3TC36: Object-Oriented Programming in Java

Data Types, Classes, Objects, and Constructors

Dominique Blouin
dominique.blouin@telecom-paris.fr
Le February 17, 2026

Learning objectives

- Scalar types and their operators
- Classes, objects, and constructors
- Reference types
- Class and instance variables
- About Java...

Basic types (scalars) in Java

- **Variable:** a named memory location where data is stored.
- Every variable or data handled by Java has a **type**.
  - Like in C programming language.
- When a variable is declared, the programmer declares its type.
  - The variable can only contain values of that type.
- Java is a **strongly typed** language.
  - In contrast to Python or JavaScript, for example.

The `int` type

- A named attribute or variable `var` of type `int` is declared as:
  `int var;`
- An `int` attribute is initialized by default to `0`.
- An `int` variable can be explicitly initialized using the **assignment operator**:
  `int var = 0;`
- It is considered very good practice to **explicitly** do the initialization.
- Integer values are represented as signed 32-bit integers, ranging from `−2 147 483 648` to `2 147 483 647`.

Other basic types

Type	Meaning	Possible values
`char`	Character	Unicode character set: `'a'`, `'b'`,...
`byte`	Very short integer	`-128` to `127`
`short`	Short integer	$-32\ 768$ to $32\ 767$
`int`	Integer	$-2\ 147\ 483\ 648$ to $2\ 147\ 483\ 647$
`long`	Long integer	$-9\ 223\ 372\ 036\ 854\ 775\ 808$ to $9\ 223\ 372\ 036\ 854\ 775\ 807$
`float`	Floating-point (real)	$1.4 * 10^{-45}$ to $3.4 * 10^{38}$
`double`	Double floating-point	$4.9 * 10^{-324}$ to $1.7 * 10^{308}$
`boolean`	Boolean	Default initialization: `true` or `false`

- Numeric type variables are initialized to `0`.
- `boolean` type variables are initialized to `false`.

Arithmetic operators

- Java includes the main arithmetic operations to construct expressions:
  - `int xVar = 20;` // Declaration and assignment
  - Multiplication: `int yVar = xVar * xVar;`
  - Addition: `int zVar = xVar + yVar;`
  - Division: `xVar = zVar / xVar;`
  - Modulo: `yVar = zVar % xVar;` // remainder of division
  - Parentheses: `int uVar = (zVar * (xVar + yVar)) - xVar;`
- Arithmetic operators also work with `float` and `double` (except %).

Comparison operators

- The comparison operators allow you to compare numbers. They return a `boolean` value.
- Examples:
  - `boolean myBool0 = (xVar <= yVar);`
  - `boolean myBool1 = (xVar < yVar);`
  - `boolean myBool2 = (xVar == yVar);` // equality test
  - `boolean myBool3 = (xVar != yVar);`
  - `boolean myBool4 = (xVar > yVar);`
  - `boolean myBool5 = (xVar >= yVar);`

Logical operators

- Logical operators work on `boolean` values.
- Examples:
  - AND (`&&`): `boolean myBool0 = (xVar < yVar) && (xVar > zVar);`
  - OR (`||`): `boolean myBool1 = (xVar < yVar) || (xVar > zVar);`
  - NOT (`!`): `boolean myBool2 = !myBool1;`

(more...)

The Class type

- Suppose we write a program dealing with geometric concepts in a plane. We would create a `Point` class to describe a point in the plane:

```
class Point {
    int xCoord;
    int yCoord;
}
```

- A point is characterized by its **two coordinates** in the plane.
  - Thus, we declare two attributes in the class: `xCoord` and `yCoord`.
  - They are of type `int`, meaning they can hold integer values (positive, negative, or zero).

The reference type

- If a class is defined (*e.g.,* the `Point` class in our example), it is possible to create an **object** of that class.
- To do so, we need to declare a **variable** that can hold a **reference** to an **object** of the `Point` class.
- This is declared as:

```
Point myPoint;
```

- A variable is a named and typed memory location:
  - Here, the name is `myPoint`.
  - It can only hold values of a certain **type**.
  - This type is a **reference** to an **object** of the `Point` class.
  - This is the same principle as a `struct` type variable in C.

Initialization of reference type variable

- A reference type variable must be initialized, possibly with a very specific value, which is written as `null`.
- This variable will not reference any object.
- Any attempt to send a message using an uninitialized reference variable (with value `null`) will result in a runtime error and program termination if the error is not handled.
  - *We will cover this later in exception handling...*

Creating an object

- The variable named `myPoint` is a **reference** type variable for an **object** of class `Point`.
- We can create an object of the `Point` class using the `new` keyword:

```
myPoint = new Point();
```

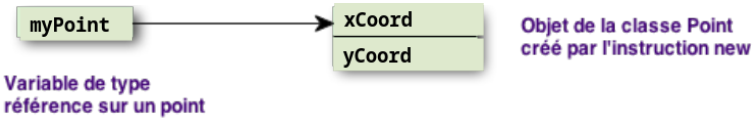
- The symbol `=` is called the **assignment operator**:
  - The variable on the left of `=` receives the value on the right of it.
  - Here, the value is a **reference** to an **object** of the `Point` class that is created by the `new` keyword.
- A common error among beginners is confusing the **assignment** operator (`=`) with the **equality** operator ($=$) in mathematics.
- In programming languages, the equality operator is written as `==`.

Creating an object (continued)

- We can combine the variable declaration and object creation:

```
Point myPoint = new Point();
```

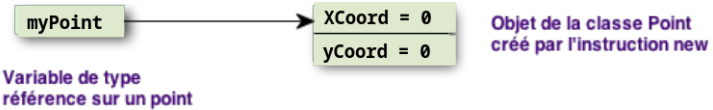
- In both cases, the result is the same.
- Illustration of what happens in memory:



The reference is represented by an arrow pointing to the referenced object.
Note that `myPoint` does not contain the point, but a **reference** to the **memory** storing the point's data (its coordinates).

Initializing object attributes

- What are the values of the `xCoord` and `yCoord` attributes after the object is created?



- The Java specification tells us that data (variables and attributes) of type `int` are initialized with the value `0`.

Attention: This is true in Java but not in all programming languages!

This highlights the importance of **explicitly** initializing variables.

Constructors

- We have seen that Java initializes attributes with default values.
  For example, an attribute of type `int` will be initialized to `0`.
- However, you may want to initialize attributes with values other than the default ones.
- To do this, you must define a **specific constructor**.

Constructors (continued)

- A **constructor** is a **method** (function in a class) that has the same name as the class.
- It may have **parameters** but no return type.
- Example:

```
Point(int xCoordInit, int yCoordInit) {
    xCoord = xCoordInit;
    yCoord = yCoordInit;
}
```

- You can then create a `Point` using the constructor:

```
Point myPoint = new Point(23, 67);
```

Constructors (continued)

- It is possible to define multiple constructors in a class.
- This allows for different initializations of attributes.
- Example of a constructor for polar coordinates:

```
Point(double rho, double theta) {
    xCoord = (int)(rho * Math.cos(theta));
    yCoord = (int)(rho * Math.sin(theta));
}
```

- You can then create a `Point` using this constructor:

```
Point myPoint = new Point(10.12, 1.34); // angle in radians
```

Constructors (continued)

- A class can have as many constructors as needed.
  - Each constructor provides a different initialization.
- Each constructor must have **different parameters**:
  - Whether in **number** or in **type** of parameters.
- The constructor used when creating an object is determined by the types of the parameters passed in the `new` statement:
  - `new Point(23, 67)`: a constructor with two `int` parameters.
  - `new Point(10.62, 1.67)`: a constructor with two `double` parameters.
- This is why different constructors must be declared with **different parameters**.
- If one or more constructors exist in a class, object creation must use one of those constructors.
- Otherwise, a **default constructor** (with no parameters) will be automatically provided, although it may not be visible in the code.

Constructor call and the `this` keyword

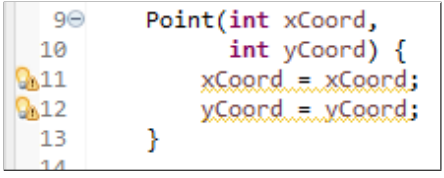
- It is possible for one constructor to call another constructor.
- This is done using the `this` keyword followed by parameters in parentheses.
- The `this` keyword specifies that we are referring to an element of the class where the code is written (*i.e.,* the constructor to call).

```
Point(double rho, double theta) {
    this((int)(rho * Math.cos(theta)),
         (int)(rho * Math.sin(theta)));
}
```

- This constructor call must be the first instruction in the calling constructor.

Constructor call and the `this` keyword

- The `this` keyword can also be used to specify that we are referring to a class **attribute**, thus avoiding **naming ambiguity** with parameters.




```
Point(int xCoord, int yCoord) {
    this.xCoord = xCoord;
    this.yCoord = yCoord;
}
```

Sending messages between objects: Methods

- For an object to receive a message, it must declare a **method** associated with the desired message type.
- The header of a method consists of:
  - The visibility keyword (to be covered later)
  - The return type, or `void` if there is no return value.
  - The method name.
  - The opening parenthesis `(`.
  - The list of parameters with their names and types.
  - The closing parenthesis `)`.
- The header is also called the method **signature**.

Sending messages between objects: Methods

- Example: displaying the coordinates of a point:

```
class Point {
        int xCoord;
        int yCoord;

        void writeCoordinates() {
            System.out.print("x = ");
            System.out.println(xCoord);
            System.out.print("y = ");
            System.out.println(yCoord);
        }
}
```

- To access an attribute or method of an object, use the dot (.) operator:

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Another solution

- Directly accessing the attributes of the class:

```
Point myPoint = new Point(10, 10);
System.out.print("x = ");
System.out.println(myPoint.xCoord);
System.out.print("y = ");
System.out.println(myPoint.yCoord);
```

- Which is the better solution: direct access to attributes or using the `writeCoordinates()` method?

Delegation and non-intrusion

- A good practice in OOP is to avoid acting from the outside on the state of an object (**non-intrusion**).
- As much as possible, avoid operations that read and write attributes.
- Instead, ask the `Point` object to display its coordinates (**delegation** principle).
- The best way:

```
Point myPoint = new Point(10, 10);
myPoint.writeCoordinates();
```

Class and instance variables

- Each object of a class has its own attribute values.
- Therefore, objects of the same class have no link to each other, except for their class.
- Everything declared in the class, such as methods, is **shared** by all objects of the class.
- However, there may be cases where it is preferable for instances of the same class to **share a value**.

Example: Counting total points

```
class Point {
    static int numberOfPoints = 0; // CLASS variable
    int xCoord; // INSTANCE variable
    int yCoord; // INSTANCE variable

    Point(int xCoordInit, int yCoordInit) {
        xCoord = xCoordInit;
        yCoord = yCoordInit;
        numberOfPoints = numberOfPoints + 1;
    }

    void writeNumberOfPoints() {
        System.out.print("number of points = ");
        System.out.print(numberOfPoints);
    }
}
```

Example: What will be displayed in the console?

```
Point point1 = new Point(0, 4);
Point point2 = new Point(5, 9);
Point point3 = new Point(9, 0);

point1.writeNumberOfPoints();
point2.writeNumberOfPoints();
```

Example: What will be displayed in the console?

```
Point point1 = new Point(0, 4);
Point point2 = new Point(5, 9);
Point point3 = new Point(9, 0);

point1.writeNumberOfPoints();
point2.writeNumberOfPoints();
```

- The `numberOfPoints` attribute is declared with the `static` keyword.
- It's a **class** variable: its value is **held by the class**.
- It is shared by **all objects** (**instances**) of the class.
- Thus, the same value (**3**) of the number of points will be displayed for both `point1` and `point2` instances.
- As we will see later, in practice, class variables are **rarely used**.

Class method

- A method can also be marked with the `static` keyword.

```
class Point {
    static int numberOfPoints = 0;
    int xCoord;
    int yCoord;

    Point(int xCoordInit, int yCoordInit) {
        xCoord = xCoordInit;
        yCoord = yCoordInit;
        numberOfPoints = numberOfPoints + 1;
    }

    static void writeNumberOfPoints() {
        // you can ONLY access static variables
        System.out.print("number of points = ");
        System.out.print(numberOfPoints);
    }
```

Calling a class method

```
Point point1 = new Point(0, 4);
Point point2 = new Point(5, 9);
Point point3 = new Point(9, 0);

point1.writeNumberOfPoints();
Point.writeNumberOfPoints();
```

- A class method is executed by the class, not by the object.
- Therefore, there is no need to know a **specific instance** to call the method.
- Also, a class method only has access to **class variables**.
- Accessing an instance variable inside a class method will result in a **compilation error**.

Running a Java program

- A Java program must always contain at least one class with a **class method** that will be called at the beginning of the program:

```
public static void main(String[] args) {
    Point myPoint = new Point(10, 10);
    myPoint.writeCoordinates();
}
```

- `void main(String[] args)` : this signature is required to indicate the entry point of the program.
- `String[] args` : the `String` class will be introduced in the next lesson...

Best naming practices in code

- English is the language of computing. Therefore, use English to name your programs, write comments, etc.
- Using non-English letters can lead to issues.
  - Avoid accented characters, for example.
- Choose **descriptive** names (avoid one or two-letter names) and **capitalize** them.

```
class Point {
    int xCoord;
    int yCoord;

    void writeCoordinates() {
        //...
    }
}
```

Other naming rules

- The name of a class starts with an **uppercase letter**.
- The name of variables, attributes, and methods starts with a **lowercase letter**.

```
class Point {
    int xCoord;
    int yCoord;

    void writeCoordinates() {
        //...
     }
}
```

- This makes it easier to identify these elements.

About Java

- Java was created in 1991 by Sun Microsystems. The goal was to program small devices like remote controls.
- Java was later applied to programming applications in web browsers: *applets*.
- Java is named after the favourite drink of its designers.
- Sun Microsystems was acquired by Oracle.

About Java

- Java is a general-purpose programming language used in various industries.
- The JDK (Java Development Kit) offers a large number of libraries that form a toolkit for software development.
  - In this course, we will learn to use this toolkit.
- Developer communities provide solutions (sets of classes) on the web, which are more or less elaborate, generic, tested, and documented, to help develop applications at a large scale.
  - Example: Eclipse and Apache foundations.
- This is one of Java's great strengths.

Java portability

- A key feature of Java is its **portability**:
  - The same application can run on different computers, abstracting hardware and operating system specifics.
- In a typical compiler architecture, the compiler produces a bytecode file, **executable**, that the CPU can execute.
  - This executable is **dependent** on the processor and operating system.
- Java is portable because it is implemented via a **virtual machine**.

Java portability

- The virtual machine hides the details of the processor and operating system by replacing (or emulating) them with a consistent execution environment across all computers.
- A virtual machine is an imaginary computer, defined by documentation, with its own processor, memory, OS (operating system), etc.
- The Java compiler produces an executable but for the **virtual machine**.
- This virtual machine is called the **JVM** (Java Virtual Machine).

Java portability

- A Java executable is not run directly by the computer.
- A program that emulates the JVM executes the bytecode file.
- Programs emulating the JVM exist for all computers and operating systems.
- SUN's motto: **Compile once, run everywhere.**

Ref: "Head First ..." The way Java works