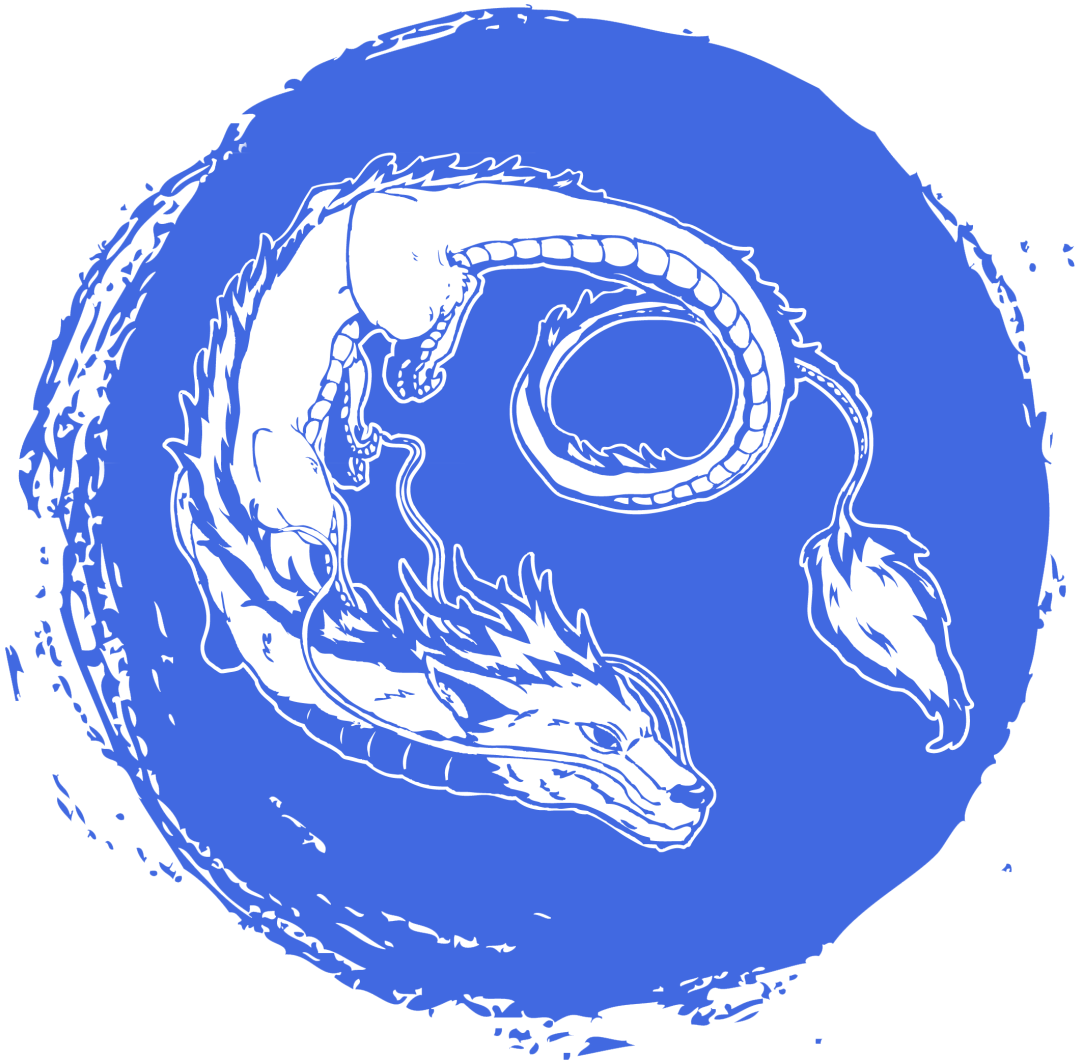
Java Workshop — Tutorial D2

version **#v1.2.0**



Yet Another Kind of Assistants 2022 <yaka@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Final

The meaning of the `final` keyword provided by *Java* depends on where it is located:

- In a **class** declaration;
- In a **method** declaration;
- In an **attribute** declaration;
- In a **variable** declaration.

1.1 Final class

When a class is declared as `final`, it means that no other class can inherit from it. If you think of inheritance as a tree, then a `final` class is a leaf.

```
public class Main {
    public static final class FinalClass {
    }

    public static class BrokenClass extends FinalClass {
        // Error: Cannot inherit from the final class FinalClass
    }
}
```

1.2 Final method

When a method is declared as `final`, it means that it cannot be overridden.

```
public class Main {
    public static class ParentClass {
        public final void finalMethod() {
            System.out.println("Final method");
        }
    }

    public static class ChildClass extends ParentClass {
        @Override
        public void finalMethod() {
            // Error: Cannot override final method finalMethod from ParentClass
            System.err.println("Attempting to override final method");
        }
    }
}
```

1.3 Final attribute

When a class attribute is declared as `final`, it means that its value is evaluated at instantiation, and that from this point it cannot be reassigned.

```
public class Main {
    public static class MyClass {
        public final int myInt = 0;

        public void myMethod() {
            myInt++; // Error: Cannot assign a value to final variable myInt
        }
    }
}
```

This point can be quite misleading: you could assume that it behaves like *C* or *C++*'s `const` keyword. Indeed, for *primitive types*, there is no great difference between `final` and `const`.

However, when we start playing with *objects*, it gets completely different: where `const` makes the object **read-only**, `final` just **prevents it from being reassigned**.

```
public static class MyClass {
    private final List<String> myList = new ArrayList<>();

    public void myMethod() {
        // Even though myList is final, we can still add elements to it...
        myList.add("Hello");
        myList.add("world!");

        // Set its elements...
        myList.set(1, "Hello yaka!");

        // Or even remove
        myList.remove(0);

        // However, you cannot reassign it.
        myList = new ArrayList<>(); // Error: Cannot assign a value to final variable myList
    }
}
```

Going further...

The correct analogy with *C++* would rather be with `const&`.

If you want one of your classes to be read-only, you can make it **immutable**! *Java 9* also provided methods to create *immutable collections*, such as List.of.

```
public class Main {
    public static void main(String[] args) {
        final var myImmutableList = List.of("Can't", "touch", "this");

        // UnsupportedOperationException (at runtime): cannot append to an immutable List
        myImmutableList.add("Attempting to add an element");
```

(continues on next page)

(continued from previous page)

```
        // UnsupportedOperationException (at runtime): cannot set an element of an immutable␣
↪List
        myImmutableList.set(0, "Attempting to set an element");
    }
}
```

However, this can only bring us as far as creating a list of `final` *references*. This means that it is possible to do this:

```
public class Main {
    public static void main(String[] args) {
        final var myList1 = new ArrayList<Integer>();
        final var myList2 = new ArrayList<Integer>();
        for (int i = 0; i < 5; i++) {
            myList1.add(2 * i);
            myList2.add(2 * i + 1);
        }
        final var myNotSoImmutableList = List.of(myList1, myList2);
        System.out.println(myNotSoImmutableList.toString());
        myList1.add(42);
        myList2.add(47);
        System.out.println(myNotSoImmutableList.toString());
    }
}
```

Output:

```
[[0, 2, 4, 6, 8], [1, 3, 5, 7, 9]]
[[0, 2, 4, 6, 8, 42], [1, 3, 5, 7, 9, 47]]
```

1.4 Blank `final`

Sometimes, you cannot know right away the value your `final` variable will have: it will depend on the arguments given to the constructor of your class. You have a solution for this: the blank final.

Evaluating the value of an attribute at instantiation means you can directly inline its value at field declaration (as you have seen above), but also assign it in the constructor. That is what a blank final is: a final attribute which value is assigned in the constructor.

Beware: if you declare a blank final, its value must still be set at instantiation. You cannot assign it in another method.

```
public static class MyClass {
    private final int inlinedFinal = 0; // Value is inlined at instantiation: good.

    // Error: variable blankFinalNeverSet might not have been assigned.
    private final int blankFinalNeverSet;
    private final int blankFinalSetInAMethod;
    private final int blankFinalTriedToBeSmart;
```

(continues on next page)

(continued from previous page)

```
    private final int goodBlankFinal;

    public MyClass() {
        usedOnlyInConstructor();
        goodBlankFinal = 4;
    }

    public void myMethod() {
        // Error: You cannot set a blank final in any other method than the constructor.
        blankFinalSetInAMethod = 2;
    }

    private void usedOnlyInConstructor() {
        blankFinalTriedToBeSmart = 3; // Well tried...
    }
}
```

1.5 Final variable

A `final` variable is pretty much like a `final` attribute, but lives in the scope of a method: it cannot be reassigned. You can also declare it without a value, if for instance its value depends on an `if ... else` or `switch ... case` statement.

```
public class Main {
    public static void main(String[] args) {
        final String assignedAtDeclaration = "Our only hope is ";
        final String blankFinal;

        if (args.length != 1) {
            blankFinal = "to recode everything!";
        } else {
            blankFinal = args[0];
        }

        System.out.println(assignedAtDeclaration + blankFinal);
    }
}
```

Likewise, a `final` argument can be given to a method, in order to check that it is never reassigned. The `final` keyword is not considered part of the method signature: therefore, it does not create issues for overriding.

2 Time

Since *Java 8*, the language provides the `java.time` library to represent dates. This API allows us to represent the date and time depending on the precision and the needed quantity of information.

2.1 Temporal

The `Temporal` package contains several interfaces that are at the root of the `java.time` API, and are implemented by the classes that will be presented in this section.

The most notable interfaces of this package are:

- `Temporal`: represents a potentially zoned date, time, point in time. To name only two, `LocalDateTime` and `Instant` implement this interface.
- `TemporalAmount`: represents an amount of time. `Period` and `Duration` implement this interface.
- `TemporalUnit`: represents a unit of time such as seconds, months, centuries... The enum `ChronoUnit` implements this interface and gives access to a wide variety of units of time.
- `TemporalField`: represents a field of a date, such as the day of the week or the month of the year. The `ChronoField` enum provides a set of those possible fields.

2.2 Dates

First of all, *Java* provides enums to represent days and months: `DayOfWeek` and `Month`. You can get their textual representation by using the method `getDisplayName`.

```
DayOfWeek dayOfWeek = DayOfWeek.FRIDAY;
dayOfWeek.getDisplayName(TextStyle.FULL, Locale.getDefault()); // Friday

Month month = Month.JUNE;
month.getDisplayName(TextStyle.SHORT, Locale.FRANCE); // juin
```

To represent a date without the notion of time nor timezone, we use the `LocalDate` class. It is instantiated with the static method `of(int year, Month month, int dayOfMonth)` and only contains those three values. You can get the `DayOfWeek` of a `LocalDate` by using the method `getDayOfWeek`.

```
LocalDate localDate = LocalDate.of(2006, 7, 9); // 9 July 2006
localDate.getDayOfWeek(); // Sunday
```

Some classes also represent parts of a date, namely `YearMonth`, `MonthDay` and `Year`: they are useful when you do not need a heavy, complete date. They also provide some useful methods such as `Year.isLeap()` to check if a year is a leap year.

```
LocalDate localDate = LocalDate.of(2006, 7, 9); // 9 July 2006

YearMonth.from(localDate); // 2006-07
MonthDay.from(localDate); // --07-09
Year.from(localDate); // 2006
```

2.3 Time

To represent the time of a day, without any notion of date nor timezone, we use the class `LocalTime`. Although it is mostly used to represent time as it could be seen on a clock, the time is actually stored with a nanosecond precision. You can obtain the current time based on the system clock with `LocalTime.now()`.

2.4 DateTime

2.4.1 LocalDateTime

One of the main classes we use to represent date and time is `LocalDateTime`. It is based on the ISO 8601, and does not contain time zone information. This is the combination of a `LocalDate` and a `LocalTime`. To instantiate a `LocalDateTime` for a known date, you can use the static method `of`, specifying at least the year, month, day, hour and minute (second and nanosecond are defaulted to 0). The class provides several useful methods:

```
// Static method that gives the current LocalDateTime
LocalDateTime currentDateTime = LocalDateTime.now();

// Static method to instantiate a LocalDateTime for a known date
LocalDateTime localDateTime = LocalDateTime.of(2021, 4, 7, 11, 42);

// Getters on each field
localDateTime.getHour(); // 11
localDateTime.getMonth(); // APRIL
localDateTime.getDayOfWeek(); // WEDNESDAY
localDateTime.getDayOfMonth(); // 7
localDateTime.getDayOfYear(); // 97

// Methods to compare LocalDateTime objects
localDateTime.isEqual(currentDateTime); // false

// Methods for adding or subtracting the specified quantity of a field
localDateTime.plusDays(2); // 2021-04-09T11:42
localDateTime.minusMinutes(72); // 2021-04-07T10:30

// Methods to modify a specific field
localDateTime.withHour(23); // 2021-04-07T23:42
localDateTime.withMonth(1); // 2021-01-07T11:42
```

2.4.2 Time zones

To represent a time zone, the `java.time` library offers two classes: `ZoneId` and `ZoneOffset`. `ZoneID` represents a time zone identifier, which can be implemented either as an `Area/City` such as `Europe/Paris`, or with a fixed offset of UTC/GMT, such as `+02:00`. `ZoneOffset` extends `ZoneId` and defines the fixed offset of the current time-zone with UTC/GMT.

```
ZoneId zoneId = ZoneId.of("Europe/Paris"); // Europe/Paris

// These two classes can be combined with a LocalDateTime
ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId); // 2021-04-
↪07T11:42+02:00[Europe/Paris]
```

2.4.3 Parsing and formatting

`LocalDate`, `LocalTime`, and more generally `LocalDateTime`, provide a `parse()` static method to parse a date and time from a string and get an object. It takes the string to parse as parameter, and can be overloaded with a second argument of type `DateTimeFormatter`. A `DateTimeFormatter` is a class that represents the formatting of a date, that can either be predefined, or user-defined through a format string. Details on how to describe patterns are listed in the class' javadoc. If no `DateTimeFormatter` is provided to `parse()`, it will expect a date in ISO standard format. Those same classes provide a `format()` method, that takes a `DateTimeFormatter` and returns a string with date and/or time expressed in the given format.

```
LocalDateTime localDateTime = LocalDateTime.parse("2021-04-07T11:42");

LocalDate date = LocalDate.parse("07-04-2021", DateTimeFormatter.ofPattern("MM-dd-yyyy"));

date.format(DateTimeFormatter.ofPattern("yyyy-dd-MM"); // 2021-04-07
```

2.5 Instant

The `Instant` class is another class widely used to represent time. Rather than representing a date and time with days, months, seconds or hours, it represents a point in time expressed in nanoseconds relatively to the EPOCH (the first of January 1970, at midnight). It does not mean that the minimal date that can be expressed is the EPOCH, a date anterior to it will be stored with a negative nanoseconds count.

Similarly to `LocalDateTime`, `Instant` provides some useful functions to manipulate it, such as `now()` to get the current `Instant`, the comparisons (`isAfter()`, `isBefore()`) and the `plus` and `minus` methods. It is also possible to create an `OffsetDateTime` and a `ZonedDateTime` from an `Instant` through the methods `atOffset()` and `atZone()`. Also, `Instant` provides a `parse()` method, but does not offer the possibility to specify a format: it must be formatted as an ISO instant. Also, there is no `format` method but `toString()` can be used to get a textual representation of the `Instant`.

2.6 Period and Duration

To express an amount of time, the `java.time` library provides two classes: `Duration` and `Period`. The former uses nanoseconds to measure the amount of time with precision, and is commonly used with `Instant`, while the latter expresses the amount of time in terms of date, with years, months and days, and is better used with `LocalDate`. Both classes can be instantiated using the static methods `of`, specifying the amount of time in the desired unit. They also offer the `plus` and `minus` operations, as well as a product with a scalar with the `multipliedBy()` method. You can also obtain a `Duration` by

calling the static method `between()` with any two `Temporal`. You can also use between with `Period`, but you can only compute the difference between two `LocalDate`.

There is a third way to express amounts of time, using the `ChronoUnit` `between()` method. It allows to express an interval of time in a single given unit. For instance, the difference between 12 January 2017 and 12 January 2018 would give a `Period` with the field year set to 1, month set to 0 and day set to 0 too. The same difference using `ChronoUnit.DAYS.between()` would give 365 days.

3 Static

Each attribute and method you have seen until now were related to an instance of a class. But what if we want some of them to be related with the class itself?

Like in *C++*, you can use the `static` keyword:

```
public class Main {
    public static class MyClass {
        public static int myInt = 47;

        public static void myPrint() {
            System.out.println("Don't be afraid, I just wanna help you.");
        }
    }

    public static void main(String[] args) {
        MyClass.myPrint();
        System.out.println(MyClass.myInt);
    }
}
```

Output:

```
Don't be afraid, I just wanna help you.
47
```

Going further...

Actually, you already have used static methods (`public` **static** `void main`) and attributes (`System.out`: `out` is a static attribute of the `System` class).

Again, like in *C++*, there are limitations: a static method can neither use non-static attributes nor call non-static methods:

```
public class Main {
    public static class MyStaticClass {
        private final String myNonStaticString = "Non-static String!";
        private static final String myStaticString = "Static String!";

        public void myNonStaticMethod() {
            System.out.println("Non-static method!");
        }
```

(continues on next page)

(continued from previous page)

```
        public static void myStaticMethod() {
            System.out.println("Static method!");
        }
        public static void run() {
            System.err.println(myNonStaticString); /* Error: Non-static field
                                                    * myNonStaticString cannot be referenced
                                                    * from a static context
                                                    */
            System.out.println(myStaticString); // Good!

            myNonStaticMethod(); /* Error: Non-static method myNonStaticMethod cannot be
                                  * referenced from a static context
                                  */
            myStaticMethod(); // Good!
        }
    }
}
```

3.1 Initializer blocks

An initializer block is a code block written in your class file. It can either be static or not, in which case it is named an “instance initializer block”.

An instance initializer block is executed at every object instantiation, while a static initializer block is executed at **Class Loading**.

These blocks are especially useful if a variable initialization requires some logic (error handling or `for` loop for instance), and you do not want to clutter your constructor with this kind of stuff since it will always be the same algorithm.

```
public class Main {
    public static class MyStaticClass {
        private static List<Integer> myStaticList = new ArrayList<>();
        private List<Integer> myList = new ArrayList<>();

        static {
            for (int i = 0; i < 5; i++) {
                myStaticList.add(2 * i);
            }

            System.out.println("Static initializer block executed!");
        }

        {
            for (int i = 0; i < 5; i++) {
                myList.add(2 * i + 1);
            }

            System.out.println("Instance initializer block executed!");
        }

        public MyStaticClass() {
```

(continues on next page)

(continued from previous page)

```
            System.out.println("MyStaticClass instantiated!");
        }
    }

    public static void main(String[] args) {
        System.out.println("Program starting!");
        final var myInstance = new MyStaticClass();
    }
}
```

Output:

```
Program starting!
Static initializer block executed!
Instance initializer block executed!
MyStaticClass instantiated!
```

Be careful!

This example might trick you into thinking that an instance initializer block is executed before the constructor. It is rather copied into the constructor, after the call to the `super` constructor (which might be implicit if the `super` constructor does not take any arguments):

```
public class Main {
    public static class ParentClass {
        public ParentClass() {
            System.out.println("Parent class constructor!");
        }
    }

    public static class ChildClass extends ParentClass {
        {
            System.out.println("Child class initializer block!");
        }

        public ChildClass() {
            System.out.println("Child class constructor!");
        }
    }

    public static void main(String[] args) {
        final var childInstance = new ChildClass();
    }
}
```

Output:

```
Parent class constructor!
Child class initializer block!
Child class constructor!
```

3.2 Static nested classes

As you have seen in the previous tutorial, *Java* allows you to define nested classes. Nested classes can either be non-static (Inner classes, that you have seen in D1) or static (Static nested classes).

Let us see how it works:

```
public class Main {
    public static class MyClass {
        public static class MyStaticNestedClass {
            public void myPrint() {
                System.out.println("I am the static nested class!");
            }
        }
    }

    public static void main(String[] args) {
        final var myInstance = new MyClass.MyStaticNestedClass();
        myInstance.myPrint();
    }
}
```

Output:

```
I am the static nested class!
```

Just like inner classes, static nested classes can be given any modifier: `public`, default, `protected` or `private`.

While inner classes, being non-static, have to exist within an instance of their enclosing class and can therefore access any of its instance members, static nested classes can only access the static fields of their enclosing class.

Therefore, they are mostly used for packaging convenience rather than for the access to these fields. For this very reason, declaring a private static nested class would not be particularly useful.

```
public class Main {
    public static class MyClass {
        public String myNonStaticString = "Non-static String!";
        public static String myStaticString = "Static String!";

        public void myNonStaticMethod() {
            System.out.println("Non-static method!");
        }

        public static void myStaticMethod() {
            System.out.println("Static method!");
        }

        public static class MyStaticNestedClass {
            public void run() {
                System.err.println(myNonStaticString); /* Error: non-static field
                                                        * myNonStaticString cannot be
                                                        * referenced from a static context
                                                        */
```

(continues on next page)

(continued from previous page)

```
                System.out.println(myStaticString); // Good!

                myNonStaticMethod(); /* Error: non-static method myNonStaticMethod cannot be
                                      * referenced from a static context
                                      */
                myStaticMethod(); // Good!
            }
        }
    }
}
```

Just like inner classes, static nested classes can declare variables that could shadow variables from their enclosing class. Once again, the solution to this is a small syntax trick:

```
public class Main {
    public static class EnclosingClass {
        private static final int myShadowedInt = 1;
        private static final int myNonShadowedInt = 2;

        public static class NestedClass {
            private final int myShadowedInt = 47;
            private final int myNonShadowedInt = 3;

            public void run() {
                System.out.println("My shadowed int: " + myShadowedInt);
                System.out.println("My non-shadowed int: " + EnclosingClass.myNonShadowedInt);
            }
        }
    }

    public static void main(String[] args) {
        final var nestedClassInstance = new EnclosingClass.NestedClass();
        nestedClassInstance.run();
    }
}
```

Output:

```
My shadowed int: 47
My non-shadowed int: 2
```

3.3 Exercise

3.3.1 Objectives

You have to use `static attributes` to keep track of the number of pens. You will have to count red pens, blue pens, and both.

3.3.2 Specifications

Create a public class named `Pen` with the `Enum` named `Color` containing the color `RED` and the color `BLUE`.

Prototype:

```
public Pen(final Color color) { /* ... */ }
```

Then create the following static methods.

getPenCounter

Get the number of pens instantiated.

Prototype:

```
public static int getPenCounter() { /* ... */ }
```

getRedPenCounter

Get the number of red pens instantiated.

Prototype:

```
public static int getRedPenCounter() { /* ... */ }
```

getBluePenCounter

Get the number of blue pens instantiated.

Prototype:

```
public static int getBluePenCounter() { /* ... */ }
```

Create the following public methods.

print

This function will print to `stdout` with the following format:

```
I'm a {color} pen.
```

Prototype:

```
public void print() { /* ... */ }
```

changeColor

Change the color of the pen. Don't forget to update the counters.

Prototype:

```
public void changeColor(final Color color) { /* ... */ }
```

4 Generics

4.1 Introduction

Generics were introduced with *Java 5* and allow to write safe and generic code. Even if *Java*'s generics and *C++*'s templates serve the same purpose, they differ. Therefore, to avoid any confusion, please forget about *C++* templates for now and focus on the following explanations.

Generics allow you to create type-safe classes, interfaces and methods working with various kinds of data.

Java always offered the possibility to create classes that operate on various types of objects by using the root of the class hierarchy: the `Object` class. Yet, this technique did not ensure type safety. Here is a simple example to understand why using generics is type-safe:

```
public class MyObject {
    private Object obj;

    public void setObj(Object obj) {
        this.obj = obj;
    }

    public Object getObj() {
        return obj;
    }
}
```

The idea is that this `MyObject` class will accept any kind of type. You could insert a `String` when an `Integer` was expected somewhere else in your code for example. By using generics, you guarantee which type the generic class operates on.

```
public class MyGenericObject<T> {
    private T obj;

    public void setObj(T obj) {
        this.obj = obj;
    }

    public T getObj() {
        return obj;
    }
}

public class Main {
    public static void main(String[] args) {
        MyGenericObject<String> str = new MyGenericObject<>();
        // ...
    }
}
```

4.2 Overview

As you have already seen, you have been using generics since day 1, with `Lists`. Indeed, when you write `List<Double>`, you are using a special case of `List<T>` with `T = Double`.

4.2.1 Example

The following example shows how to use generics in *Java*.

```
// The class takes a generic parameter ELEMENT_TYPE
public static class Stack<ELEMENT_TYPE> {
    private List<ELEMENT_TYPE> list = new ArrayList<>();

    // The elements we can push on the stack can only be of type ELEMENT_TYPE
    void push(ELEMENT_TYPE element) {
        list.add(0, element);
    }

    ELEMENT_TYPE pop() {
        ELEMENT_TYPE result = list.get(0);
        list.remove(0);
        return result;
    }

    // Does not compile as the method is static
    static ELEMENT_TYPE myStaticMethodNotWorking(ELEMENT_TYPE t) {
        return t;
    }
```

(continues on next page)

(continued from previous page)

```
    // U is declared between chevrons at the beginning of the prototype
    public static <U> U myStaticMethod(U u) {
        return u;
    }
}
```

You can create generic classes and interfaces but also generic methods, that introduce their own type parameters, limited to the method's scope, like the last method in the above example. You have to specify the type parameters between angle brackets before the return type of the method.

Be careful!

Static methods can be called even if no object of the class has been instantiated. Due to this, you cannot rely on generic types declared with the class. You have to redeclare it on the method prototype like in the example above.

Going further...

You can check the available slides on the intranet for further explanations and examples about generics in *Java*.

5 Collections

There are different kinds of collections, the most common ones being **lists**, **maps** and **sets**.

Be careful!

We **strongly** recommend you to check the documentation of each collection to get used to the tools Java gives you.

5.1 Lists

- ArrayList: A simple vector with lots of methods. Can be used as a LIFO. It is optimized for get $\mathcal{O}(1)$, but $\mathcal{O}(n)$ at worst for insertion/deletion. An `ArrayList` increases its internal array size by 50 percent when it runs out of space.
- LinkedList: Can be used as a FIFO. It is optimized for insertion/deletion $\mathcal{O}(1)$, but $\mathcal{O}(n)$ at worst for get.
- Vector: `ArrayList`'s thread-safe counterpart. As it has to lock, it is slower: therefore, we advise you to use `ArrayList` whenever possible for performance reasons. A `Vector` doubles the size of its internal array when it runs out of space.
- Deque: stands for Double-Ended Queue. The name speaks for itself: it is a linear collection supporting access, insertion and removal at both ends.

5.2 Maps

`Maps` associate a key to a value. They are sometimes called *associative arrays*.

- HashMap: Uses a hash method. Insertion, lookup and deletion are executed in constant time $\mathcal{O}(1)$ but linear time $\mathcal{O}(n)$ is required for iteration.
- ConcurrentHashMap: `HashMap`'s thread-safe counterpart. Just like `Vector` with `ArrayList`, `ConcurrentHashMap` is slower than `HashMap`: use the latter whenever possible. Moreover, the `ConcurrentHashMap` allows neither keys nor values to be `null`.
- TreeMap: Uses a balanced tree (depending on the underlying implementation). Therefore, this map is guaranteed to be ordered, but it cannot afford constant-time performance where `HashMap` can: all basic operations, such as get, put and remove, are executed in $\mathcal{O}(\log n)$.

5.3 Sets

A set is a container in which, compared to lists, **ordering is less important than existence**. A list can contain the same element several times, whereas a set can't say more than "the element is (or isn't) present". Therefore, just like maps, some sets do not have any ordering. This makes them more efficient than lists for search operations.

- HashSet: Implemented with a hash method (actually a `HashMap`): no ordering. Basic operations are executed in constant-time $\mathcal{O}(1)$.
- TreeSet: Implemented with a balanced tree (actually a `TreeMap`). Basic operations are executed in logarithmic-time $\mathcal{O}(\log n)$, but we can safely iterate upon a `TreeSet`. `TreeSet` is an ordered container.
- LinkedHashSet: Extends `HashSet`, but maintains a doubly-linked list running through its entries. This way, the order of iteration across the Set is the order in which the elements were inserted into the Set.

Unfortunately, there is no built-in thread-safe set amongst Java collections. You can however make an existing set thread-safe by using Collections#synchronizedSet.

5.4 Exercise

5.4.1 Objectives

In this exercise, you will have to implement a simplistic `Set` of `Integer`, **sorted in ascending order**.

5.4.2 Specifications

- Step 1

For this step, you **must** use an `ArrayList` of `Integer` named `base_`. All functions will be tested separately.

- Step 2

For this step, you have to make the Set generic. Copy the Set class you implemented in the first step into the `genericset` package and try to make it generic (not only for `Integers`).

Tips

Try to remember about the `Comparable` interface.

5.4.3 Prototypes

```
public Set()
public void insert(Integer i)
public void remove(Integer i)
public Boolean has(Integer i)
public Boolean isEmpty()
public Integer min()
public Integer max()
public Integer size()
public static Set intersection(Set a, Set b)
public static Set union(Set a, Set b)
```

Be careful!

All your functions will be tested separately.

The *remove* method will do nothing if the element *i* is not present in the set. The case of an empty set will not be tested with the methods *min* and *max*.

6 Exceptions

6.1 Introduction

You can handle errors with the exception mechanism. You should already know the basics of exception handling thanks to your experience with *C++* which provides a similar mechanism.

As a small reminder, you can interrupt a procedure's execution by throwing an exception, say `MyException`, and unstack all parent function calls until the exception is caught. If you do not catch an exception that has been thrown, the program terminates.

6.2 Basics

An exception is a class that extends the abstract class Exception. You can use any non-abstract exception from the *Java* standard library, or create your own, like so:

```
public class Main {
    public static class InvalidNumberException extends Exception {
        private final int number;

        public InvalidNumberException(final int number) {
            this.number = number;
        }

        public void print() {
            System.err.println("Invalid number: " + number);
        }
    }
}
```

To throw an exception, you must use the `throw` keyword. However, when declaring a method, you **must** declare all the exceptions it may throw in its signature, using the `throws` keyword.

```
public static long factorial(int n) throws InvalidNumberException {
    if (n < 0) {
        throw new InvalidNumberException(n);
    } else {
        return (n == 0) ? 1 : n * factorial(n - 1);
    }
}
```

By checking for uncaught exceptions, the compiler forces you to handle all of those which were declared, thus making your code more robust!

Like in *C++*, when you want to handle an exception, you must enclose the method call that may throw it into a `try` block. You then just have to handle it inside the following `catch` block.

Be careful!

If you do not manage an exception that may be thrown by a method call inside your method, you must declare the exception in the method's signature.

```
public static void main(String[] args) {
    int i = 5;
    while (true) {
        try {
            final long fact = factorial(i);
            System.out.println(fact);
            i--;
        } catch (InvalidNumberException e) {
            e.print();
            break;
        }
    }
}
```

Output:

```
120
24
6
2
1
1
Invalid number: -1
```

Going further...

The method `Exception#printStackTrace` prints the state of the stack at the time the exception has been thrown. This can come in handy, make sure to check it out!

6.3 Advanced usage

6.3.1 Runtime exceptions

It can be troublesome to add a `catch` block for an exception that may never be thrown. For example, *Java* provides the exception `NullPointerException` which is thrown each time one tries to invoke a method on an object which value is `null`. Adding a `catch (NullPointerException e)` for each `object.method(...)` call would completely worsen the code readability.

Runtime exceptions are exceptions that *do not* require you to add a `throws` declaration to the signature of the method in which you throw your exception. Thus, no `try...catch` is needed to call your method. These exceptions extend the class `RuntimeException`. You still can catch them in a `catch` block, but you do not have to.

Going further...

For more information, read this documentation page.

6.3.2 Multiple exceptions

A method can throw more than one exception: you just have to chain the exception types with commas after the `throws` keyword. Several `catch` blocks can also be associated to the same `try` block.

```
public class Main {
    public static class InvalidNameException extends Exception{
        public InvalidNameException() {
            System.err.println("Empty name");
        }
    }

    public static class InvalidAgeException extends Exception {
        public InvalidAgeException(final int age) {
            System.err.println("Invalid age: " + age);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    public static class Person {
        public final String firstName;
        public final String lastName;
        public final int age;

        public Person(final String firstName, final String lastName, final int age)
                throws InvalidNameException, InvalidAgeException { // Multiple throws
            if (firstName.isEmpty() || lastName.isEmpty()) {
                throw new InvalidNameException();
            } else if (age < 0) {
                throw new InvalidAgeException(age);
            }

            this.firstName = firstName;
            this.lastName = lastName;
            this.age = age;
        }
    }

    public static void main(String[] args) {
        try {
            final var myPerson = new Person("Papy", "Russe", -21);
        } catch (InvalidNameException e) { // Multiple catch blocks
            e.printStackTrace();
        } catch (InvalidAgeException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Invalid age: -21
Main$InvalidAgeException
  at Main$Person.<init>(Main.java:24)
  at Main.main(Main.java:35)
```

6.3.3 Throwing caught exceptions

Sometimes, an exception has to be caught to be processed, but has to be thrown again for the parent function calls to catch it. Thankfully, the `throw` keyword also accepts a variable name. Instead of creating a new instance of the exception, pass the former to throw.

```
public static void main(String[] args) throws InvalidNameException, InvalidAgeException {
    try {
        final var myPerson = new Person("Papy", "Russe", -21);
    } catch (InvalidNameException e) {
        e.printStackTrace();
        throw e;
    } catch (InvalidAgeException e) {
```

(continues on next page)

(continued from previous page)

```
        e.printStackTrace();
        throw e;
    }
}
```

Output:

```
Invalid age: -21
Main$InvalidAgeException
  at Main$Person.<init>(Main.java:24)
  at Main.main(Main.java:35)
Exception in thread "main" Main$InvalidAgeException
  at Main$Person.<init>(Main.java:24)
  at Main.main(Main.java:35)
```

6.3.4 The `finally` keyword

You can also define a `finally` block at the end of the `try...catch`. Its content is executed regardless of whether an exception has been caught.

```
public static void main(String[] args) {
    try {
        final var myPerson = new Person("Papy", "Russe", -21);
    } catch (InvalidNameException e) {
        e.printStackTrace();
    } catch (InvalidAgeException e) {
        e.printStackTrace();
    } finally {
        System.out.println("Yep, everything went fine.");
    }
}
```

Output:

```
Invalid age: -21
Main$InvalidAgeException
  at Main$Person.<init>(Main.java:24)
  at Main.main(Main.java:35)
Yep, everything went fine.
```

6.4 Exercise

6.4.1 Goals

You will implement 3 custom exceptions which will be thrown when a given input does not respect a set of constraints.

You will have to write a class `Student` that will contain three fields:

- A `String` named `name` representing the name of the student;

- A `int` named `age` representing the age of the student;
- A `String` named `major` representing the academic major of the student;

The class `Student` constructor will throw one of these exceptions if one of the arguments is invalid. Those conditions are described in the specification part.

Each exception class will have a constructor that takes a `String` or an `int` as an argument. In these constructors, you will create the exception message: you will have to concatenate the name of the exception and the string or int passed as an argument, with **no newline** at the end: `ExceptionName: Argument`

6.4.2 Specifications

The constructor of the class `Student` will throw exceptions according to the following rules:

- If the student's name contains a number: throw an `InvalidNameException`;
- If the student's age is inferior or equal to 0 or superior or equal to 130: throw an `InvalidAgeException`;
- If the student's major is not part of EPITA's majors (image, gistre, srs, scia, mti, tcom, sigl, gitm): throw an `InvalidMajorException`. The major comparison should be case insensitive, however you have to store it in the major field in **lower case**.

If there are multiple invalid arguments, you have to throw the first exception you encounter in this order: `InvalidNameException`, `InvalidAgeException`, `InvalidMajorException`.

You have also to implement a `toString` method which lists all Student fields with **no newline** at the end. Here is an example for Jean-Claude, 21 years old in the mti major: `Name: Jean-Claude, Age: 21, Major: mti`

6.4.3 Prototypes

The `Student` class must implement the following methods:

```
public Student(String name, int age, String majeure) throws
    InvalidNameException, InvalidAgeException, InvalidMajeureException

@Override
public String toString()
```

6.4.4 Examples

For example, doing:

```
new Student("K3vin", 21, "mti");
new Student("Paul", -2, "gistre");
new Student("Maxime", 21, "unknown");
```

Will throw the following error:

```
InvalidNameException: K3vin
InvalidAgeException: -2
InvalidMajeureException: unknown
```

And the following code:

```
Student paul = new Student("Jean-Michel", 19, "iMAgE");
System.out.println(paul.toString());
```

will print:

```
Name: Jean-Michel, Age: 19, Major: image
```

6.5 Exceptions and Generics

Let us try something out! Write an exception class named `TypeException`, which implements a `print` method which prints the type it contains.

```
public static void main(String[] args) throws TypeException {
    /* FIXME: Throw TypeException with whatever type, let's say String.
     *        Catch it and call its 'print' method.
     */
}
```

Output:

```
Faulty type: class java.lang.String
```

Did you know that you cannot define a generic exception? That is called type erasure: since genericity is a compile-time mechanism and exceptions belong to runtime, there is no way to find out what would be the generic type of a caught generic exception.

```
public class Main {
    public static class TypeException<FAULTY_TYPE> extends Exception {
        // Error: Generic type may not extend Throwable? This is madness!
    }
}
```

Therefore, in order to dispel any confusion, the compiler forbids you to declare any of those. In this situation, you may want to use your good old workaround of choice.

Output:

```
Faulty type: class java.lang.String
```

Don't be afraid, I just wanna help you.