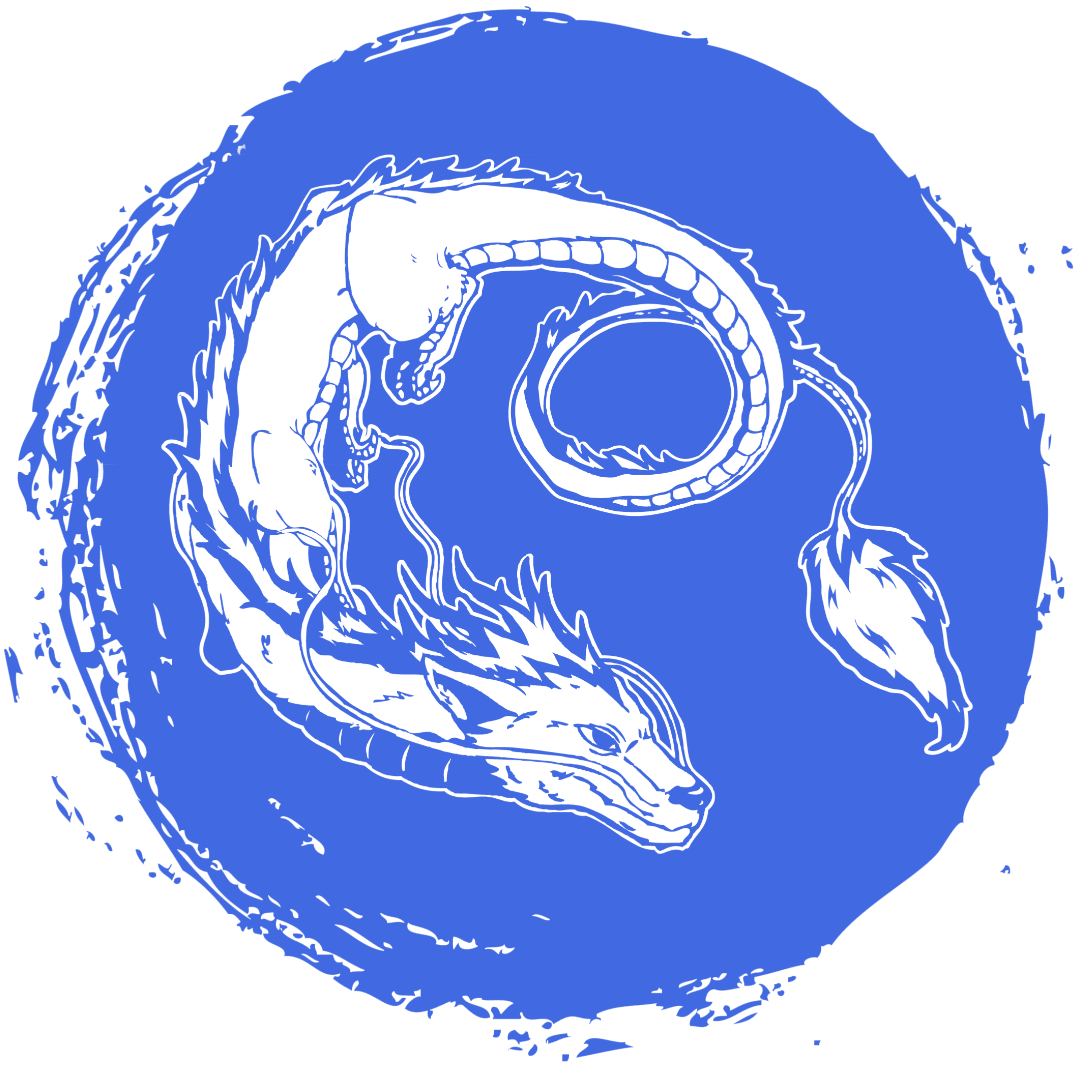
Java Workshop — Tutorial D1

version **#v1.1.0**



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

Copyright

Contents

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

1 Boxing and unboxing

Java is a strongly typed object-oriented language. Every *Java* class inherits from the `Object` class. Yet, for performance reasons, a part of the *Java* language does not inherit from the `Object` class and is not object-oriented: the primitive types.

Those primitive types are `int`, `short`, `long`, `byte`, `char`, `float`, `double` and `boolean`. They represent single values, not complex objects.

However, sometimes you will need to use an object representation of such types. For example, data structures in *Java* only work with objects. This is where wrapper types are useful: `Integer`, `Short`, `Long`, `Double`... They are classes that encapsulate a primitive type within an object.

1.1 What are boxing and unboxing?

Java supports the conversion of *primitive types* into their object equivalents, *wrapper types*, in assignments or methods and constructors invocations. The encapsulation of a primitive value within its object equivalent is known as *boxing*.

The reverse operation is called *unboxing*: *Java* supports the conversion of *wrapper types* into their *primitive* equivalents (if needed) for assignments or methods and constructors invocations.

```java
public int extract(Integer iObj)
{
    int j = iObj.intValue(); // unboxing

    return j;
}

public Integer encapsulate(int i)
{
    Integer jObj = new Integer(i); // boxing

    return jObj;
}
```

These conversions can also be performed automatically by the *Java* compiler. This is called **autoboxing** and **auto-unboxing**. For autoboxing, the primitive type is automatically converted to its object equivalent when needed without calling the constructor of this class.

```java
public Integer autoConvert(int i)
{
    Integer jObj = i; // autoboxing

    return jObj;
}
```

1.2 When should you use boxing/unboxing?

It is not appropriate to use boxing and unboxing without a specific need (i.e. when using data structures) as a repeated conversion will have a performance impact on your program. Moreover, an `Integer` is not a substitute for an `int` — boxing and unboxing blur the distinction between primitive types and reference types, but they don't eliminate it.

2 String

2.1 Operations

There are several ways to instantiate a new `String` object from the String class. Here is a short extract:

```java
public class Main {
    public static void main(String[] args) {
        char[] hello = { 'H', 'e', 'l', 'l', 'o' };
        var helloStr = new String(hello);
        var world = "World!";
        var empty = new String(); // Builds an empty String ("")
        var goodBye = new String("Goodbye");
        var worldCpy = new String(world);

        System.out.println(helloStr);
        System.out.println(world);
        System.out.println(empty);
        System.err.println(goodBye);
        System.err.println(worldCpy);
    }
}
```

Output:

```
Hello
World!

Goodbye
World!
```

Going further...

The `var` keyword allows you to use *local type inference*: the compiler can deduce the variable's type from the given value, in a similar way as *C++*'s `auto`.

Java's `String` doesn't behave the same way as *C++*'s `std::string`. There is no `operator[]` here: if you want to access a character at a certain index, you shall use the `charAt` method.

```java
public class Main {
    public static void main(String[] args) {
        var myString = "Java";

        System.err.println(myString[0]); //Error: array required, String found
```

(continues on next page)

(continued from previous page)

```
        System.out.println(myString.charAt(0)); // Output: J
    }
}
```

Java's `String` also implements several really useful methods:

- `strip`: returns the `String` stripped from all leading and trailing whitespaces;
- `substring`: returns a substring of the `String`;
- `toUpperCase`/`toLowerCase`: returns the given `String` respectively in upper or lower case.

```
public class Main {
    public static void main(String[] args) {
        var tooManyWhitespaces = "      I just  ";
        var bigString = "I wanna feel sunlight on my face.";
        var lowerCase = "help you";

        System.out.print(tooManyWhitespaces.strip());
        System.out.print(bigString.substring(1, 8));
        System.out.println(lowerCase.toUpperCase());
    }
}
```

You can find the documentation about these methods (and a lot more) in the Oracle documentation.

Going further...

More generally, think about reading the Oracle doc about every class from the *Java* API that you have to use.

If you want to compare the content of two `String`, the == operator will not do what you want. Remember that in *Java*, every time you create a new variable, it then holds a reference to an allocated object (except for primitive types). Therefore, comparing two `String` with the == operator will compare their addresses (the same way as comparing two `char*` in *C*). Comparing two `String`'s contents shall be done through the `equals` method.

```
public class Main {
    public static void main(String[] args) {
        var myString = "Don't be afraid.";
        var myStringCpy = new String(myString);

        System.err.println(myString == myStringCpy); // false
        System.out.println(myString.equals(myStringCpy)); // true

        // However...
        var myOtherString = "Don't be afraid.";
        // myString and myOtherString are string literals, they refer to the same memory area.
        System.out.println(myString == myOtherString); // true
    }
}
```

Going further...

If you want to know more about string literals, read the part 3.10.5 of the Java specification.

You can also append a String to another with the operator +. However, you should not use this. *Java* Strings are **immutable**: once they have been instantiated, they cannot be modified. Therefore, using the operator + allocates a brand new String and shoves the contents of both String into it.

2.2 StringBuilder

The StringBuilder class represents a mutable sequence of characters. This is what you should use in order to build a String through appending, using the append method, and then toString, or String's constructor which takes a StringBuilder as argument.

```java
public class Main {
    public static void main(String[] args) {
        var stringBuilder = new StringBuilder("Hello ");

        stringBuilder.append('w');
        stringBuilder.append(0); // Appends '0', not (char)0!
        stringBuilder.append((char)114); // 114 is the ASCII value of 'r'
        stringBuilder.append("ld!");
        System.out.println(stringBuilder.toString());

        var myString = new String(stringBuilder);
        System.out.println(myString);
    }
}
```

Output:

```
Hello w0rld!
Hello w0rld!
```

There is another class, named StringBuffer, which does pretty much the same job as StringBuilder. The main difference between the two of them is that StringBuffer is thread-safe. However, since it has to lock, it is way slower than a StringBuilder. Therefore, use StringBuffer only when necessary.

Be careful!

During this workshop, you might come across some String appended with the operator +, in tutorials for example. This is done only to avoid cluttering. You should use StringBuilder if you want to append anything to a String. Note that, in simple cases, the + operator can be desugared to StringBuilders by the compiler.

3 List

3.1 ArrayList

For the next parts of this tutorial, you will need to use lists. *Java* provides an implementation of this data structure, named ArrayList.

```java
public class Main {
    public static void main(String[] args) {
        var myList = new ArrayList<Integer>();
        // Appends the given list at the end of myList.
        myList.addAll(Arrays.asList(1, 3, 5, 7, 9));

        myList.add(2); // Appends 2 at the end of myList
        myList.remove(3); // Remove the element at index 3 (7)
        myList.set(0, 47); // Sets the value at index 0 to 47

        final var myListSize = myList.size(); // Number of elements in the list
        // Print the content of myList
        for (var index = 0; index < myListSize; index++) {
            var valAt = myList.get(index); // Gets the element at the given index
            System.out.println(valAt);
        }
    }
}
```

Output:

```
47
3
5
9
2
```

The syntax of `ArrayList` might remind you of *C++*'s templates. The type between angle brackets ('< >') is called a generic type. You will see more about them in the next tutorial (D2). As we said in the part about *boxing*, data structures in Java only work with objects: keep in mind that you cannot pass a primitive type (`int`, `float`, `char`...) as a generic type, you can only use their object counterpart (respectively `Integer`, `Float`, `Character`).

3.2 Iteration

Iterating over an `ArrayList` using an `int` index can be quite tiresome. Thankfully, *Java* provides a convenient way to smoothly iterate over data structures:

```java
public class Main {
    public static void main(String[] args) {
        var myList = new ArrayList<Integer>(Arrays.asList(1, 3, 5, 47, 9));
        for (var value : myList) {
            System.out.println(value);
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Output:

```
1
3
5
47
9
```

You might already have used a similar iteration method in *C++*. However, while in *C++* you iterated over... iterators, here you directly iterate over the elements, which is much more instinctive.

3.3 Exercise

In this exercise, you will have to create a class `RequestBuilder` that will prepare SQL requests using the class `String`.

Be careful!

This exercise does not aim at teaching you how to properly manage Databases, but rather at making you practice `String` manipulation in Java.

Each `RequestBuilder` instance will be associated to one table, given as argument in the constructor. The class must also implement 3 public methods:

- `public String buildSelect(List<String> columnNames)`, that will return a SQL SELECT Request as a String. It will take as argument the list of columns to select. If none is specified, you will have to select all of the colums.
- `public String buildInsert(String columnName, List<String> toInsert)`, that will return a SQL INSERT Request as a String. It will take as arguments the name of the column to insert into and a list of values to insert.
- `public String buildUpdate(String columnName, String newValue, List<String> conditions)`, that will return a SQL UPDATE Request as a String. It will take as arguments the name of the column to alter, the new value and a list of conditions to match, following the pattern: key='word'. You have to check that this pattern is correct for the given conditions, and to return null if it is not. You can only chain conditions using `AND`.

Your class should also implement the `getTableName` and `setTableName` methods.

In addition, you must check that all the elements passed as arguments are case sensitive alphanumerics that can also contain dashes and spaces, otherwise, you will return `null`. The documentation of `String#matches` might help you. Also, you might want to take a look at the documentation of the class `StringJoiner` in order to make a clever build of your requests.

The output should have the following format:

```
SELECT <columns> FROM <table_name>;
INSERT INTO <table_name> (<columns>) VALUES ('<value1>'), ('<value2>');
UPDATE <table_name> SET <column>='<value>' [WHERE <conditions>];
```

Which would look like, with a given `movie-theater` table:

```
SELECT title, duration, director FROM movie-theater;
INSERT INTO movie-theater (title) VALUES ('Alien'), ('The Departed');
UPDATE movie-theater SET director='Ridley Scott' WHERE title='The Martian';
```

4 Annotations

Java annotations provide a way to ease the development of an application and prevent potential future errors. They are like "meta-tags" that you can add directly inside your code and apply to package declarations, type declarations, constructors, methods, fields, parameters, variables... As a result, they allow you to provide additional information about your code to the compiler or the JVM.

The basic form of an annotation is a keyword preceded by an `@` symbol. For example:

```
@Annotation
public void MyMethod() { ... }
```

An annotation can include elements with values:

```
@Authors(
    name = "Name"
    date = 11/04/2018
)
class MyClass() { ... }
```

There are different types of annotations. The *Java* language provides you some predefined annotations. Several of them are helpful to the *Java* compiler.

The annotation you will the most frequently use is the `@Override` annotation, defined in `java.lang`. This annotation can only be used in the context of inheritance. If you want to write a method in a subclass that will override a method in a superclass, you might want to annotate it with the `@Override` annotation. The compiler will then check if it does override its superclass method correctly (with the correct return value and arguments) and will generate an error if it is not the case. We **strongly** recommend you to use it: by doing so, you will easily see if you made a mistake when you tried to override a method.

```
public class Example
{
    @Override
    public String toString()
    {
        // this is a valid override
        return super.toString() + "Testing annotation name: `Override`";
    }
}
```

In the previous example, `toString` is a method of the `Object` class, so we can override it (as every class has `Object` as a superclass). You will see another example in a few minutes.

There are other predefined annotations such as `@FunctionalInterface` or `@Deprecated`.

You will also encounter annotations when building testsuites using JUnit, as they allow you to configure your tests (`@Test`, `@Timeout`, `@Before`...).

Another kind of annotations are meta-annotations, annotations that apply to other annotations, defined in `java.lang.annotation`. If you want to know more about them, please see the corresponding documentation page.

Finally, it is also possible to create your own annotations. This is not the subject of this tutorial but if you are curious about this concept, you can learn more there.

5 Inheritance

Java being intrinsically an Object-Oriented language (except for primitive types, everything is an object), it implements inheritance, in a way that is pretty similar to what you have seen in *C++*.

5.1 Modifiers

Like in *C++*, different access permissions can be set for *Java* class members:

- `public`: accessible by any other object or class;
- `protected`: accessible by the class, subclasses and classes in the same package;
- *default*: used if you do not specify any visibility; the member is accessible by any class in the same package;
- `private`: accessible only by instances of the class.

Summary:

Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
default	Yes	Yes	No	No
`private`	Yes	No	No	No

Going further...

To indicate a **default** visibility, there is no need for a keyword.

5.2 Basics

An `extends` keyword put in a class declaration means that the class on the left of the keyword inherits from the class on its right.

```java
public class Main {
    public static class ParentClass {
        protected String inheritedString = "Inheritance is cool!";

        public void printPublic() {
            System.out.println("Hello world!");
        }

        protected void printProtected() {
            System.out.println("Hello subclasses!");
        }

        void printDefault() {
            System.out.println("Hello package!");
        }

        private void printPrivate() {
            System.err.println("Hello myself...");
        }
    }

    public static class ChildClass extends ParentClass {
        public void run() {
            printPublic();
            printProtected();
            printDefault();
            printPrivate(); // Error: printPrivate has private access in ParentClass

            System.out.println(inheritedString);
        }
    }

    public static void main(String[] args) {
        var child = new ChildClass();
        child.run();
    }
}
```

Going further...

All Java classes inherit from the Object class.

In *Java*, methods are virtual by default, which means that if you write in your child class a method with the same prototype as one of its superclass, the superclass' method will automatically be overridden. Moreover, just like *C++*, *Java* is **polymorphic**, which means an instance of a class can be cast into any of its superclasses.

```java
public class Main {
    public static class ParentClass {
```

(continues on next page)

(continued from previous page)

```
        public void inheritedMethod() {
            System.out.println("Parent method");
        }
    }

    public static class ChildClass extends ParentClass {
        public void inheritedMethod() {
            System.out.println("Child method");
        }
    }

    public static void main(String[] args) {
        ParentClass parent = new ParentClass();
        ChildClass child = new ChildClass();

        /* Polymorphism: ParentClass is actually polymorphicChild's static type
         * (compile-time).
         * Its dynamic type (runtime) is ChildClass.
        */
        ParentClass polymorphicChild = new ChildClass();

        parent.inheritedMethod();
        child.inheritedMethod();

        /* Dynamic dispatch: call the method from the dynamic type, since methods are virtual
         * by default.
        */
        polymorphicChild.inheritedMethod();
    }

}
```

Output:

```
Parent method
Child method
Child method
```

Yet, through the annotation mechanism, the Java language provides a way to ensure that the superclass method is overridden correctly by indicating this method as overridden to the compiler, using the `@Override` annotation.

```
public class Main {
    public static class ParentClass {
        public void overriddenMethod(Integer i) {
            System.out.println("You gave me an Integer: " + Integer.toString(i));
        }
    }

    public static class ChildClass extends ParentClass {
        @Override // Error: Method does not override method from its superclass
        public void overriddenMethod(int i) { // int instead of Integer
            System.err.println("You gave me an int: " + Integer.toString(i));
```

(continues on next page)

(continued from previous page)

```java
        }
    }

    public static void main(String[] args) {
        var parent = new ParentClass();
        var child = new ChildClass();

        parent.overriddenMethod(47);
        child.overriddenMethod(47);
    }
}
```

If the superclass of a class does not define a constructor which takes no argument (called *no-arg constructor*), the child class' constructor will not compile: that is because no constructor can be implicitly called for the superclass. Worry not! You can call the parent class' constructor with the `super` keyword.

The `super` keyword is a little peculiar, as it takes arguments. Used in a constructor, it calls to the constructor of the superclass with the given arguments. If you don't explicitly call `super` in the child class' constructor, it will try to implicitly call the superclass' no-arg constructor: therefore, if the superclass does not define one, the compiler will issue an error.

```java
public class Main {
    public static class ParentClass {
        // Implicitly defines a default no-arg constructor
    }

    public static class ChildClass extends ParentClass {
        protected String inheritedString;

        public ChildClass(final String inheritedString) {
            // Implicit call to the parent class' no-arg constructor (super())
            this.inheritedString = inheritedString;
        }
         // No need to define a default no-arg constructor, since one is provided
    }

    public static class BadGrandChildClass extends ChildClass {
        /* Error: Cannot find any no-arg constructor for ChildClass: cannot define a default
         * no-arg constructor
         */
    }

    public static class GoodGrandChildClass extends ChildClass {
        public GoodGrandChildClass(final String inheritedString) {
            super(inheritedString); // Call to the parent constructor
        }
    }
}
```

Going further...

It is a good practice to always explicitly call the parent constructor, even when it could be done implicitly through a no-arg constructor.

You can also use the `super` keyword in a method: in this case, it will try to call the overridden superclass' method.

```java
public class Main {
    public static class ParentClass {
        public void myPrint() {
            System.out.println("Hello");
        }
    }

    public static class ChildClass extends ParentClass {
        @Override
        public void myPrint() {
            super.myPrint();
            System.out.println("you!");
        }
    }

    public static void main(String[] args) {
        var child = new ChildClass();
        child.myPrint();
    }
}
```

Output:

```
Hello
you!
```

5.3 Runtime type-checking

The `instanceof` keyword checks if an object reference is an instance of a type and returns `true` if the object is an instance of the given type or any of its supertypes. Notice that `instanceof Object` always returns `true` since all Java objects are inherited from `Object`, and `instanceof NullType` always returns false.

```java
public class Main {
    public static class ParentClass {
    }

    public static class ChildClass extends ParentClass {
    }

    public static void main(String[] args) {
        ParentClass parent = new ParentClass();
        ChildClass child = new ChildClass();
        ParentClass polymorphicChild = new ChildClass();

        System.out.println(parent instanceof Object); // Always true
        System.out.println(parent instanceof NullType); // Always false

        System.out.println(parent instanceof ParentClass); // true
        System.out.println(parent instanceof ChildClass); // false
```

(continues on next page)

(continued from previous page)

```
        System.out.println(child instanceof ParentClass); // true (inheritance)
        System.out.println(child instanceof ChildClass); // true

        System.out.println(polymorphicChild instanceof ParentClass); // true
        System.out.println(polymorphicChild instanceof ChildClass); /* true (these checks are
                                                                      * performed at runtime,
                                                                      * keep polymorphism in
                                                                      * mind!)
                                                                      */
    }
}
```

In *Java*, classes are also objects. Indeed, there is a class in *Java* called Class, which any class is an instance of. You can retrieve a `Class` instance from an object or from a type name:

- If you want to retrieve it from a type name (called *type literal*), append `.class` to it.
- If you want to retrieve it from an object, you can call the `getClass` method on it (inherited from `Object`).

```
public class Main {
    public static void main(String[] args) {
        System.out.println(String.class); // From type literal

        var myString = new String();
        System.out.println(myString.getClass()); // From instance
    }
}
```

Output:

```
class java.lang.String
class java.lang.String
```

The `Class` class defines methods which allow you to perform runtime type-checks similar to those you did with `instanceof`:

- `isAssignableFrom`: returns `true` if the calling `Class` is the same or a superclass of the one given as argument;
- `isInstance`: returns `true` if the `Object` given as argument is an instance of the calling `Class` or one of its child classes.

```
public class Main {
    public static class ParentClass {
    }

    public static class ChildClass extends ParentClass {
    }

    public static class GrandChildClass extends ChildClass {
    }

```

(continues on next page)

(continued from previous page)

```java
    public static void main(String[] args) {
        var myClass = ChildClass.class;
        System.out.println(myClass.isAssignableFrom(GrandChildClass.class)); // true
        System.out.println(myClass.isAssignableFrom(ChildClass.class)); // true
        System.out.println(myClass.isAssignableFrom(ParentClass.class)); // false

        var child = new ChildClass();
        System.out.println(ParentClass.class.isInstance(child)); // true (inheritance)
        System.out.println(ChildClass.class.isInstance(child)); // true
        System.out.println(GrandChildClass.class.isInstance(child)); // false
    }
}
```

Going further...

Notice that `Class#isAssignableFrom` checks for an inheritance relationship opposite to the one checked by `instanceof` or `Class#isInstance`.

5.4 Abstract classes

As a small reminder, an **abstract class** is a class that is not made to be instantiated, but rather to define methods and attributes common to several other classes, that will inherit from it.

In *C++*, you could define abstract classes by declaring at least one virtual method without an implementation (for instance: `virtual void foo() = 0;`). *Java* provides a keyword to make it apparent right away: you can declare a class as `abstract`.

```java
public class Main {
    public static abstract class ParentClass {
    }

    public static class ChildClass extends ParentClass {
    }

    public static void main(String[] args) {
        // Error: Cannot be instantiated
        ParentClass parent = new ParentClass();


        // Good: polymorphicChild is actually an instance of ChildClass
        // stored in a ParentClass
        ParentClass polymorphicChild = new ChildClass();
    }
}
```

You can also define **abstract methods** in a class: a method that is not yet implemented, but will be in the child classes. You can do so by adding the `abstract` keyword to a method declaration. Beware: if your class declares any abstract method, it **must** be declared as `abstract`.

One last thing: if your abstract class declares abstract methods, any non-abstract class inheriting from it **must** implement all of them.

```java
public class Main {
    public static abstract class ParentClass {
        public abstract void abstractPrint();
    }

    public static class BadParentClass {
        public abstract void abstractPrint(); // Error: Abstract method in non-abstract class
    }

    public static class IncompleteChildClass extends ParentClass {
        /* Error: Since IncompleteChildClass doesn't implement abstractPrint, it must also be
         * marked as abstract.
         */
    }

    public static class ChildClass extends ParentClass {
        @Override
        public void abstractPrint() {
            System.out.println("Knowledge only grows through challenge.");
        }
    }
}
```

6 Interface

You might remember that *C++* allows **multiple inheritance**. Well, *Java* does not. Or "*not exactly*". In fact, event though a class can inherit from only one other **class**, it can inherit from multiple **interfaces**.

An `interface` is kind of an abstract class, which only defines unimplemented methods. Hence the name: any class which inherits from it must at least implement its *interface*. To make a class inherit from an `interface`, you must use the `implements` keyword.

```java
public class Main {
    public interface OutPrinter {
        void printOut();
    }

    public interface ErrPrinter {
        void printErr();
    }

    public static class MyClass implements OutPrinter, ErrPrinter {
        private int myInt;

        public MyClass(int myInt) {
            this.myInt = myInt;
        }

        @Override
        public void printOut() {
            System.out.println(myInt);
        }
```

(continues on next page)

(continued from previous page)

```
        @Override
        public void printErr() {
            System.err.println(myInt);
        }
    }

    public static void main(String[] args) {
        var myClass = new MyClass(47);

        myClass.printOut();
        myClass.printErr();

        /* Output:
            47
            47
         */
    }
}
```

Beware: if a non-abstract class implements an `interface`, it **must** implement all the methods declared by this `interface`.

```
public class Main {
    public static interface MyInterface {
        public void overriddenMethod();
        public void notOverriddenMethod();
    }

    public static class WrongInheritanceClass implements MyInterface {
        @Override
        public void overriddenMethod() {
            System.out.println("Overridden method!");
        }

        // Error: WrongInheritanceClass does not override notOverriddenMethod
    }

    public static abstract class PartialInheritanceClass implements MyInterface {
        @Override
        public void overriddenMethod() {
            System.out.println("Overridden method!");
        }

        /* Correct: Every subclass from PartialInheritanceClass will have to implement
         * notOverriddenMethod.
         */
    }
}
```

Going further...

The `instanceof` keyword can be used to check if a reference to an object matches an `interface`.

6.1 Exercise

This is an exercise about *inheritance*.

Create the class hierarchy allowing the implementation of the following description using classes, abstract classes, enums, interfaces and collections.

- An individual:
  - Has a name.
- A super-individual:
  - Is an individual.
  - Has a super-name.
  - Can be a neutral, a hero or a villain.
  - Has super-powers.
  - Has a species: human, machine, extra-terrestrial or unknown.
- A super-power:
  - Has a name.
  - Has a category: movement, offense, defense, other (and possibly a sub-category).

Some super-individuals have interesting features. Some are bankable: they are so well-known that it becomes interesting to make a production solely about them. Others, the younglings, attract younger audiences and can be cast in teen-friendly productions.

Tips

There are several working answers, the idea behind this exercise is to manipulate the different types of inheritance and composition, go wild!

7 Sealed classes and interfaces

The possibility to seal classes and interfaces was introduced as a preview in the JDK 15 and improved in the JDK 16, with the intent to give a finer grained control over class inheritance and interface implementation mechanisms.

Through class hierarchy, *Java* allows you to factorize and reuse code in numerous subclasses. Yet, reusing code is not always a goal when developing. Sometimes, it might be useful, for a clarity purpose, to restrict the classes that will be able to inherit from a certain superclass or implement a certain interface.

To be able to do so, some keywords were introduced in the *Java* language: `sealed`, `permits` and `non-sealed`.

Let us say we want to represent the majors that exist at Epita by creating an abstract class `Major`. We know that only some specific classes will be valid subclasses of `Major`: Finance is not a valid major name at Epita for example. Thus, we would like to restrict the use of our superclass `Major` and specify the classes that are allowed to extend it, by making `Major` a sealed class:

```java
public abstract sealed class Major
    permits Image, Gistre, MTI, TCOM, Santé, GITM, SRS, SIGL, SCIA
{
    /* ... */
}
```

The `permits` keyword must be placed after the `extends` or `implements` clauses of a superclass or interface declaration.

Then you must declare your subclasses either in the same module as your superclass or in the same package.

The permitted subclasses must also declare a modifier:

- `final`: you will see this keyword in detail tomorrow. In this case, it prevents any further extension of the subclass;
- `sealed`: to restrict the extension of the subclass to specific classes;
- `non-sealed`: to open anew the possibility to freely extend the subclass.

The purpose of sealed classes is to limit the use of a superclass and at the same time, let it be accessible. Declaring a class with the default visibility mode restrains the classes that are able to extend it to only those present in the same package. At the same time, it conceals the existence of this superclass to all the classes outside the package.

8 Pattern matching

Since the recent release of the JDK 16, the *Java* language supports pattern matching on the `instanceof` operator.

As a reminder, pattern matching consists of testing the form or the type of a value to associate to this pattern an expression to compute.

Before, to manipulate an object with a more specific type, you had to use explicit casts, which was cumbersome. Let us say we want to override the `equals(Object obj)` method to compare two `Rectangle` objects. We used to do it like so:

```java
@Override
public boolean equals(Object obj) {
    if (obj instanceof Rectangle) {
        Rectangle rect = (Rectangle) obj;
        return rect.width == this.width && rect.height == this.height;
    }
    return false;
}
```

To be able to manipulate the variable `obj` as a `Rectangle`, we had first to test that the object is really of `Rectangle` type, then to cast it to this given type and finally create a new variable. Using pattern matching, the syntax is much lighter and the code shortened:

```java
@Override
public boolean equals(Object obj) {
    if (obj instanceof Rectangle rect) {
        return rect.width == this.width && rect.height == this.height;
    }
    return false;
}
```

If `obj` is an instance of `Rectangle`, it is cast to `Rectangle` and its value assigned to the new local variable `rect`.

9 Nested classes

9.1 Introducing: nested classes

Java provides the possibility to define *nested classes*: a class defined within another one. A class containing a nested class is called the **enclosing class** of the nested one.

Let's see how it works:

```java
public class Main {
    public static class EnclosingClass {
        public class NestedClass {
            public void myPrint() {
                System.out.println("Nested class!");
            }
        }

        private NestedClass nestedClass = new NestedClass();

        public void myPrint() {
            nestedClass.myPrint();
        }
    }

    public static void main(String[] args) {
        var enclosingClass = new EnclosingClass();
        enclosingClass.myPrint();
    }
}
```

Output:

```
Nested class!
```

9.2 Why would I use it?

You may want to nest a class into another when you want the first one to be used only by the second. Doing so will give your code a stronger logic arrangement, but it will also make it more readable: the code is located right where it is used.

Moreover, where other classes can only be declared either public or default (package-private), a nested class can also be declared private or protected: this will allow you to hide your nested class from the outside.

As nested classes are instance attributes, any of their instances has to exist within an instance of their enclosing class. An instance of an inner class has direct access to any field of their related enclosing class instance, even the private ones.

```
public class Main {
    public static class EnclosingClass {
        private int myInt = 1;

        private class HiddenClass {
            public void myPrint() {
                System.out.println("Nested class can access this: myInt = " + Integer.
↪toString(myInt));
            }
        }

        private HiddenClass hiddenClass = new HiddenClass();

        public void myPrint() {
            hiddenClass.myPrint();
        }
    }

    public static void main(String[] args) {
        var enclosingClass = new EnclosingClass();
        enclosingClass.myPrint();
    }
}
```

Output:

```
Nested class can access this: myInt = 1
```

This property of nested classes encourages encapsulation: if you need a class to have access to the private fields of another one, you can nest the first in the second.

Be careful!

For this very reason, instantiating an inner class outside of its enclosing class is a very bad practice: it practically gives access to the private state of the enclosing class' instance. Besides, this syntax is quite ugly: `enclosingClassInstance.new NestedClass`.

9.3 Shadowing

Beware: if a nested class defines a field with the same name as a field defined in its outer class, the inner field will **shadow** the outer one.

```java
public class Main {
    public static class EnclosingClass {
        private int myInt = 1;

        private class HiddenClass {
            private int myInt = 47;

            public void myPrint() {
                System.out.println("From nested class: myInt = " + Integer.toString(myInt));
            }
        }

        private HiddenClass hiddenClass = new HiddenClass();

        public void myPrint() {
            System.out.println("From enclosing class: myInt = " + Integer.toString(myInt));
            hiddenClass.myPrint();
        }
    }

    public static void main(String[] args) {
        var enclosingClass = new EnclosingClass();
        enclosingClass.myPrint();
    }
}
```

Output:

```
From enclosing class: myInt = 1
From nested class: myInt = 47
```

Worry not, there is a solution to this. You can still access the outer class field, at the cost of a slightly unsettling syntax:

```java
public class Main {
    public static class EnclosingClass {
        private int myInt = 1;

        private class HiddenClass {
            private int myInt = 47;

            public void myPrint() {
                // This syntax `EnclosingClass.this.myInt` allow us to access the first myInt
                System.out.println("From nested class: myInt = " + Integer.
↪toString(EnclosingClass.this.myInt));
            }
        }

        private HiddenClass hiddenClass = new HiddenClass();
```

(continues on next page)

(continued from previous page)

```
        public void myPrint() {
            System.out.println("From enclosing class: myInt = " + Integer.toString(myInt));
            hiddenClass.myPrint();
        }
    }

    public static void main(String[] args) {
        var enclosingClass = new EnclosingClass();
        enclosingClass.myPrint();
    }
}
```

Output:

```
From enclosing class: myInt = 1
From nested class: myInt = 1
```

9.4 Exercise

In this exercise, we will simulate the flow of cash and apple juice that can transit between different places during a party.

The party has a safe for money and a stock for apple juice. When the stock runs low (below 20 bottles), someone is sent to buy 20 bottles at the cost of 2 euros per unit at a local store. If, and only if, the party runs low on money, the organizers will gather the money from the entrance safe. If there is still not enough money to buy new bottles, the party is over.

There are two types of stands at these parties:

- Entrance: a stand that will have its own safe in order to cash in the arriving party goers. An entrance is charged 1 euro. There is a maximum to the number of guests inside.
- Bar: a stand that will serve apple juice. The bar will have its own safe in order to store the money made from the sales and its local stock of apple juice. Each drink will be charged 1 euro and will cost 1 bottle to the bar. When there are no more bottles in the stand and someone is trying to buy, someone is sent to refill at the party stock and will also have the charge to bring all the money the stand earned to the party safe. The refiller must bring back 5 bottles to the stand.

In order to challenge you on your knowledge about inner classes, you will have to respect the following constraints:

- `Bar` and `Entrance` will be two InnerClass of the OuterClass Party, with a default visibility.
- `Entrance` has to have a private int attribute named `cash`, and a boolean method named `accept` that accepts a new guest and takes no parameters.
- `Bar` has to have two private int attributes named `cash` and `stock`, and two void methods: `sell` to sell a bottle for 1 euro, and `refill` to ask the party to refill their stock. These methods take no parameters.
- `Party` has to have two private int attributes named `cash` and `stock`.
- `Party` has to comply to the behavior observed in the given file main.

- If the party has to end because of a lack of money, you have to throw an exception like this `throw new RuntimeException("Sorry we're broke");`

The constructor of the `Party` class takes two arguments: the first is the initial amount of cash in the safe, the second is the maximum of guests.

10 Javadoc

10.1 Presentation

Javadoc is a tool used to produce HTML documentation from comments in the source code. It shares some similarities with Doxygen, so you should be able to handle it easily.

Here are some useful Javadoc tags:

- `@author`: the developer's name
- `@deprecated`: points out the method as deprecated
- `@param`: defines a method parameter
- `@return`: documents the value returned by the method
- `@see`: adds a "See also" reference
- `@since`: version of the software in which the method appeared
- `@throws`: documents an exception thrown in the method
- `@version`: version of the class or method

Read the Javadoc reference to have the full available tags list (the Tag Description part) and some other useful information about the functioning of Javadoc.

Example of how to use Javadoc tags:

```
/**
 * Represents a person
 * @author John Doe
 * @version 2.6
 */
public class Person {
    // ...
}
```

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