"
                                   : "Too many pets";
        System.out.println(status); // "Pet limit not exceeded"

        int sizeOfYard=7;
        numPets=6;
        // nested
        status = (numPets<4) ? "Pet count OK"
                           : (sizeOfYard > 8) ? "Pet limit on edge"
                           : "Too many pets";
        System.out.println(status); // "Too many pets"
    }
}
```

Compound Assignment Operators

- In addition to =, Java also provides:

➤ += e.g. `x += y;` \Leftrightarrow `x = x+y;`

➤ -= e.g. `x -= y;` \Leftrightarrow `x = x-y;`

➤ *= e.g. `x *= y;` \Leftrightarrow `x = x*y;`

➤ /= e.g. `x /= y;` \Leftrightarrow `x = x/y;`

➤ note: in-built implicit cast

```
byte b1=3;  
int i1 = 4;  
b1 = b1 + i1; // byte = int ERROR!
```

```
b1+=i1; // no error => b1 = (byte) (b1 + i1);
```

Compound Assignment Operators - careful

```
int x = 2;
```

```
// tricky - bearing in mind that * has higher precedence than +  
// The expression on the RHS is always placed in parentheses and  
// therefore the expression evaluates as x = x * (RHS)
```

```
x *= 2 + 5; // x = (x * 2) + 5 == 9
```

```
           // x = x * (2 + 5) == 14 (the correct one)
```

```
System.out.println(x); // 14
```


Compound Assignment Operators - careful

```
int k = 1;  
// operand += operand (+= is the operator)  
// k = k + (RHS) i.e. k value of 1 will be used regardless of RHS  
// k = 1 + ( (k = 4) * (k + 2) )  
// k = 1 + ( 4 * 6)  
// k = 1 + 24  
// k = 25  
k += (k = 4) * (k + 2);  
System.out.println(k); // 25
```