Programming in Java - Part 02 Data types, operators, and flow control

Paolo Vercesi

ESTECO SpA

Variables and data types

Arrays

Operators

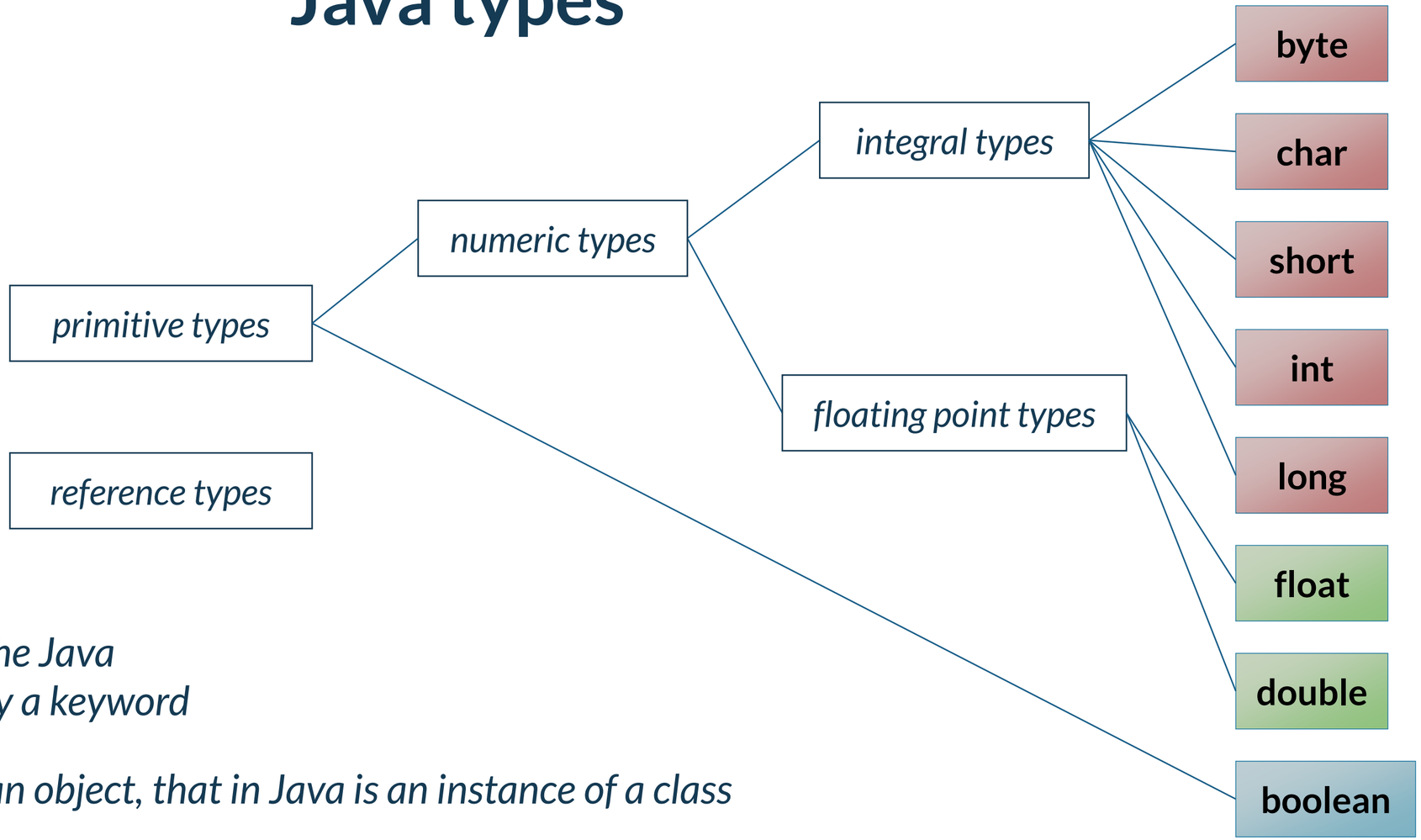
Flow control

Variables and data types

Java types

There are two kinds of types in Java, primitive data types and reference types

Correspondingly there are two kinds of data values that can be stored in variables, passed as arguments, returned by methods, or used in expressions: primitive values and reference values



Primitive types are predefined by the Java language, and they are identified by a keyword

A reference value is a reference to an object, that in Java is an instance of a class

The null "value" has no type and you can assign it to any reference

The void keyword does not represent a data type. It used to denote a method that does not return any value, e.g., the main() method. You cannot declare a variable of type void

Variable and constant definition

```
int x;
double d = 0.33;
float f = 0.22F;
char c = 'a';
boolean ready = true;
x = 15;
```

Variables are declared specifying their type and name, and initialized in the point of declaration, or later with the assignment expression

Variables cannot be used before initialization

Constants are declared with the word final in front, the specification of the initial value is compulsory

```
final double pi = 3.1415;
final int maxSize = 100_000;
final char lastLetter = 'z';
```

```
var f = 10.0; // a double variable
var i = 50;     // an int variable
```

Local variables can be declared without an explicitly declared type by using the so-called type inference

Data type ranges

Integral type	Width [bits]	Range
byte	*8*	*-128 to 127*
short	*16*	*-32,768 to 32767*
int	*32*	*-2,147,483,648 to 2,147,483,647*
long	*64*	*-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807*
char	*16*	*0 to 65535*

Floating point type	Width [bits]	Range
float	*32*	*1.4e-45 to 3.4e+38*
double	*64*	*4.9e-324 to 1.8e+308*

Differences with Python and C++	
Python	*No primitive data types, everything is an object, e.g., integers have unlimited precision*
C++	*The range of integer and floating-point type is implementation dependent with some constraints, e.g., minimum width for integer types. Furthermore, C++ allows integer types with the unsigned modifier*

Type conversion

When you assign a value (or a variable) of one primitive type to a variable of another primitive type, Java performs a so-called type conversion

TestCast.java

```
void main() {

    int a = 'x';          // 'x' is a character
    long b = 34;           // 34 is an int
    float c = 1002;        // 1002 is an int
    double d = 3.45F;      // 3.45F is a float

    int f = (int) b;            // b is a long
    float h = (float) d;        // d is a double
    byte g = (byte) c;          // c is a float
  }
```

When the magnitude of a numeric value cannot be preserved, it is compulsory to use the cast () operator

Type conversion and casting

Java performs automatic conversions when the magnitude of a numeric value is preserved,
widening primitive conversion

- *from int to long*
- *from long to double*
- *from float to double*
- *…*

When the magnitude of a numeric values cannot be preserved, you must declare the explicit type conversion, narrowing primitive conversion or casting

Narrowing conversion	Rules
from integer to integer (e.g., long to int)	*Integer component is reduced modulo the target type size*
from floating point to integer (e.g., double to int)	*Fractional component is truncated* *Integer component is reduced modulo the target type size*
from double to float	*The number is rounded to the closest float, including +Infinity and -Infinity*

Strings

Strings are not a primitive type, but they are objects of the String class

```
String a = "abc";
```

The variable a contains a reference to a String object, not the string "abc"

In Java, the '+' operator is left-associative, and when one operand is a String, the other operand is converted to a String and concatenation is performed

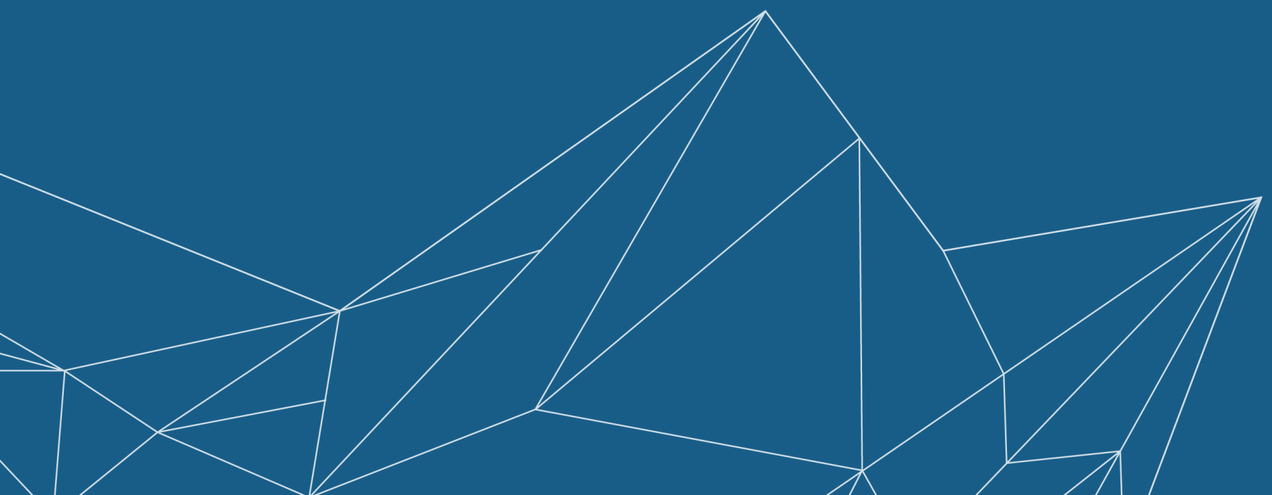
```
int cost = 2;
String b = "the cost is " + cost + " euro";
```

```
the cost is 2 euro
```

What's the output of?

```
String bb = "the cost is " + cost + cost + " euro";
IO.println(bb);
```

Arrays



Arrays

Arrays are used to group elements of the same type

The declaration does not specify a size nor allocates any memory

```
int[] a;
double[] b;
String[] c;
```

```
int size = 3;
double[] d1, d2;

d1 = new double[3];
d2 = new double[size];
```

We allocate memory for arrays by using the new operator

By using an array initializer, you can allocate and initialize an array while declared

```
int[] a = {13, 56, 2034, 4, 55};
double[] b = {1.23, 2.1};
String[] c = {"Java","is","great"};
```

Working with arrays

Java arrays are 0-based and every array has a field called length representing the length of the array

```
int len = a.length;
```

We access the array components by using the [] operator with an integer index from 0 to length-1

```
a[2] = 1000;
```

```
int[] a = new int[2];
IO.println(a[0]);
IO.println(a[1]);
```

```
0
0
```

Components of the arrays are initialized with the default value of the type

```
String[] b = new String[2];
IO.println(b[0]);
IO.println(b[1]);
```

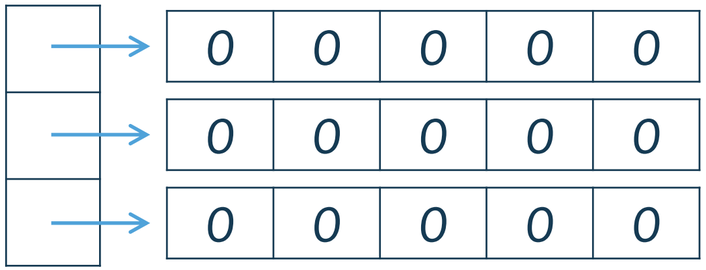
```
null
null
```

null is not the default values for objects, but components are initialized using null

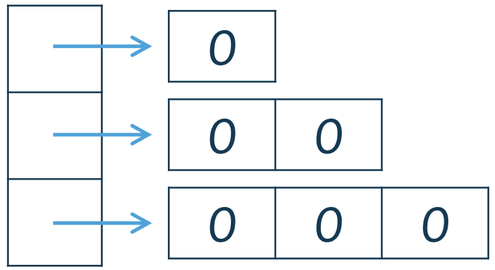
Multidimensional arrays

Multidimensional arrays are arrays of arrays

```
int[][] a = new int[3][5];
```




```
int[][] a = new int[3][];
a[0] = new int[1];
a[1] = new int[2];
a[2] = new int[3];
```



In Java is possible to define jagged arrays

When you allocate a multidimensional array only the first dimension is compulsory

Working with multidimensional arrays

Multidimensional arrays can be initialized by nesting array initializers

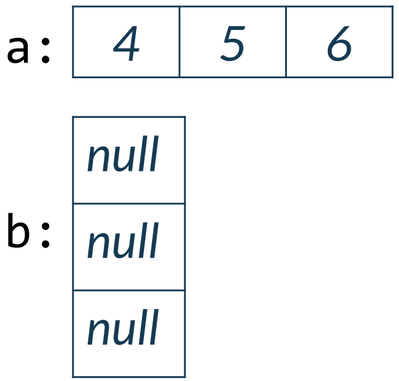
```
int[][] a = new int[][] {
                            {1},
                            {1, 1},
                            {1, 2, 1},
                            {1, 3, 3, 1}
                          };
int[][][] b = new int[2][][];
b[0] = new int[3][];
b[0][0] = new int[7];
b[0][1] = a[0];
b[0][2] = a[2];
b[1] = a;
```

In Java, arrays are objects, so a multidimensional array is an array of references to other arrays

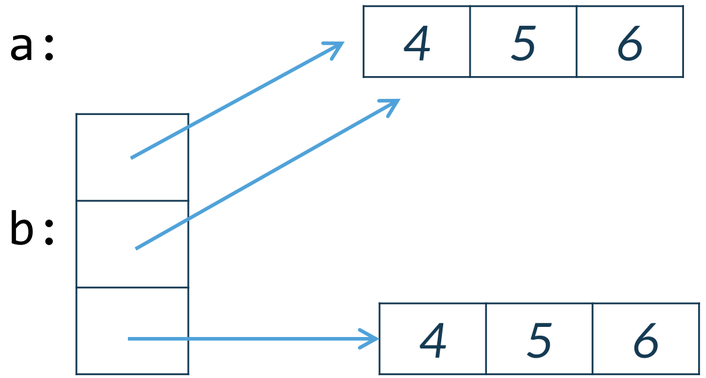
Objects, references, and the new operator

The concepts of objects and references have not yet been formally introduce, but you have already used them in combination with the new operator

```
int[] a = new int[] {4, 5, 6};

int[][] b = new int[3][];
```




```
b[0] = a;

b[1] = a;

b[2] = new int[] {4, 5, 6};
```



What does it happen to the array a *when you set* `b[1][1] = 3;` *?*

Exercise - A revised «Hello, World!»

```
$ java Hello.java Paolo
Hello, Paolo!
```

Hello.java

```
void main(String[] args) {
    IO.println("Hello, " + args[0] + "!" );
}
```

Operators

Operators & expressions

By combining variables and operators you can define expressions

Java defines the following type of operators

- *arithmetic*
- *assignment*
- *relational*
- *logical*
- *bit-level (rather rarely used)*

Arithmetic operators

Operator	Name
+	*addition*
-	*subtraction*
*	*multiplication*
/	*division*
%	*modulus*
++	*increment*
--	*decrement*

Increment and decrement, operates on variables, not on values

Type promotion in arithmetic expressions

byte, *short* and *char* operands are always converted to *int* in arithmetic expressions

If an operand is a long, the whole expression is converted to long

If one operand is a float, the whole expression is converted to float

If one operand is a double, the whole expression is converted to double

```
byte b1 = 3;
byte b2 = 4;
byte b3 = b1 * b2; // Incompatible types
byte b4 = (byte) (b1 * b2);
```

Can you explain this result?

```
double q = 3 / 2; // 1.0 !!!!
```

Assignment operators

Operator	Name
=	*assignment*
+=	*addition assignment*
-=	*subtraction assignment*
*=	*multiplication assignment*
/=	*division assignment*
%=	*modulus assignment*

`a op= b`

is equivalent to

`a = a op b`

Assignment.java

```
void main() {
  int x = 12, y = 33;
  double d = 2.45, e = 4.54;

  y %= x;
  d -= e;
  IO.println(y);
  IO.println(d);
}
```

```
$ java Assignment
9
-2.09
```

Relational operators

Operator	Name
==	*equality*
!=	*(unequality)*
<	*less than*
>	*great than*
<=	*less or equal than*
>=	*great or equal than*

Relational operators return boolean values

Relational.java

```
void main() {
  int x = 12, y = 33;

  IO.println(x < y);
  IO.println(x != y - 21);

  boolean test = x >= 10;
  IO.println(test);
}
```

```
$ java Relational
true
false
true
```

Logical operators

Operator	Name
&&	*conditional-and*
||	*conditional-or*
!	*not*

They operates on boolean values, and they return boolean values too

Logical.java

```
void main() {
  int x = 12, y = 33;
  double d = 2.45, e = 4.54;

  IO.println(x < y && d < e);
  IO.println(!(x < y));

  boolean test = 'a' > 'z';
  IO.println(test || d - 2.1 > 0);
}
```

```
$ java Logical
true
false
true
```

The ? operator

Sort of if-then-else that given a conditional expression chooses between two expressions

```
condition ? expression1 : expression2
```

If condition is true, expression1 is evaluated, otherwise expression2 is evaluated

The ?-expression assumes the result of the evaluated expression

```
IO.println(expression ? "It rains" : "It doesn’t rain")
```

Equivalent ternary operator in Python

```
expression1 if condition else expression2
```

Additional content – Operator overloading

In Java, as in many other languages some operators are overloaded, i.e., they have different implementations depending on the type of the operators

In the following expressions, you are using three distinct '+' operators

```
1 + 1
1.0 + 3.14
"Hi, " + "Java!"
```

One for integral numbers, one for double precision floating point numbers, and a last one for Strings

In practice, all the arithmetic operators are overloaded, because of the way type promotion is implemented, e.g., you can verify that the result of `1f/3` *is different than* `1d/3`

Additional content – Short-circuiting

Operator	Name
&&	*conditional-and*
||	*conditional-or*
&	*and*
|	*or*
!	*not*

Conditional-and and conditional-or are the most used and/or operators, but since they are short-circuiting, they are more a control flow operator than logical operator

For example, the expression `a || b` *is executed as: if* `a` *is true then return true, else return* `b`*, while* `a | b` *is executed as evaluating the or value of a and b*

The first version is said to shot-circuit because it skips the evaluation of b when a is true. E.g., this can make a difference if the evaluation of b involves a function call that has some side effect

Additional content - Bit level operators

Operator	Name
&	*and*
|	*or*
^	*xor*
~	*not*
<<	*left shift*
>>	*signed right shift*
>>>	*unsigned right shift*

They operates on integral values as sequences of bits, and they return integral values too

Integral types in Java are mostly signed, except for the char type

By mixing arithmetic and bit level operators you can get unexpected results, especially when negative numbers are involved

Bits.java

```
void main() {
  int x = 0b00000000000000000000000000010110;
  int y = 0b00000000000000000000000000110011;

  IO.println(x & y);       // 00000000000000000000000000010010
  IO.println(x | y);       // 00000000000000000000000000110111
  IO.println(x ^ y);       // 00000000000000000000000000100101
  IO.println(~x);          // 11111111111111111111111111101001

  x = 0b00000000000000000000000000001001; // 9
  IO.println(x >> 3);      // 00000000000000000000000000000001
  IO.println(x >>>3);      // 00000000000000000000000000000001

  x = -9;                  // 11111111111111111111111111110111
  IO.println(x >> 3);      // 11111111111111111111111111111110
  IO.println(x >>>3);      // 00011111111111111111111111111110
}
```

Flow control

Flow control statements

Selection

- *if-then-else*
- *switch*

Iteration

- *while*
- *do-while*
- *for*
- *for-each*

Jump

- *continue*
- *break*
- *return*

If-then-else

IfThenElse.java

```
void main() {
    char c = 'x';
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
        IO.println("letter: " + c);
    else if (c >= '0' && c <= '9')
        IO.println("digit: " + c);
    else {
        IO.println("the character is: " + c);
        IO.println("it is not a letter nor a digit");
    }
}
```

```
$ java IfThenElse
letter: x
```

Switch

Switch.java

```
void main() {
    boolean leapYear = true;
    int days = 0;
    for (int month = 1; month <= 12; month++) {
        switch (month) {
            case 2:
                days += leapYear ? 29 : 28;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                days += 30;
                break;
            default:
                days += 31;
                break;
        }
    }
    IO.println(days);
}
```

"Thirty days has September,
April, June, and November,
All the rest have thirty-one,
Save February at twenty-eight,
But leap year, coming once in four,
February then has one day more."

```
$ java Switch
366
```

Grouped switch

GroupedSwitch.java

```
void main() {
    boolean leapYear = true;
    int days = 0;

    for (int month = 1; month <= 12; month++) {
        switch (month) {
            case 2:
                days += leapYear ? 29 : 28;
                break;
            case 4, 6, 9, 11:
                days += 30;
                break;
            default:
                days += 31;
                break;
        }
    }
    IO.println(days);
}
```

By using the commas ‘,’ you can group together multiple cases

```
$ java GroupedSwitch
366
```

Enhanced switch

EnhancedSwitch.java

```
void main() {

    boolean leapYear = true;
    int days = 0;

    for (int month = 1; month <= 12; month++) {
        switch (month) {
            case 2 -> days += leapYear ? 29 : 28;
            case 4, 6, 9, 11 -> days += 30;
            default -> days += 31;
        }
    }
    IO.println(days);
}
```

```
$ java EnhancedSwitch
366
```

The traditional colon ':' and break can be replaced by the arrow '->'

Switch expression

SwitchExpression.java

```
void main() {
    boolean leapYear = true;
    int days = 0;

    for (int month = 1; month <= 12; month++) {
        days += switch (month) {
            case 2:
                yield leapYear ? 29 : 28;
            case 4, 6, 9, 11:
                yield 30;
            default:
                yield 31;
        };
    }
    IO.println(days);
}
```

```
$ java SwitchExpression
366
```

The switch expression construct directly returns a value by using the yield keyword

Expressions terminate with a semicolon

Not exactly a flow control statement!

Enhanced switch expression

EnhancedSwitchExpression.java

```
void main() {
    boolean leapYear = true;
    int days = 0;

    for (int month = 1; month <= 12; month++) {
        days += switch (month) {
            case 2 -> leapYear ? 29 : 28;
            case 4, 6, 9, 11 -> 30;
            default -> 31;
        };
    }
    IO.println(days);
}
```

```
$ java EnhancedSwitchExpression
366
```

The colon ‘:’ and yield can be replaced by the arrow ‘->’ providing a very compact syntax

Summary of switch

	Traditional ':'	Enhanced '->'
Switch	*traditional use*	*no need to break the fall-through*
Expression switch	*use yield to return the case value*	*no need to break the fall-through* *no need to yield to return the case value*

The case-expression must evaluate to byte, short, char, int, enum, or String

Cases can be grouped together

Add a semicolon ';' after the switch expression

While

While.java

```
void main() {
    final double initialValue = 2.34;
    final double step = 0.11;
    final double limit = 4.69;

    double var = initialValue;
    int counter = 0;
    while (var < limit) {
        var += step;
        counter++;
    }
    IO.println("Incremented " + counter + " times");
}
```

```
$ java While
Incremented 22 times
```

Do-while

DoWhile.java

```
void main() {
    final double initialValue = 2.34;
    final double step = 0.11;
    final double limit = 4.69;
    double var = initialValue;

    int counter = 0;
    do {
        var += step;
        counter++;
    } while (var < limit);
    IO.println("Incremented " + counter + " times");
}
```

```
$ java DoWhile
Incremented 22 times
```

For

For.java

```
void main() {
    final double initialValue = 2.34;
    final double step = 0.11;
    final double limit = 4.69;

    int counter = 0;
    for (double var = initialValue; var < limit; var += step)
        counter++;
    IO.println("Incremented " + counter + " times");
}
```

```
$ java For
Incremented 22 times
```

For-each

ForEach.java

```
void main() {
    int[] a = {2, 4, 3, 1};

    int sum = 0;
    for (int x : a) {
        sum += x;
    }

    double[] d = new double[sum];
    for (int i = 0; i < d.length; i++) {
        d[i] = 1.0 / (i + 1);
    }

    for (var f : d) {
        IO.println(f);
    }
}
```

```
$ java ForEach
1.0
0.5
0.3333333333333333
0.25
0.2
0.16666666666666666
0.14285714285714285
0.125
0.1111111111111111
0.1
```

Continue

Continue.java

```
void main() {

    for (int counter = 0; counter < 10; counter++) {
        // start a new iteration if the counter is odd
        if (counter % 2 == 1) continue;
        IO.println(counter);
    }
    IO.println("done.");
}
```

```
$ java Continue
0
2
4
6
8
done.
```

Break

Break.java

```
void main() {
    for (int counter = 0; counter < 10; counter++) {
        // start a new iteration if the counter is odd
        if (counter % 2 == 1) continue;
        // exit the loop if the counter is equal to 8
        if (counter == 8) break;
        IO.println(counter);
    }
    IO.println("done.");
}
```

```
$ java Break
0
2
4
6
done.
```

Return (exit from method)

Return.java

```
void main() {
    for (int counter = 0; counter < 10; counter++) {
        // start a new iteration if the counter is odd
        if (counter % 2 == 1) continue;
        // exit the method if the counter is equal to 8
        if (counter == 8) return;
        IO.println(counter);
    }
    IO.println("done.");
}
```

```
$ java Return
0
2
4
6
```

Return a value

Sum.java

```
int sum(int[] array) {
    int s = 0;
    for (int i : array) {
        s = s + i;
    }
    return s;
}

void main(String[] args) {
    int[] a = {71, 4, 31, 53};
    IO.println(sum(a));
}
```

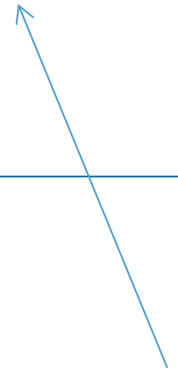
```
$ java Sum
159
```

Exercises

Exercise 1

Implement a Hello program to say hello to multiple people

```
$ java Hello Paolo Dario
Hello Paolo and Dario!

$ java Hello Francesco Joe Arthur
Hello Francesco, Joe, and Arthur!

$ java Hello
Hello everybody!
```



Note the usage of the Oxford comma

Exercise 2

Implement a Calculator program to perform arithmetic operations.

```
$ java Calculator 6 + 4.1
10.1
$ java Calculator 3.6 / -2
-1.8
$ java Calculator 8.5 * 9
76.5
$ java Calculator -3.14
-3.14
```

I let you discover how to convert strings to numbers

Enhance the calculator so that it can handle concatenated operations

```
$ java Calculator 6 + 4.1 * 3
10.1
30.3
$ java Calculator 3.6 / 2 + -0.3 / .5
1.8
1.5
3
```

References

References

Variables, data types, operators, and flow control

- *https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html*

Self assessment

Quiz 1: Data Types, Variables, and Arrays

1. **Which of the following is a primitive data type in Java?**
   A. `String`
   B. `Array`
   C. `int`
   D. `Object`

2. **What are the contents of the array a after the following line is executed: `int[] a = new int[2];`?**
   A. The array contains two `null` values
   B. The code will not compile because the values are not initialized
   C. The array contains `0` and `0`
   D. The array contains uninitialized or random memory values

3. **In which of these situations is an explicit type cast required?**
   A. Assigning an `int` value to a `long` variable
   B. Assigning a `float` value to a `double` variable
   C. Assigning a `char` literal like `'a'` to an `int` variable
   D. Assigning a `double` value to a `float` variable

Quiz 2: Operators and Expressions

1. **What is the output of the following code snippet?**

```
IO.println(1 + 9 + " vs " + 1 + 9);
```

   A. *10 vs 19*
   B. *19 vs 10*
   C. *19 vs 19*
   D. *10 vs 10*

2. **What is the result value of the following code snippet?**

```
9 / 4
```

   A. *2.25*
   B. *2*
   C. *2.0*
   D. *The code will cause a compilation error*

Quiz 2: Operators and Expressions

3. **What is printed to the console by the following code?**

```
int a = 12;
IO.print(a++);
IO.print(a);
```

A. *1213*

B. *1313*

C. *1212*

D. *1312*

Quiz 2: Operators and Expressions

4. **What is the result of compiling and running the following code snippet?**

```
byte b1 = 10;
byte b2 = 5;
byte b3 = b1 * b2;
IO.println(b3)
```

A. It will print 50.

B. It will cause a compilation error.

C. It will print 0.

D. It will compile but throw an exception at runtime.

Quiz 3: Flow Control

1. **In a traditional switch statement, what happens if you omit the break keyword in a case?**
   A. *The program will produce a syntax error*
   B. *Execution "falls through" to the statements in the next case*
   C. *The loop automatically exits*
   D. *Only the default case will be executed*
2. **Which looping statement in Java is guaranteed to execute its body at least one time?**
   A. *for*
   B. *while*
   C. *for-each*
   D. *do-while*
3. **What is the primary advantage of using the enhanced switch with an arrow (->)?**
   A. *It provides a more compact syntax that replaces both the colon (:) and the break statement*
   B. *It can handle floating-point numbers in its cases*
   C. *It forces all cases to have a break statement*
   D. *It is the only way to group multiple cases together*

Correct answers

Quiz 1

1. *C*
2. *C*
3. *D*

Quiz 2

1. *A*
2. *B*
3. *A*
4. *B*

Quiz 3

1. *B*
2. *D*
3. *A*

Thank you!

esteco.com