

# ***1**science de la programmation*



Java<sup>TM</sup>



## Table of Contents

1. Object-oriented programming (OOP).....	5
1.1. Encapsulation.....	5
1.1.1. Benefits.....	5
1.1.2. Accessibility levels.....	5
1.2. Inheritance.....	7
1.2.1. The "extends" keyword.....	8
1.2.2. Exercise solved.....	8
1.2.3. Redefining methods.....	10
1.2.4. Access to elements of the superclass - <i>super</i> .....	11
1.2.5. Redefining the manufacturer.....	12
1.2.6. Abstract classes.....	13
1.3. Polymorphism.....	14
1.3.1. Polymorphism and method redefinition ( <i>late binding</i> ).....	15
1.3.2. Abstract methods.....	17
1.4. The "instanceof" operator.....	18
1.5. Interfaces.....	19
2. Decentralised version management.....	21
2.1. Operating principle.....	21
2.2. Common operations in NetBeans.....	21
2.2.1. Initialisation.....	21
2.2.2. Upload to a server.....	22
2.2.3. Validation of modifications.....	22
2.2.4. Recovering changes from the server.....	22
2.2.5. Going back.....	22
3. Algorithms.....	23
3.1. Sorting algorithms.....	23
3.1.1. Selection sort.....	23
3.1.2. Bubble sort.....	25
3.1.3. Other sorting algorithms.....	26
3.2. Search algorithms.....	27
3.2.1. Principle of minimum and maximum search.....	27
3.2.2. Principle of linear search.....	28
3.2.3. Principle of binary search.....	29
3.3. Complexity.....	30
3.3.1. Space complexity.....	30
3.3.2. Time complexity.....	30

3.3.3. Landau rating (Big O).....	31
4. JSON exchange format.....	32
4.1. The structure.....	32
4.2. The GSON library.....	33
4.3. The JSON-P library.....	34
5. Data structures.....	36
5.1. Java Collection Interface.....	36
5.1.1. Lists.....	37
5.1.2. Sets.....	41
5.1.3. Association array.....	43
5.1.4. Comparison ArrayList, HashSet and HashMap.....	45
5.1.5. Time complexity.....	46
5.2. Simple linked lists.....	47
5.2.1. Operating principle.....	47
5.2.2. Operations.....	48
5.2.3. Stack.....	56
5.2.4. Queue.....	58
6. Recursion.....	59
6.1. Definition.....	59
6.1.1. Example.....	59
6.2. The termination condition.....	59
6.3. Why use recursion?.....	61
7. Connecting to a database.....	62
7.1. Introduction to JDBC.....	62
7.2. Connection to a MySQL server.....	62
7.3. Executing an SQL query and analysing the results.....	64
7.4. Full source code.....	65
7.5. Statement vs. PreparedStatement (for advanced users).....	66

## Sources

- *"Introduction to object-oriented programming - GTG classes"* by the PrograTG working group of the Commission Nationale des Programmes d'Informatique - ESG.
- *"Introduction to object-oriented programming - GIN classes"* by Fred Faber
- *"Connecting to a database"* by Robert Fisch
- *"The basics of programming"* by Fred Faber
- *Learn Java - Courses and exercises*  
<https://perso.telecom-paristech.fr/hudry/coursJava>
- *The course is constantly being adapted and added to.*

## Editors

- Robert Fisch
- Laurent Haan
- François Thillen

The section on the "Java Collection Interface" and many other ideas and links have largely been taken from the online course on EduMoodle set up by Laurent Haan.

## Reference website

Additional documents are available on the website: <http://java.cnpi.lu>

## 1. Object-oriented programming (OOP)

### 1.1. Encapsulation

**Encapsulation** is the act of hiding the structure and internal workings of an object and making accessible only those elements and operations that are absolutely necessary for its use.

We separate the interface (graphical user interface) from the realisation (implementation). We've already applied the principle of encapsulation when we protected **attributes** and gave access only via an **accessor**. In the same way, **methods** and **attributes** can be hidden inside the class without anyone outside being aware of them.

The principle of encapsulation is the same in other fields, for example in electronics, where the complexity of electronic devices is hidden in a casing, with access to functionality only via buttons, standardised connectors, or remote controls. Users don't need to know all the intricacies of **building** electronic circuits and making connections between different circuit boards, they just need to master what the manufacturers have provided as a **user interface**.

In IT, electronics and mechanical engineering, the design of the encapsulation and user interface determines the user-friendliness and security of objects.

#### 1.1.1. Benefits

Encapsulation has several advantages:

- to hide the complexity of an object and make it easier to use (*information hiding*),
- protect the object against illicit operations from the outside,
- be able to modify the internals of an object without the object's user having to change its code.

#### 1.1.2. Accessibility levels

In Java, encapsulation is achieved through accessibility levels. Up to now, you have been familiar with two levels of accessibility (**private** and **public**) for classes, attributes, and methods. In Java, there are 4 different levels of accessibility:

- **Accessibility by default**

Key word: (none)

UML symbol: ~

If the accessibility level of an element is not specified, all classes in the same **package** have access to this element.

- **Protected access**

Keyword: **protected**

UML symbol: #

If an element is prefixed with the **protected** keyword, all classes in the same package as well as all direct **inheritors** (→ see chapter on inheritance) have access to this element.

- **Private access**

Keyword: **private**

UML symbol: **-**

An element prefixed with the **private** keyword is only accessible from within the class itself (instances of a class still have access to the private elements of other instances of the same class).

- **Public access**

Keyword: **public**

UML symbol: **+**

If an element is marked as **public**, there are no restrictions on its visibility.

### Example

Let the following two classes be in the same package:

```
public class Test
{
    int defaultInt = 0;
    protected int protectedInt = 1;
    private int privateInt = 2;
    public int publicInt = 3;
}
```

+	Test
~	defaultInt : int
#	protectedInt : int
-	privateInt : int
+	publicInt : int

```
public class Launcher
{
    public static void main(String[] args)
    {
        Test test = new Test();
        test.defaultInt = 10 ;           // OK, because same package
        test.protectedInt = 10;          // OK, because same package
        test.privateInt = 10;            // !! ERROR !!!
        test.publicInt = 10;             // OK
    }
}
```

## 1.2. Inheritance

**Inheritance** is a relationship between two classes. **Inheritance** is the ability to define a class by specifying only the differences between it and an existing class. This relationship is called a **generalisation relationship**, or an "is-a" relationship.

In OOP, it is possible for a class **B to inherit from a class A**, which means that class **B** has all the elements of class **A** (without even writing a line of code). Then, of course, elements are added to class **B**, making class **B** more **specific** than **A**. This concept is very common in real life.

**Example: Consider the terms 'person', 'employee' and 'customer'.**

*An employee **is a** person* and therefore has everything that defines a person. However, to say that a person is an employee is more specific and implies that they have additional characteristics (employer, salary, etc.).

- 'Person' is therefore a more general term,
- 'Employee' is a more specific and restrictive term,
- 'Employee' has all the characteristics of 'Person' and
- 'Employee' has additional features.

*A customer **is a** person*, and we can repeat the same observations as for 'employee' and 'person'.

If we need to model a program comprising employees and customers, it is therefore very useful to group together in a **Person** class all the points in common between customers and employees. The **Client** and **Employee** classes will be defined based on **Person**. In this way, you only need to define the specific characteristics that distinguish **Employee** and **Client** from **Person** => you avoid duplicating code.

We can therefore say that:

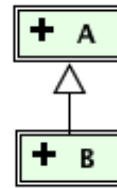
<b>Employee is derived from Person</b>	or	<b>Employee inherits from Person.</b>
<b>Client is derived from Person</b>	or	<b>Client inherits from Person.</b>

In UML, this generalisation relationship is represented as follows:

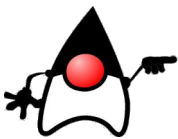


To express this inheritance relationship between **A** and **B** we say that:

- A is the **base class**.
- B is a **specialisation** of A
- A is the **generalisation** of B
- B is a **subclass** of A
- A is the **superclass** of B
- B is the **daughter class** of A
- A is the **parent class** of B



### Remarks



- Provided it is not protected, it is even possible to extend a class for which you do not own the source code!
- With the shortcuts <Alt-F12> in Windows and <Ctrl-F12> in Mac OSX, you can display the inheritance hierarchy of the current class.

#### 1.2.1. The "extends" keyword

To express inheritance in Java, all you must do is add the keyword **extends** to the definition of the subclass, followed by the name of the base class:

```
public class B extends A
{
    ...
}
```

Now **B** contains all the elements of **A** and we can add the attributes and methods that distinguish **B** from **A**.

#### 1.2.2. Exercise solved

Let's say we need to write a program for a small business. We already know that we'll need to implement classes to manage customers and employees. Both have common elements that require attributes, with management methods (surname, first name, address, date of birth, etc.). Instead of repeating the elements in both classes, we gather the elements in a **Person generalisation class**.

We then create the **Client** class by adding 'extends **Person**' behind the declaration. (In Unimozer, you can alternatively use the 'Add Class...' wizard and enter 'Person' in the 'extends' field).

+ <b>Person</b>
# givenName : String
# surName : String
# dateOfBirth : String
# address : String
+ setGivenName(pGivenName : String) : void
+ setSurName(pSurName : String) : void
+ setDateOfBirth(pDateOfBirth : String) : void
+ setAddress(pAddress : String) : void
+ getGivenName() : String
+ getSurName() : String
+ getDateOfBirth() : String
+ getAddress() : String
+ toString() : String

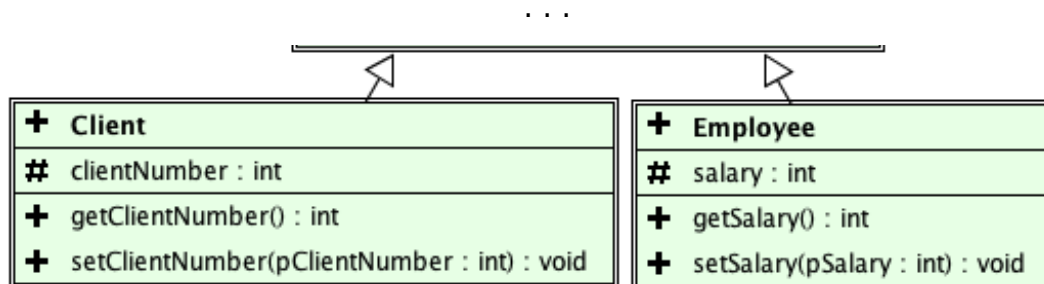


The `toString` method is defined so that it returns text of the form:

```
<givenName> <surName> *<dateOfBirth> (<address>)
```

If you now create a new **Client**, you'll see that it already has the attributes and methods of **Person**. Also add an **Employee** class for employees.

Let's now add the specific elements for customers (customer card no.) and employees (salary) to the **Customer** and **Employee** subclasses.

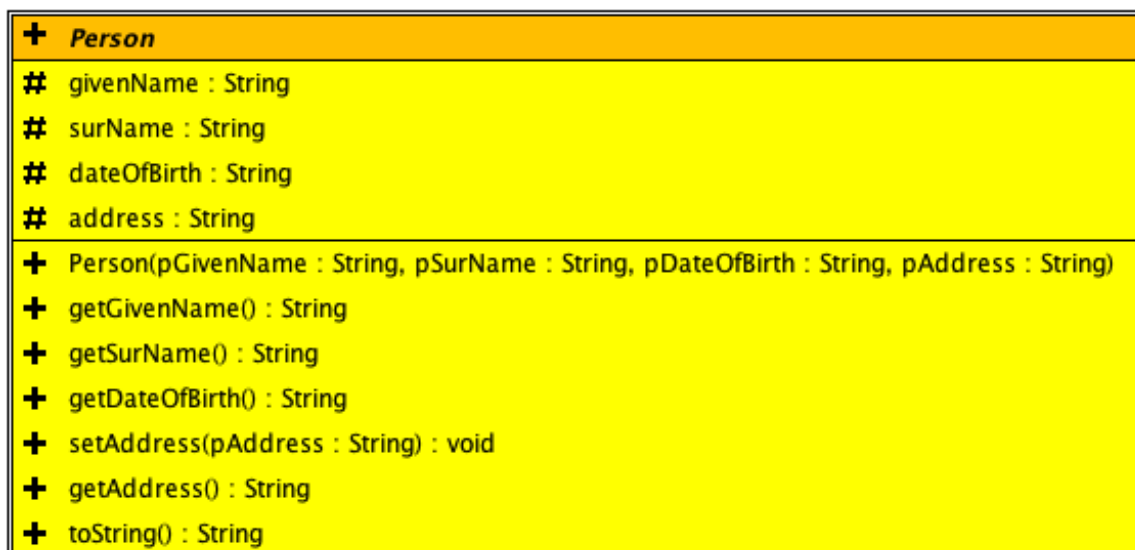


If you now create a new **Client**, you will see that it has the **clientNumber** attribute in addition to the **Person** attributes and methods.

Checking the three classes, we see that the implementation is far from perfect in the spirit of **encapsulation**. In terms of **data accuracy**, it's even possible to create customers without names, dates of birth or customer numbers. What's more, people's names and dates of birth can be changed several times.

The best solution is therefore to add constructors that force us to initialise instances correctly, and then remove the possibility of modifying persistent data (surname, first name, date of birth, customer number). A person's address or an employee's salary can also change later.

First, let's change the **Person** base class:



Before continuing, we would like to make several observations:

- The **toString** method inherited from **Person** is no longer correct (complete) for **Client** and **Employee** (the client number and salary are missing).
- The **Client** and **Employee** constructors will differ from the **Person** builder, but they will have a large part in common.
- In addition, the compiler displays two error messages like '**... cannot find symbol**' as soon as we add a constructor (with parameters) to **Person**. The error messages are displayed at the start of the **Client** and **Employee** classes (even though we haven't changed anything in these classes...).

Let's start by dealing with the first "problem" ...

### 1.2.3. Redefining methods

**Redefining a method** is the act of defining in a subclass a method that already exists **under the same name** (and with the same number and type of parameters) in a base class of the class. In this case, the redefined method of the base class will no longer be available (directly) to instances of the subclass.

#### Please note:

- Methods with the same name but other parameters are not affected by the redefinition.
- In the subclass code we still have access to the redefined methods, but indirectly: i.e., we can still call them by "**super**" (→ see also Chapter 1.2.4), even if they are no longer available directly in the subclass instances.

#### Exercise solved (continued):

Let's redefine the **toString** method for our **Client** class:

```
public String toString()
{
    return givenName + " " + surName + " *" + dateOfBirth
        + "(" + address + ") no.:" + clientNumber;
}
```

Now, a **toString** call from a client will use this new method. The original **Person** method will no longer be available to **Client** instances.

Also adapt **Employee's toString** method.

#### 1.2.4. Access to elements of the superclass - *super*

A class can use inherited elements as if they were in the class itself. In the **Client** class, we can therefore write:

```
System.out.println(givenName + " " surName);  
setAddress("57, rue Bellevue L-3883 Manternach");
```

If, on the other hand:

- we need to call a method of the superclass that exists **under the same name** in the subclass, or
- if we want to call the **constructor** of the superclass,

then we need to place the **super** keyword in front of the method name.

The **super** keyword can be considered as a reference to the super-class.

#### Exercise solved (continued):

When you redefined the **toString** method, you probably noticed that you had to repeat some code that was already in the **Person** class. You can of course copy the code using *copy-paste*, but it would be much more flexible to make a reference to the inherited method instead of copying it.

In the **Client** class, we can write very elegantly:

```
public String toString()  
{  
    return super.toString() + " no.:" + clientNumber;  
}
```

This will be even more important in more complex methods, where each time the code is modified it will be necessary to copy it from the superclass to all the subclasses...

### 1.2.5. Redefining the manufacturer

#### CAUTION: CONSTRUCTORS ARE NOT INHERITED!

**But:** When it is constructed, the subclass automatically calls the default constructor of its superclass (if it exists)!

**Default constructor:** If we don't define a constructor for a class, then the class has a default constructor with no parameters. This default constructor is inherited from the `java.lang.Object` class, which is directly or indirectly the superclass of all classes. The default constructor builds an instance of this class.

As soon as we define our own constructor, it replaces the default constructor. If we define a constructor with parameters in a superclass, there is no longer a constructor without parameters (unless we also define a constructor without parameters). **The subclass can therefore no longer find a default constructor that it can call** and, as a result, it produces an error of the type '`... cannot find symbol`'.

#### Exercise solved (continued):

Finally, we can explain the error messages in our project and remedy them: The error messages in **Client** and **Employee** can be explained by the fact that we replaced the default constructor (without parameters) of the **Person** superclass with a constructor with parameters.

The **Client** and **Employee** constructors must therefore call the **Person** constructor explicitly. To do this, we need the **super** reference. Calling the default constructor of the superclass would simply be done by **super()**, but as the constructor we want to call has parameters, we need to indicate them in brackets behind **super**.

The constructor of a subclass typically has two charges:

1. call the constructor of the super-class,
2. initialise its own attributes (here: `clientNumber`).

In **Client**, we will define the constructor as follows:

```
public Client(String pGivenName, String pSurName, String pDateOfBirth,
String pAddress, int pClientNumber)
{
    super(pGivenName, pSurName, pDateOfBirth, pAddress);           //(1.)
    clientNumber = pClientNumber;                                   //(2.)
}
```

Also define the constructor for **Employee** and test the two classes.

### 1.2.6. Abstract classes

An **abstract class** is a class which is not used to create instances of itself, but only as a generalisation class from which more specialised classes can be derived. In UML, the name of the abstract class is written in italics.

An abstract class is most often used to group together the common elements (methods and attributes) of a set of classes that will later be specialised.

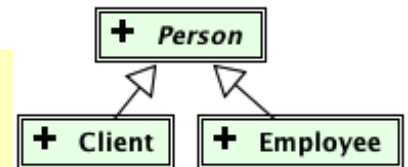
In contrast, a **concrete class** is a class that is used to create instances.

In Java, abstract classes are preceded by the **abstract** keyword so that the compiler recognises them as such and can issue a compilation error when trying to instantiate an object.

#### Exercise solved (continued):

When modelling our small business, the **Person** class must be implemented as an abstract class, since it only serves as a 'common denominator' for the **Client** and **Employee** classes, without ever being used to derive instances of them.

```
public abstract class Person
{
    . . .
}
```



In the UML representation, the **Person** class then appears in italics and a call such as `new Person(...)` will cause a compilation error. (Of course, calls like `new Client(...)` or `new Employee(...)` will still be possible).

### 1.3. Polymorphism

**Polymorphism** is the ability to consider an instance of a class as also being an instance of that class's base classes.

In our example, it is possible to treat an instance of **Employee** as also being an instance of **Person** or even **Object**. This is extremely practical, because you can send an instance of **Employee** to a method that wants an instance of **Object** or **Person** as its parameter. In this way, you can define general methods that work for an entire subtree of the class hierarchy.

Similarly, you can define an attribute or variable of a basic type (e.g., **Object** or **Person**) and assign instances of one of these subclasses (direct or indirect) to it.

```
Person p = new Employee("James", "Black", "12/3/1960", "2323 Downtown NY", 5800);
Object o = new Employee("James", "Black", "12/3/1960", "2323 Downtown NY", 5800);
```

Polymorphism only works in one direction, i.e., an instance of a class cannot be an instance of one of its subclasses. The following assignment is therefore incorrect:

```
Employee e = new Person("James", "Black", "12/3/1960", "2323 Downtown NY"); // FALSE!
```

#### Compatibility of assignments:

A class is compatible with its ancestor classes.

#### Note: Polymorphism and 'real life

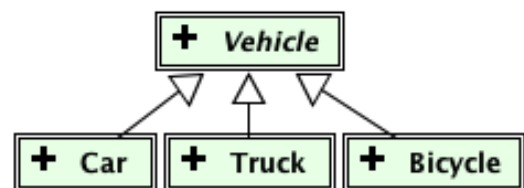
This way of dealing with inheritance fits in well with our conception of things in real life: yes, an employee is a person and can be treated as such, but conversely you can't say that every person is an employee and treat them as such ...

#### Exercise:

Let the classes **Vehicle**, **Car**, **Truck**, **Bicycle** be defined as follows:

What can you say about the **Vehicle** class?

Consider the following statements:



```
Car myCar;
Truck myTruck;
Bicycle myBicycle;
Vehicle myVehicle;
```

Which of the following assignments are compatible?

Which are incompatible, or incorrect for other reasons?

1. Truck t = myVehicle;
2. myVehicle = myBicycle;
3. Vehicle v = myTruck;
4. myCar = new Vehicle();

```
5. Car t = new Truck();  
6. Vehicle v = new Car();
```

### 1.3.1. Polymorphism and method redefinition (*late binding*)

The concept of polymorphism becomes particularly interesting if we also consider that it is possible to redefine methods that have already been defined in base classes (→ see Chapter 1.2.3).

To illustrate this, let's consider the following example from a drawing program:

```
public abstract class Figure  
{    //definition of methods and attributes common to all figures  
    . . .  
    public void draw(Graphics g)  
    {    // ? ? ?  
    }  
}  
  
public class Rectangle extends Figure  
{  
    public void draw(Graphics g)  
    {    //instructions for drawing a rectangle  
        . . .  
    }  
}  
  
public class Circle extends Figure  
{  
    public void draw(Graphics g)  
    {    //instructions for drawing a circle  
        . . .  
    }  
}  
  
public class Line extends Figure  
{  
    public void draw(Graphics g)  
    {    //instructions for drawing a line  
        . . .  
    }  
}  
  
public class RoundedRectangle extends Rectangle  
{  
    public void draw(Graphics g)  
    {    //instructions for drawing a rectangle with rounded corners  
        . . .  
    }  
}
```

We can see that each class derived from **Figure** has redefined the **draw** inherited method so that the objects are drawn individually and correctly. However, we need to ask ourselves what happens if, when applying polymorphism, we proceed as follows:

```
Figure f1 = new Rectangle();  
Figure f2 = new RoundedRectangle();  
f1.draw(g);  
f2.draw(g);
```

**f1** and **f2** are of type **Figure**, but they refer to instances of type **Rectangle** and **RoundRectangle**.

Which **draw** method will then be called, the **Figure** method, the **Rectangle** method or the **RoundRectangle** method?

Java follows the most comfortable principle (for us) by automatically calling the method that corresponds to the nature of the instance (and not that of the reference).

So our call to **f1.draw(g)** will actually produce a rectangle on the screen and the call to **f2.draw(g)** will produce a rectangle with rounded corners.

If now, instead of two figures **f1** and **f2**, you imagine a whole list (*array*, **Vector** or **ArrayList**) of figures, which is filled in disorderly fashion with rectangles, circles, lines etc., you'll be able to appreciate the usefulness of this way of proceeding...!

### Note for advanced users: *late binding vs. early binding*

This way of automatically linking a redefined method to the instance's real type requires a technique known as *late binding* or *dynamic binding*.

Let's explain this using the example of an **ArrayList<Figures> figureList** filled with all sorts of figures: at compile time, we can't yet know which method should be linked to the call to **figureList.get(i).draw()** (the **Rectangle** method, the **RoundRectangle** method, the **Circle** method, etc.). This binding of the **draw** call to the code of the 'correct' method (the one corresponding to the actual type of the instance) only takes place when the program is executed - i.e., very late and dynamically. In principle, Java uses the principle of *late binding* for any call to an instance method.

When the compiler can resolve addresses during compilation (e.g., the addresses of attributes or static methods), this is called *early binding* or *static binding*.



### 1.3.2. Abstract methods

If we look closely at the example in chapter 1.3.1, we notice an inconsistency in the definition of the **Figure.draw** method. What is the point of the **Figure.draw** instruction block? This method is completely redefined in each subclass and there isn't a single useful line of code that we could write in its instruction block. However, we can't delete it, otherwise the polymorphic method redefinition mechanism (described in the previous chapter) wouldn't work.

In Java, such a method is defined as an abstract method.

An **abstract method** is a method which is not implemented, but which is declared solely so that it can be overloaded in derived classes.

**abstract:** keyword marking a method whose implementation is transferred to its descendant classes.

**Please note:**

- An abstract method has no instruction block, but its declaration is followed by a semi-colon.
- An abstract method can only be found in an abstract class.
- Each descendant class must redefine the abstract method.

In our example:

```
public abstract class Figure
{
    //definition of methods and attributes common to all figures
    . . .

    public abstract void draw(Graphics g);
}
```

## 1.4. The "instanceof" operator

This operator is used to test whether an object is an instance of a given class.

### **Example**

Let's take the following code:

```
!! Animal myAnimal = new Cat();

if(myAnimal instanceof Animal)
{
    System.out.println("It is an animal ...");
}
if(myAnimal instanceof Cat)
{
    System.out.println("It is a cat ...");
}
if(myAnimal instanceof Dog)
{
    System.out.println("It is a dog ...");
}
```

After execution, the code displays that **myAnimal** is an animal and that **myAnimal** is a cat.

If you apply the OOP principle correctly, you generally don't need the instanceof operator. However, there are still some special cases where its use is essential or even simply practical.

## 1.5. Interfaces

In some programming languages (e.g., C++), a class can inherit from several base classes. This is known as **multiple inheritance**. Because of the complexity of such an architecture and the difficulties that can arise from it (e.g., inheriting a method with the same name from several different classes/diamond inheritance), the creators of Java decided not to implement multiple inheritance in Java.

However, we thought it would be useful to implement the possibility that several classes with completely different hierarchical structures could inherit the same definitions for their **interfaces**.

From a programming point of view, an interface is a special type of a class, which contains only **abstract methods** (*public abstract*) and **constants** (*public static final*). The definition of attributes or constructors is not permitted. When defining a class, the keyword **class** is replaced by **interface**. The fact that a class inherits or **implements an interface** is indicated by the keyword **implements** (instead of **extends**). A class can inherit from a base class and implement several interfaces at the same time. If a class implements an interface, it can be treated as if it were a subclass of the interface.

From a conceptual point of view, an interface can be seen as a contract or a 'promise': by using the **implements** keyword, a class undertakes to implement or redefine all the methods prescribed by the interface. Each interface has a different purpose and specific features. For the exact specifications of interfaces and the exceptions they are supposed to throw, see **JavaDoc**.

The usefulness of interfaces becomes clearer by looking at a few examples. The most common interfaces are:

**Comparable:** allows instances of a class to be compared using **compareTo(...)**

**Serializable:** saves instances of a class in a file (or other persistent media)

**Iterable, Iterator, ListIterator:** used to browse a **collection** of objects (using **hasNext()**, **next()**) without knowing exactly how the objects are stored (see collections: **Map**, **List**, **ArrayList**, etc.). Optionally, the iterator can define the **remove()** method, which is used to delete objects. Iterators are often used to browse the elements of a collection using the 'for-each' structure.

**Cloneable:** allows instances of a class to be copied exactly using **clone()**. Special case: **.clone** is already defined in **Object**, but **implements Cloneable to ensure that** a correct copy (of all fields) is guaranteed. A class implementing **Cloneable** generally has a public **.clone** method which redefines the inherited **.clone** method.

GUI programming in Swing uses the **Observer** interface or the famous **...Listener** or **...Adapter**.

Remarks:

- Interfaces can also inherit from another interface to extend it, e.g., to add another method to the interface.
- It is possible to create an instance of an interface (→ see code generated by Net-Beans...).
- Interfaces are often defined as **parameterised types** (EN: **generics**), by indicating the class name between '<' and '>'. For example **implements Iterable<Shape>** indicates that the class can provide an iterator that returns elements of type **Shape**.

Learn more about interfaces in **JavaDoc** and the following documents:

- <http://download.oracle.com/javase/tutorial/java/IandI/createinterface.html>
- <http://www.codestyle.org/java/faq-Interface.shtml>

Example:

Let's imagine we have a project with several hierarchies of classes and objects. We'd like all the objects to be able to print their current information on the screen.

A first idea might be to define an abstract **Printable** class and derive all the objects in our project from it. However, this would be rather artificial, and would destroy the logic of the inheritance of the classes in our project. What's more, if we use objects derived from predefined classes in our project, it wouldn't be possible to insert our **Printable** class into the inheritance hierarchy.

The solution is to define a **Printable** interface containing the declaration of an abstract **print()** method. We add the **implements Printable** instruction to all the classes in our project and implement the **print()** method in all the classes. In this way, the original hierarchy of our classes is not affected, but all the objects have the ability to print their information on screen.

Aspect of the "Multiple Inheritance ":

We can treat all objects that implement **Printable** as if they inherited not only from their superclass, but also from the **Printable** class. So we can go through a list of completely different objects that implement 'Printable' and call their **print()** method - just as if they all inherited from that class!

Each class has only one parent class, so we avoid all the difficulties of true multiple inheritance.

## 2. Decentralised version management

According to Wikipedia<sup>1</sup>, *version control consists of managing all the versions of one or more files (usually text files). Essentially used in software development, it is mainly concerned with the management of source code.*

One well-known system is the "Git" system. Git is decentralised version management software. It is free software created by Linus Torvalds, author of the Linux kernel, and distributed under the terms of the GNU General Public License version 2. In 2016, it was the most popular version management software, used by more than twelve million people.

### 2.1. Operating principle

See on the following pages:

- <https://fr.wikipedia.org/wiki/Git>
- <https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/>
- <https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Rudiments-de-Git>
- <http://adopteungit.fr/methodologie/2017/07/02/git-les-principes-de-base-pour-le-prendre-en-main.html>
- <https://karac.ch/blog/maitriser-essentiel-de-git-en-quelques-minutes>

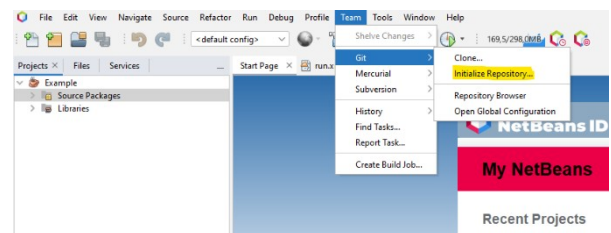
### 2.2. Common operations in NetBeans

#### 2.2.1. Initialisation

To put a project under Git in NetBeans:

- select the project,
- click on the menu:

**Team > Git > Initialise Repository**



At the end of this operation, your project has been placed under Git, but only locally. This allows you to take advantage of all Git's features, without sharing them with others.

Then you need to make an initial commit using the:

**Team > Commit**

This will place all the project files under Git. If you don't want any of the files to be placed there, take the action of your choice before confirming the action.

<sup>1</sup>[https://fr.wikipedia.org/wiki/Gestion\\_de\\_versions](https://fr.wikipedia.org/wiki/Gestion_de_versions)

### 2.2.2. Upload to a server

To upload local changes to the server, go to the menu:

***Team > Remote > Push***

The screen that appears has three sections:

1. A first one that includes information about the remote server on which you want to upload the project. The first time you run this operation, you need to specify a new "Git Repository Location". This will be indicated to you by your teacher.
2. On the second screen, you need to select the branch you want to upload. As a Git beginner, you probably only have one.
3. The third screen shows the selection of the server branch on which you want to perform an update.

Confirm the action by clicking on "Finish".

### 2.2.3. Validation of modifications

Once you've made a major change to your project, you'll probably want to validate it. This is done via the:

***Team > Commit***

When you do this, NetBeans shows you the files that have been modified as well as those that have been deleted or added.

If you want to upload your changes to the server, do the following:

***Team > Remote > Push***

### 2.2.4. Recovering changes from the server

If someone else has updated the code on the server and you want to update your local copy, you need to go to:

***Team > Remote > Pull***

### 2.2.5. Going back

If you want to cancel the changes you have made since your last validation, use the menu:

***Team > Revert Modifications ...***

Sometimes you want to go further back. In this case, use the menu:

***Team > Checkout > Checkout Revision ...***

to restore an older validated state.

## 3. Algorithms

### 3.1. Sorting algorithms

#### 3.1.1. Selection sort

FR: Tri par sélection directe, EN: Selection Sort, DE: Sortieren durch direkte Auswahl

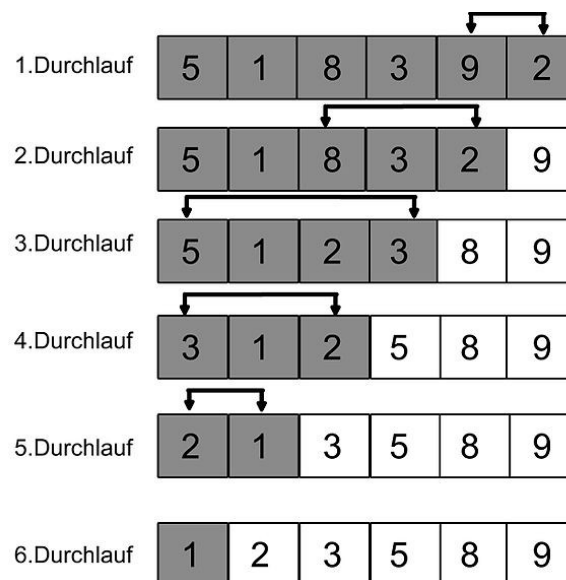
Selection sort is a comparison sort algorithm. This algorithm is simple, but considered inefficient, because it executes in quadratic time depending ( $O(n^2)$ ) on the number of elements to be sorted, and not in pseudo-linear time.<sup>2</sup> The chapter on the complexity of algorithms will contain details about execution time.

##### **Problem:**

Sort the elements of a list in ascending order (DE: *aufsteigend*) using in-place comparison sorting method.

##### **Method:**

Go through the unsorted part of the list looking for the smallest element and, if necessary, swap this element with the first element in the unsorted part. In this way, the sorted part is increased by one element while the unsorted part is reduced. By applying this principle as many times as the size of the list minus one, the list will be completely sorted.



<sup>2</sup>[https://en.wikipedia.org/wiki/Selection\\_sort](https://en.wikipedia.org/wiki/Selection_sort)

**Example:**

Consider the array of integers below. At the beginning there is no sorted part, and the unsorted part consists of the whole array. The smallest element in the unsorted part is at position #1 and will therefore be swapped with the first element in the unsorted part, which here is 5 at position #0.

#0	#1	#2	#3	#4
5	1	8	2	4

↑

Applying the principle again to the unsorted part, we find that the smallest element is at position #3 and will have to be swapped with the one at position #1.

#0	#1	#2	#3	#4
1	5	8	2	4

↑

In the example, the list is made up of 5 elements, so we need to re-apply the search for the minimum followed by an exchange 4 times. Because 4 elements have been placed in the right place, the last element must also be in the right place.

#0	#1	#2	#3	#4		#0	#1	#2	#3	#4		#0	#1	#2	#3	#4
1	2	8	5	4	>	1	2	4	5	8	>	1	2	4	5	8
			↑						↑							

**Example code:**

```
// external path (from the first to the penultimate element)
for(int i=0; i<alNumbers.size()-1; i++)
{
    // find the position of the minimum
    int posmin = i;
    // run from next element to end
    for(int j=i+1; j< alNumbers.size(); j++)
    {
        // compare the two elements to determine the smaller of the two
        if(alNumbers.get(j) < alNumbers.get(posmin))
            posmin=j;
    }

    // exchange -- if necessary
    if(posmin!=i)
    {
        int tmp = alNumbers.get(i);
        alNumbers.set(i, alNumbers.get(posmin));
        alNumbers.set(posmin,tmp);
    }
}
```



### 3.1.2. Bubble sort

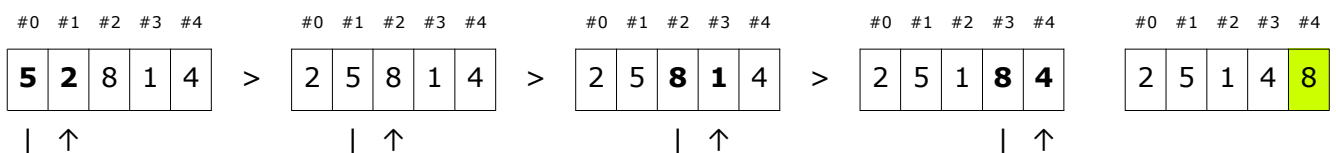
FR: Tri à bulles, EN: Bubble sort, DE: Bubblesort

Bubble sorting or sinking sorting is a sorting algorithm that consists of repeatedly comparing consecutive elements in an array/list and permuting them when they are incorrectly sorted. It gets its name from the fact that it quickly moves the largest elements to the end of the array, like air bubbles rising rapidly to the surface of a liquid.

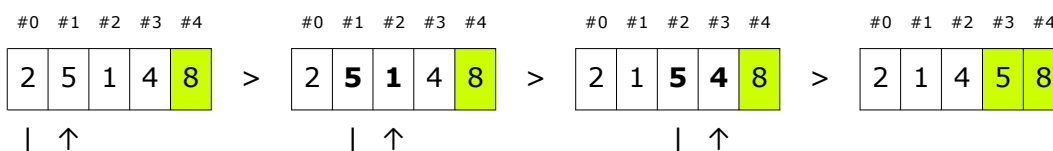
Bubble sort is often taught as an algorithmic example because its principle is simple. But it is the slowest of the commonly taught sorting algorithms, so it is rarely used in practice.<sup>3</sup>

#### Example:

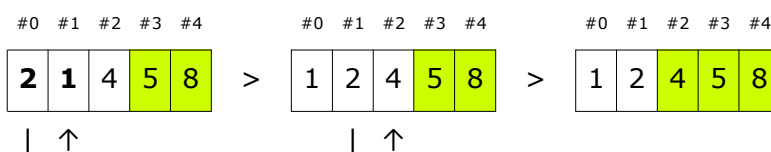
Consider the array of integers below. At the beginning there is no sorted part, and the unsorted part consists of the whole array. We start a run from the beginning of the list to the penultimate element of the unsorted part, comparing the current element with its next. If the current element is larger than the next one, the two are swapped. So, after the first run, the largest element is at the end of the list.



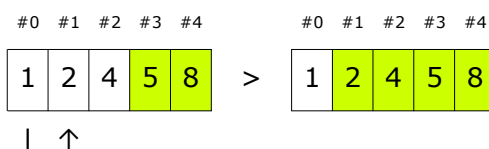
As the sorted part has grown, there is one less comparison to make during the next run.



We continue ...



... until the list is completely sorted.



<sup>3</sup>[https://fr.wikipedia.org/wiki/Tri\\_à\\_bulles](https://fr.wikipedia.org/wiki/Tri_à_bulles)

**Example code :**

```
// from the penultimate element to the first
for(int i=alNumbers.size()-1; i>=0; i--)
{
    // route the unsorted part
    for(int j=0; j<i; j++)
    {
        // compare the current element with the one after it
        // to exchange it if it's smaller
        if(alNumbers.get(j) < alNumbers.get(j+1))
        {
            Person tmp = alNumbers.get(j);
            alNumbers.set(j, alNumbers.get(j+1));
            alNumbers.set(j+1, tmp);
        }
    }
}
```

**3.1.3. Other sorting algorithms**

There are a multitude of other sorting algorithms. There are a variety of videos showing how they work and their performance. Here is an example:

<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

If you watch this video, you'll notice that the two previous sorting algorithms are generally the slowest and not very efficient when it comes to sorting large numbers of items.

In Java, there is an "internal" possibility of sorting lists of elements, namely the **Collections.sort()** method. It uses 'merge sort' internally, which is a very stable sort algorithm with a guaranteed complexity of  $O(n \cdot \log(n))$  (→ see chapter 3.3 Complexity).

**Example:**

The **sort()** method needs to be passed the name of the list to be sorted and a comparator. The comparator is used to determine whether one element of the list is larger or smaller than another. For a list of integers, this would be:

```
Collections.sort(alNumbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1<o2;
    }
});
```

On the other hand, if we wanted to sort a list of people by name, for example, the call to the **sort()** method would look like the following code:

```
Collections.sort(alPersons, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
```

## 3.2. Search algorithms

### 3.2.1. Principle of minimum and maximum search

To determine the minimum or maximum of an (unsorted) list, we start by storing its first element - assuming that this is the value we are looking for - and then comparing this value with all the other values in the list. Each time a more appropriate value is found, it replaces the one previously stored.

**Example:**

```
public Integer getMin(ArrayList<Integer> alNumbers)
{
    // stop immediately if the list is empty
    if(alNumbers.isEmpty())
        return null;

    // assume that the first value is correct
    int result = alNumbers.get(0);

    // test all other values
    for(int i=1; i<alNumbers.size(); i++)
        // if a more suitable value is found
        if(alNumbers.get(i)<result)
            // memorise this one
            result=alNumbers.get(i);

    // return the result
    return result;
}
```

### 3.2.2. Principle of linear search

FR: recherche séquentielle, EN: linear search, DE: lineare Suche

In a linear search, you go through a list sequentially, starting with the first (or last) element, then moving on element by element, testing each time whether the current element matches the search criterion.

There are several different cases:

- Browse through all the elements in the list, storing and returning the last element found.
- Browses the elements of the list, stopping and returning to the first element found.

#### **Examples:**

```
public Person findByName(ArrayList<Person> alPersons, String name)
{
    // initialise the result
    Person result = null;

    // list path
    for(int i=0; i<alPersons.size(); i++)
        // if the search criterion matches
        if(alPersons.get(i).getName().equals(name))
            // memorise the object found
            result = alPerson.get(i);

    // return the result
    return result;
}
```

```
public Person findByName(ArrayList<Person> alPersons, String name)
{
    // list path
    for(int i=0; i<alPersons.size(); i++)
        // if the search criterion matches
        if(alPersons.get(i).getName().equals(name))
            // return found object
            return alPerson.get(i);

    // no object found :-(
    return null;
}
```

### 3.2.3. Principle of binary search

FR: recherche dichotomique, EN: binary Search, DE: binäre Suche

In computer science, binary search is a search algorithm that finds the position of a target value within a **sorted** array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array<sup>4</sup>.

#### For example:

Let's suppose we have a list of names.

The list must be sorted lexicographically by name.

Initially, the search area corresponds to the entire list.

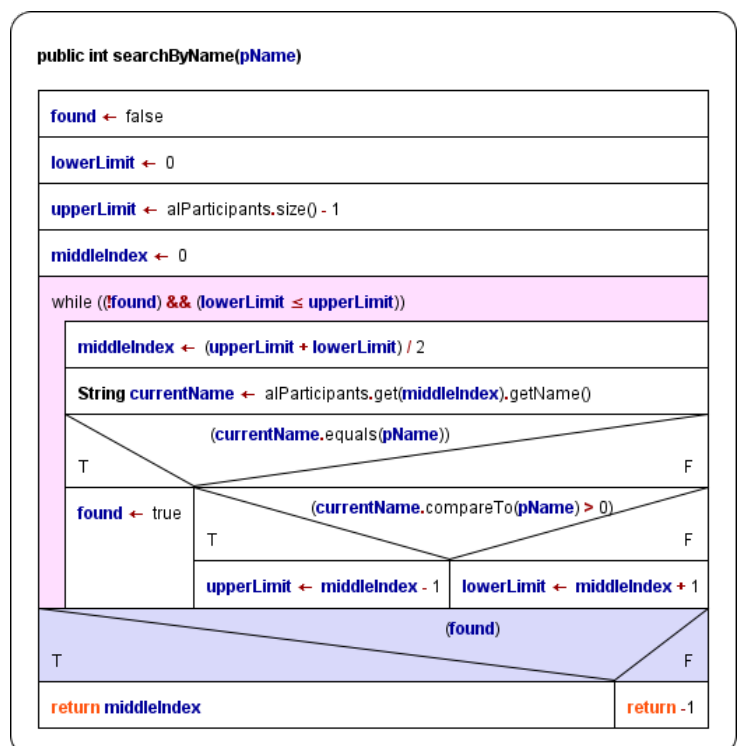
Each time it passes, the search area is reduced to half its size.

On each pass, the name to be searched is compared with the name in the middle of the search area:

- if there is a tie or if the list is exhausted, processing is stopped and the current position is returned as the result (if the list is exhausted, -1 is returned),
- if the name searched for precedes (lexicographically) the name at the current position, the search continues in the half of the search area to the left of the current position,
- if the name searched for follows (lexicographically) the name at the current position, the search continues in the half of the search area to the right of the current position.

#### Example:

Suppose `alParticipants` is a list containing participants sorted in ascending order by name.



<sup>4</sup> [https://en.wikipedia.org/wiki/Binary\\_search\\_algorithm](https://en.wikipedia.org/wiki/Binary_search_algorithm)

### 3.3. Complexity

Complexity theory is the field of mathematics, and more specifically of theoretical computer science, which formally studies the computation time, memory space (and more marginally the size of a circuit, the number of processors, the energy consumed...) required by an algorithm to solve an algorithmic problem. The aim is therefore to study the intrinsic difficulty of problems, organize them into complexity classes and study the relationships between complexity classes.<sup>5</sup>

#### 3.3.1. Space complexity

In algorithms, space complexity is a measure of the space used by an algorithm, expressed as a function of the size of the input. Space counts the maximum number of memory cells used simultaneously during a calculation. For example, the number of symbols that need to be retained to continue the calculation.

Usually, the space considered when talking about the space required for inputs of size  $n$  is the largest space for inputs of that size; this is referred to as worst-case space complexity. Most complexity studies focus on asymptotic behaviour, when the size of the inputs tends towards infinity, and Landau's large  $O$  notation is commonly used.

Another measure of complexity is time complexity.<sup>6</sup>

#### 3.3.2. Time complexity

This is the same as space complexity, with the difference that for time complexity the study is carried out in relation to the time needed by an algorithm and not in relation to the memory it needs to run.

Since time complexity is the most common measure in algorithms, we sometimes simply talk about the complexity of an algorithm.

Analysing the complexity of an algorithm can become very complex, so much so that it constitutes a separate discipline.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/Computational\\_complexity\\_theory](https://en.wikipedia.org/wiki/Computational_complexity_theory)

<sup>6</sup>[https://en.wikipedia.org/wiki/Space\\_complexity](https://en.wikipedia.org/wiki/Space_complexity)

### 3.3.3. Landau rating (Big O)

In computing, we use the symbol  $O$ , i.e., the Landau concept (introduced by the German mathematician Edmund Landau, who specialised in number theory) to note the time complexity of an algorithm in relation to the number of inputs  $n$ .

#### **Examples:**

- |                              |   |             |
|------------------------------|---|-------------|
| • access a value in an array | $O(1)$                                      | constant    |
| • linear search              | $O(n)$                                      | linear      |
| • binary search              | $O(\log_2(n)) = O(\log(n))$                 | logarithmic |
| • selection sort             | $O(n^2)$                                    | quadratic   |
| • bubble sort                | $O(n^2)$                                    | quadratic   |
| • quick sort                 | $O(n \cdot \log_2(n)) = O(n \cdot \log(n))$ | logarithmic |

Please note that indicating the base of the log function is of no importance for time complexity, as the base maybe changed, resulting in a simple constant difference – which will then be ignored for big O notation.

Let's change log base 2 into log base 10:

$$\log_2(x) = \log_{10}(x) \cdot \log_{10}(2)$$

As  $\log_{10}(2)$  is a constant, it will be ignored for time complexity notation.

Thus we can say that:

$$O(\log_2(x)) = O(\log_{10}(x)) = O(\log(x))$$

## 4. JSON exchange format

JSON (JavaScript Object Notation) is a lightweight data exchange format. It is easy for humans to read and write, but also simple for a machine to interpret or generate. It is based on a subset of the JavaScript programming language standard ECMA-262, 3rd edition - December 1999. JSON is a completely language-independent text format that uses conventions well known to programmers in the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data exchange language.

More information can be found on the official JSON website: <http://json.org/>

### 4.1. The structure

In simple terms, JSON is made up of **objects**, **lists** and **values**.

**Value** means

- a numerical value,
- a textual value,
- an **object**,
- a **list**,
- or one of the values: **true**, **false** or **null**.

A **list is** a set of values separated by commas and surrounded by square brackets.

#### Examples

```
[3, 27, 34, 567, 3, 2]           // list of integers
["Abba", "Queen", "AC/DC", "Nirvana"] // list of text values
```

As JSON is a data exchange format, the representation of an **object** does not contain methods, only attributes. An object is noted between braces in the following way:

```
{ "attribute 1": "text value", "attribute 2": numeric value, ... }
```

**Example of** the representation of a person

```
{
  "name":           "MUSTERMANN",
  "firstname":      "Max",
  "age":            25,
  "ownsCar":        true,
  "preferedGroups": ["Abba", "Queen", "AC/DC", "Nirvana"]
}
```

As you can see from this example, the various structures can be nested within each other. Just as an object can contain a list, it is also possible to create lists of objects... and so on.



## 4.2. The GSON library

The GSON library, developed by Google, can be used to serialise and de-serialise whole objects to or from JSON text. It is available at the following address:

<https://search.maven.org/remotecontent?filepath=com/google/code/gson/gson/2.10.1/gson-2.10.1.jar>

Or more simply: <http://java.cnpi.lu/download.php?id=521>

To continue with the previous example, let's take the following class:

```
public class Person {
    private String name;
    private String firstname;
    private int age;
    private boolean ownsCar;
    private ArrayList<String> preferredGroups = new ArrayList<>();

    public Person(String name, String firstname, int age, boolean ownsCar)
    {
        this.name = name;
        this.firstname = firstname;
        this.age = age;
        this.ownsCar = ownsCar;
    }

    // + setters & getters
}
```

Now we create a person as follows:

```
Person p = new Person("MUSTERMANN", "Max", 25, true);
p.addGroup("Abba");
p.addGroup("Queen");
p.addGroup("AC/DC");
p.addGroup("Nirvana");
```

The **p** object can be automatically transformed into text as follows:

```
// create the gson object
Gson gson = new Gson();
// transformation of the person
String json = gson.toJson(p);
```

The reverse transformation is as simple as this:

```
String json = "{ \"name\": \"MUSTERMANN\", \"firstname\": \"Nico\", \"age\": 25, \"ownsCar\": true, \"preferredGroups\": [ \"Abba\", \"Queen\", \"AC/DC\", \"Nirvana\" ] }";
// create the gson object
Gson gson = new Gson();
// create a Person object from JSON text
Person p = gson.fromJson(json, Person.class);
```

Note that using the GSON library is much easier than using JSON-P. However, any automation on one side implies a loss of control on the other.

### 4.3. The JSON-P library

Another library for processing (analysing, generating, transforming, and querying) texts in JSON format is the Java API for JSON Processing. It gives you more control over what is generated and, as a result, has fewer automatic functions than the GSON library.

This chapter is included in the course **for information purposes** only and will not be covered in the course!

It is available at the following address:

<https://javaee.github.io/jsonp/>

A pre-compiled version is available here:

<http://java.cnpi.lu/download.php?id=520>

Here is an example of code used to generate JSON text:

```
// create a JsonObjectBuilder
JsonObjectBuilder job = Json.createObjectBuilder();

// add attributes
job.add("name", "MUSTERMANN");
job.add("firstname", "Max");
job.add("age", 25);
job.add("ownsCar", true);

// creation of the JSON object
JsonObject jo = job.build();

// create JSON text
String json = jo.toString();
```

At the end, the `json` variable will contain the following text:

```
{"name": "MUSTERMANN", "firstname": "Max", "age": 25, "ownsCar": true}
```

In addition to the `JsonObjectBuilder`, there is also the `JsonArrayBuilder`, which can be used to create a list.

```
// createArrayBuilder
JsonArrayBuilder jab = Json.createArrayBuilder();

// add items
jab.add("Abba");
jab.add("Queen");
jab.add("AC/DC");
jab.add("Nirvana");

// create the JSON list
JsonArray ja = jab.build();
```

Taking the example on the next page, we could add the following line to the "add attributes" block in the example of creating a JSON object, in order to add the list of groups preferred by Max MUSTERMANN:

```
job.add("preferredGroups", ja);
```

The corresponding JSON text would therefore be:

```
{"name": "MUSTERMANN", "firstname": "Max", "age":25, "ownsCar":true, "preferredGroups":["Abba", "Queen", "AC/DC", "Nirvana"]}
```

To interpret an existing JSON text, which is the opposite operation, there are two possibilities:

1. a sequential run through the data structure with event triggering,
2. interpretation of the entire data structure and creation of a similar structure in memory.

The first method is faster and uses less memory, while the second gives us much simpler access to the data. That's why we're only going to deal with the second.

To do this, proceed as follows:

```
String json = "{ \"name\": \"MUSTERMANN\", \"firstname\": \"Max\", \"age\": 25, \"ownsCar\": true }";

// creation of a JsonReader from a text
JsonReader jsonReader = Json.createReader(new StringReader(json));

// read the JSON object
JsonObject jo = jsonReader.readObject();

// extract data by direct access to fields
String name = jo.getString("name");
String firstname = jo.getString("firstname");
int age = jo.getInt("age");
boolean ownsCar = jo.getBoolean("ownsCar");
```

By loading the JSON text with the list of preferred groups, you could access it, for example, as follows:

```
// extract from the list of favourite groups
JsonArray ja = jo.getJsonArray("preferredGroups");

// browse the list and display it in the console
for(int i=0; i<ja.size(); i++)
{
    System.out.println(ja.getString(i));
}
```

## 5. Data structures

In programming, it is often necessary to manipulate not only individual values, but also groups of values. For example, an address book management program needs to be able to manage any, and a priori unknown, number of addresses.

In Java, arrays provide a way of storing an arbitrary number of objects. But arrays are not ideal in all situations, for a variety of reasons. For example, the fact that their size is fixed at creation time makes them difficult to use when the number of elements to be stored varies.

As a result, we need ways other than arrays to store and organise groups of objects. In Java, the standard library provides a set of classes for this purpose, including the **ArrayList** class, which we already use frequently.

### 5.1. Java Collection Interface

A collection, or data structure, is an object that serves as a container for other objects. For example:

- paintings,
- lists,
- sets,
- association tables.

Each type of collection has its own characteristics, strengths, and weaknesses. The choice of collection to use in a particular case therefore depends on what you want to do with it.

We will be looking at the following three types of collection, the most frequently encountered in practice, not only in Java but in most of today's programming languages:

- **Lists**, an ordered collection in which a given element may appear several times,
- **Sets**, an unordered collection in which a given element can appear at most once,
- **Maps**, collections associating values with keys.

For each of these three types of collection there are several different implementations. For example, a list can be implemented by means of an array in which the elements are stored side by side, or by means of nodes chained together via references.

Different implementations of a collection generally make different trade-offs, which mean that a given implementation will be best in some situations, but not in all. The choice of implementation is therefore determined by how the collection is used. It is therefore important to be familiar with the characteristics of the available implementations.

### 5.1.1. Lists

#### 5.1.1.1. The "List" interface

Lists are very similar to arrays, so much so that the difference between the two is often blurred. However, arrays are generally of fixed size and random access, whereas lists are generally of variable size and sequential access<sup>7</sup>.

The list concept is represented in the Java library by the **List** interface in the **java.util** package.

The Java library offers two main implementations of the List interface, namely list arrays (**ArrayList** class) and linked lists (**LinkedList** class).

The choice of one or other of these implementations in each situation depends on how the list is used. The table below, which compares the complexities of the most frequent operations for the two implementations, can serve as a guide.

Operation	<b>ArrayList</b>	<b>LinkedList</b>
add, remove	$O(n)$	$O(1)$
get, set	$O(1)$	$O(n)$

**But beware: the complexities given above are only valid in very specific cases!**

Adding or removing an element from a chained list is  $O(1)$  only if it is not necessary to go through the list first to access the point at which it is added or removed. In practice, this is only true if the **add** and **remove** methods of a list iterator<sup>8</sup> are used, or if the addition or removal is done at the beginning or end of the list (first or last element).

On the other hand, adding and deleting an element in a list-array is  $O(1)$  when done at the end of the list.

#### 5.1.1.2. ArrayList

The **ArrayList** class can be used to store an ordered list of any number of objects. However, it allows you to specify the type of object you want to place in it. It implements the **List** interface. A list is an ordered, dynamic (and therefore variable-size) sequence of objects.

The **ArrayList** class is contained in the **java.util** package. It must therefore be imported before it can be used.

##### **Example:**

```
// before you can use it, you need to import the ArrayList class
import java.util.ArrayList;

...

// declaration of a list of persons (public class Person)
```

<sup>7</sup>Random access means that an element whose index is known is accessed in  $O(1)$ , whereas sequential access means that the same operation is performed in  $O(n)$ .

<sup>8</sup> Not part of the official programme

```
ArrayList<Person> alList = new ArrayList<>();
```

The generic syntax for declaring a list is as follows:

```
ArrayList<item_class> name = new ArrayList<>();
```



This is where you define the type of elements in the list.

So, we can set the type of elements in the list ourselves. The **ArrayList** class is therefore a kind of **template** (DE: **Vorlage**) that can be applied to another class. For readability, we will prefix our lists with an "al".

Since Java 7, it is no longer necessary to specify the type of the elements contained in the list when calling the constructor. It is therefore sufficient to indicate **<>** when initialising.

The size of such a list is dynamic, i.e., it adapts automatically according to the number of items placed in it. When the list is created, it is of course empty.

#### Examples:

The following line creates an **ArrayList** of starting size 1. In Java, the default size of an **ArrayList** is 10.

```
ArrayList<String> alDemoList = new ArrayList<>(1);
```

alDemoList =

--

```
alDemoList.add("Hello");
```

```
alDemoList.add("World!");
```

alDemoList =

Hello	World!
-------	--------

The size of the array list was automatically adjusted.

```
alDemoList.add("Some");
```

```
alDemoList.add("1 2 3");
```

alDemoList =

Hello	World!	Some	1 2 3
-------	--------	------	-------

#### 5.1.1.3. Lists of primitive types

If you want to create lists containing elements of primitive types, such as **int** or **double**, you **must** use the envelope classes!

Declaration of a list of integers:

```
ArrayList<Integer> alListName = new ArrayList<>();
```

Declaration of a list of decimal numbers:

```
ArrayList<Double> alListName = new ArrayList<>();
```

#### 5.1.1.4. Methods

The **ArrayList** class has several methods that allow us to manipulate the list. These are the ones we'll be using in this course:

Method	Description
<b>boolean add(Object)</b>	Adds an element to the list (always returns <b>true</b> - the result is generally ignored).
<b>void clear()</b>	Clears the list by deleting all items.
<b>boolean contains(Object)</b>	Tests whether the list contains a given element.
<b>Object get(int)</b>	Returns the element at the position specified in the parameter.
<b>int indexOf(Object)</b>	Returns the position of the element specified in the parameter, or -1 if the element is not included in the list.
<b>Object remove(int)</b>	Deletes the element at the position specified in the parameter. Elements following the deleted element automatically move one position forward. (Returns the deleted element as the result).
<b>Object set(int, Object)</b>	Replaces the element at the specified position with the element passed as parameter (returns the replaced element as result).
<b>int size()</b>	Returns the size of the list, i.e. the number of elements it contains.
<b>boolean isEmpty()</b>	Tests if the list is empty (i.e. test identical to <b>size()==0</b> )
<b>Object[] toArray()</b>	Converts the list into an " <b>Array</b> ". This method is often used for displaying elements with a " <b>JList</b> " component.

#### Remarks:

- The elements of an **ArrayList** are indexed from **0**.
  - E.g.: If a list contains 10 elements (**size()==10**) then these elements have the indices (positions): 0, 1, 2, ... , 9.

**Examples:**

The following list:

```
ArrayList<String> allList = new ArrayList<>();
```

Effect	Code
Add the names "Jean", "Anna" and "Marc" to the list.	<code>allList.add("John");</code> <code>allList.add("Anna");</code> <code>allList.add("Marc");</code>
Delete the first number in the list (the one at position 0).	<code>allList.remove(0);</code>
Save the number of elements in the <b>count</b> variable.	<code>int count = allList.size();</code>
Save the position of the "Marc" element in the variable <b>pos</b> .	<code>int pos = allList.indexOf("Marc");</code>
Replace the element at position <b>pos</b> (here the name "Marc") with the name "Michelle".	<code>allList.set(pos, "Michelle");</code>
Save the last item in the list in the <b>last</b> variable.	<code>String last = allList.get(allList.size()-1);</code>
Test whether the list contains the value "Marc".	<code>if (allList.contains("Marc")) {</code> <code>System.out.println("contained");</code> <code>}</code> <code>else{</code> <code>System.out.println("not contained");</code> <code>}</code>
Empty the list.	<code>allList.clear();</code>
Test if the list is empty.	<code>if (allList.isEmpty()) ...</code> or: <code>if (allList.size()==0) ...</code>



### 5.1.2. Sets

A **Set** is an unordered collection of objects in which an object can appear at most once. This notion of set corresponds to the mathematical notion.

In the Java library, the concept of a set is represented by the interface **Set**, and one of its implementations is the class **HashSet**.

The programme extract below illustrates their use by first creating the set of unaccented vowels in the Latin alphabet and then using it to determine the number of vowels in the word "deinstitutionalization":

```
Set<Character> vowels = new HashSet<>();
vowels.addAll(Arrays.asList('a', 'e', 'i', 'o', 'u', 'y'));

String word = "deinstitutionalization";
int vowelCount = 0;
for (int i = 0; i < word.length(); i++)
{
    if (vowels.contains(word.charAt(i))) {
        vowelCount++;
    }
}
System.out.println("The word " + word + " contains " + vowelCount + " vowels.");
```

The **Set** concept is represented in the Java library by the **Set** interface in the **java.util** package.

#### 5.1.2.1. Hashing

Hashing involves transforming any data into an integer, generally within a bounded range. This transformation is performed by a hash function which, when applied to the data, produces the corresponding integer, known as the hash value.

An example of a string hash function is the **hashCode()** method of the **String** class, defined as follows:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$$

with **s[i]** the  $i^{\text{th}}$  character of the string, **n** its length.

Examples:

"dog".hashCode() → 99644 ( $100 * 31^2 + 111 * 31^1 + 103$ )

"cat".hashCode() → 98262

"zoo".hashCode() → 120794

Hashing is a fundamental technique in computer science that has many applications, for example in cryptography. Here, we will only look at its use in collections.

A hash function is a mathematical function, in the sense that when applied to two equal values, it produces the same hash value. In other words, any hash function  $h$  satisfies the following property:

$$\forall x, y: x = y \Rightarrow h(x) = h(y)$$

A hash function  $h$  is said to be perfect if it maps a different hash value to each piece of data to which it can be applied, i.e., if:

$$\forall x, y: x \neq y \Rightarrow h(x) \neq h(y)$$

Since hash functions generally cannot be perfect, they must be designed to distribute the values encountered in practice as well as possible. Unfortunately, this condition is very difficult to specify and guarantee, as the definition of good hash functions is more of an art than a science.

When two distinct data items have the same hash value, i.e., when  $x \neq y$  but  $h(x) = h(y)$ , a hashing collision occurs.

### 5.1.2.2. HashSet

**HashSet** owes its name to the fact that it stores the elements of the set in a hash table. **HashSet** can perform the main operations on sets (add, membership test, etc.) in  $O(1)$ , which is remarkable!

The **HashSet** class implements sets of elements by distributing them in memory according to their hash value, so that they can be accessed more quickly. One consequence of this implementation technique is that the order in which the elements of such a set are traversed is arbitrary.

The **HashSet** class can therefore be used to store an unordered list of any number of objects without accepting duplicates.

The **HashSet** class is contained in the `java.util` package.

### 5.1.2.3. Browsing Sets

Since sets are unordered, it is not possible to specify a position and thus get the object at that position. Since the **Set** interface (indirectly) implements the **Iterable** interface, the elements of a set can be traversed using an iterator or the "for-each" loop, just like those of a list.

Note: unlike the elements of a list, the elements of a set are not ordered. The order in which the elements of a set are browsed therefore depends on the implementation used. **HashSet** traverses them in an arbitrary order, which can change from one execution of a program to the next, and even be different between two instances containing the same elements!

### 5.1.2.4. Hashing in Java

The Java designers chose to provide the `hashCode()` method in the **Object** class, which returns a hash value for the object to which it is applied, in the form of an integer of type `int`.

The default implementation of this method, which is inherited by any class that does not re-define it, returns a value that depends on the identity of the object. By default, two distinct objects therefore generally have a distinct hash value. However, this is not guaranteed!

It is absolutely essential that two objects considered equal by `equals()` have the same hash value according to `hashCode()`. In other words, the following property must be satisfied for any pair of objects `x` and `y`:

$$x.equals(y) \Rightarrow x.hashCode() = y.hashCode()$$

The `hashCode()` and `equals()` methods of a given class are said to be **compatible** if this condition is satisfied.

### 5.1.3. Association array

An associative array (**Map**) is a collection that associates values with keys.

Examples:

- The index of a book is an associative table which associates different words (the keys) with the list of page numbers on which that word appears (the values).
- In computing, an array can be seen as a special case of an associative table whose keys are the integers between 0 and the size of the array -1 and whose values are the elements of the array.

#### 5.1.3.1. Associative arrays in Java

In the Java library, the concept of an associative arrays is represented by the **Map** interface, and its implementations include the **HashMap** class.

The programme extract below illustrates their use by translating the word 'java' into Morse code. To do this, a table associating their Morse encoding (the values) with the characters of the alphabet (the keys) is first constructed. Once this has been done, the string 'java' is scanned, character by character, and the Morse translation of each character is obtained from the table and displayed on the screen:

```
Map<Character, String> morse = new HashMap<>();
morse.put('a', ".-");
morse.put('j', ".---");
morse.put('v', "...-");
// ... ditto for the other letters and punctuation.

String java = "java";
for (int i = 0; i < java.length(); i++)
    System.out.print(morse.get(java.charAt(i)) + " ");
```

The concept of an associative table is represented in the Java library by the Map interface in the `java.util` package. This interface takes two type parameters named **K** and **V**, which represent the type of keys and values respectively.

### 5.1.3.2. HashMap

The **HashMap** class is used to store a collection in an unordered way in the form of an association of key/value pairs.

The **HashMap** class is also contained in the **java.util** package.

The **HashMap** class requires its keys to be **hashable**, and uses this feature to provide the main  $O(1)$  operations. Note that only the keys must be **hashable**, there is no requirement on the values of an associative table.

### 5.1.3.3. Association table route

Unfortunately, the **Map** interface does not extend the **Iterable** interface, so it is not possible to directly browse the key/value pairs of an associative table using an iterator. To perform a traversal, you need to use the **entrySet()** method, which returns a set of elements contained in the associative table.

```
// a hash table containing students and their grades
HashMap<String, Integer> hm = new HashMap<>();

// adding a few students
hm.put("Jos" , 54);
hm.put("Nico", 34);
hm.put("Paul", 47);

// table layout
for (Map.Entry<String, Integer> mapElement : hm.entrySet()) {
    // recover the key
    String key = mapElement.getKey();

    // extract value
    int value = mapElement.getValue();

    // on-screen display
    System.out.println(key+" - "+value);
}
```

#### 5.1.4. Comparison ArrayList, HashSet and HashMap

ArrayList	HashSet	HashMap
<b>ArrayList</b> implements the <b>List</b> interface.	<b>HashSet</b> implements the <b>Set</b> interface.	<b>HashMap</b> implements the <b>Map</b> interface.
You can store objects in an <b>ArrayList</b> . If we have an <b>ArrayList</b> of <b>String</b> elements, we can represent it like this: <code>{ "Hello", "Good morning", "Goodbye", "Run" }</code>	You can store objects in a <b>HashSet</b> . If we have a <b>HashSet</b> of <b>String</b> elements, we can represent it like this: <code>{ "Hello", "Goodbye", "Run" }</code>	<b>HashMap</b> is used to store {key, value} pairs. In short, <b>HashMap</b> maintains the mapping between the key and the value. Here's how you can represent <b>HashMap</b> elements if they have an <b>integer</b> key and a <b>string</b> value:  <code>{1 -&gt; "Hello", 2 -&gt; "Hello", 3 -&gt; "Goodbye", 4 -&gt; "Execute" }</code>
<b>ArrayList</b> allows double elements to be stored.	<b>HashSet</b> does not allow duplicate elements, which means that the same element cannot be stored twice.	<b>HashMap</b> does not allow you to have the same key twice, but storing the same object twice with different keys is allowed.
The number of times <b>null</b> is stored in an <b>ArrayList</b> is unlimited.	<b>HashSet</b> allows the <b>null</b> value to be stored once.	<b>HashMap</b> allows you to have a single key with the value <b>null</b> . The number of times <b>null</b> is stored as a value is unlimited.
<b>ArrayList</b> maintains the order in which elements are inserted.	<b>HashSet</b> does not guarantee that the order in which elements are inserted will be maintained.	<b>HashMap</b> does not guarantee that the order in which elements are inserted will be maintained.

The time complexity of the basic operations (add, delete, etc.) of **HashSet** and **HashMap** is constant, i.e.,  $O(1)$ , which is not true for **ArrayList**.

More details and a comparative table, including references to other classes, can be found on the following website:

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>

## 5.1.5. Time complexity

	ArrayList	LinkedList	HashSet	HashMap
add an element at the end	$\Omega(1)$ $O(n)$	$\Omega(1)$ $O(n)$	$\Omega(1)$ $O(1)$	$\Omega(1)$ $O(1)$
access to an item	$\Omega(1)$ $O(1)$	$\Omega(n)$ $O(n)$	$\Omega(1)$ $O(1)$	$\Omega(1)$ $O(1)$
deleting an item	$\Omega(n)$ $O(n)$	$\Omega(n)$ $O(n)$	$\Omega(1)$ $O(1)$	$\Omega(1)$ $O(1)$
contains test	$\Omega(n)$ $O(n)$	$\Omega(n)$ $O(n)$	$\Omega(1)$ $O(1)$	$\Omega(1)$ $O(1)$

$\Omega$  = Best case (very rarely used)

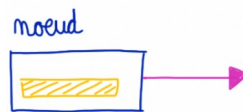
$O$  = Worst case (mostly used)

## 5.2. Simple linked lists

### 5.2.1. Operating principle

In computer science, a **linked list** is a data structure that represents an ordered collection of elements of the same type and of any size. In computer memory, the linked list is represented as a sequence of cells with one content and a pointer to another cell. In visual terms, the set of cells resembles a chain whose links are the cells.<sup>9</sup>

Each element in the chain is called a '**node**' and consists of a piece of data (orange rectangle in the image below) and a pointer to the next node in the chain (purple arrow):



In this way it is possible to create a linked list, node by node:



Keeping things generic, the source code for a node would look like this, although the **data** attribute could be any other type of data.

```
public class Node {
    private Object data;
    private Node next = null;

    public Node(Object data) {
        this.data = data;
    }
    // getters & setters have been omitted here
}
```

Given that the **Node** class only represents a node in a linked list, we necessarily still need a class to represent the list itself. It has an attribute (called **root** in the example below) pointing to the first node in the chain, and it also contains the various methods for operating the list (see 5.2.2 Operations).

```
public class MyList {
    protected Node root = null;

    // class methods do come here
}
```

<sup>9</sup>[https://en.wikipedia.org/wiki/Linked\\_list](https://en.wikipedia.org/wiki/Linked_list)

## 5.2.2. Operations

### 5.2.2.1. Browsing

To go through the list, start at the first node and proceed sequentially until you reach the end, i.e. until you reach a zero element.

The following method is used to print all the elements of a list in the console and is a good example of how to perform a sequential traversal of a linked list.

```
public void print()
{
    // save a pointer to the first node
    Node actual = root;

    System.out.print("[");
    // continue until the current node is null
    while(actual != null)
    {
        // display the data for the current node
        System.out.print(actual.getData());
        // jump to the next node
        actual=actual.getNext();
        if(actual != null) System.out.print(", ");
    }
    System.out.println("]");
}
```



### 5.2.2.2. Add at the end

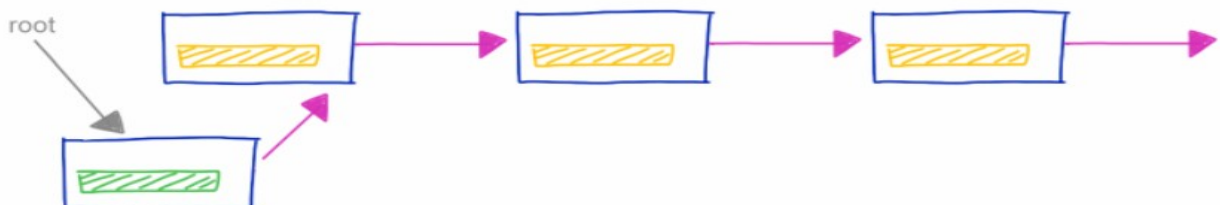
To add a node at the end of the chained list, find the reference of the last node, then add a new node (in the figure below, the node with the green rectangle):



```
public void addTail(Object o)
{
    // create a new node with the data passed as a parameter
    Node n = new Node(o);
    // special case if the list is still empty
    if (root == null)
    {
        root = n;
    }
    else
    {
        // route to find the last node
        Node last = root;
        // stop when the pointer from the current node to its next node is zero
        while (last.getNext() != null)
        {
            // advance to the next node
            last = last.getNext();
        }
        // adds new node
        last.setNext(n);
    }
}
```

### 5.2.2.3. Add at the beginning

Adding an element to the top of the list is simpler, as there are no special cases. In all cases, first modify the pointer of the new node so that it points to the current first element of the list, then modify the **root** pointer so that it points to the new node.



```
public void addHead(Object o)
{
    // create a new node with the data passed as a parameter
    Node n = new Node(o);
    // point the new node at the first node in the list
    n.setNext(root);
    // the new node becomes the base node
    root = n;
}
```

#### 5.2.2.4. Search

Research can be interpreted in different ways:

- search for a node at a given position,
- search for information stored at a given position,
- finding the position of a given piece of information,
- ...

To find a node at a given position, simply check that the position exists, then move to the desired position to return the node.

```
protected Node getNode(int index) throws ArrayIndexOutOfBoundsException
{
    // stop if the index is not correct >> throw an exception
    if(index<0 || index>=size()){
        throw new ArrayIndexOutOfBoundsException(index);
    }

    // start at the base node
    Node actual = root;
    // advance to the desired position
    for(int i=0; i<index; i++){
        actual=actual.getNext();
        // the current node is the one that was searched for
    }
    return actual;
}
```

Why is this method protected and not made public?

With the previous method at hand, finding the information stored at a given position becomes very straightforward:

```
public Object get(int index)
{
    return getNode(index).getData();
}
```

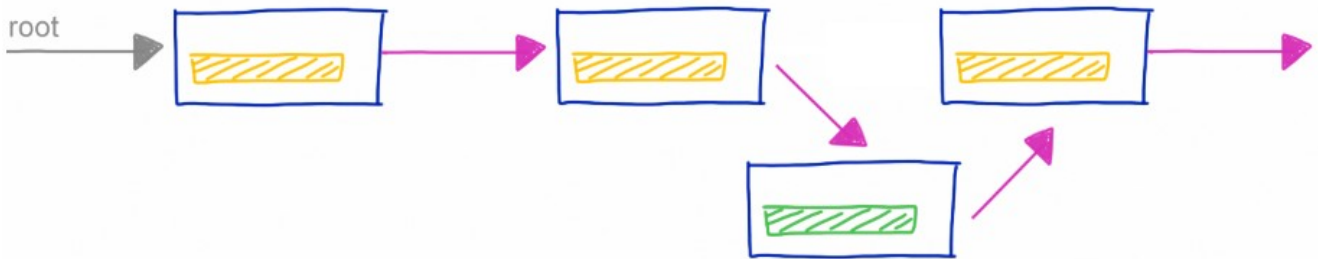
To find the position of a given item of information, you need to move forward while keeping a counter up to date. When you have found the item, you are looking for, you can stop, otherwise you must continue the search. If you reach the end of the list without having found what you were looking for, the value -1 is returned.

```
public int indexOf(Object o)
{
    // don't look for NULL values ;-)
    // if(o==null) return -1;
    // start at the base node
    Node actual = root;
    // to determine the position, we need a counter
    int i = 0;
    // keep going while we can
    while(actual!=null)
    {
        // test whether the data for the current node matches ...
        if(o.equals(actual.getData()))
        {
            // ... and return to the what position if this is the case.
            return i;
        }

        // if not, you have to carry on
        i++;
        // jump to the next node
        actual=actual.getNext();
    }
    // we found nothing :-(
    return -1;
}
```

### 5.2.2.5. Insertion at a precise location

To insert a node at a specific point, you need to look for a reference to the previous node. Only then is it possible to attach the tail of the list to the new node, then connect it to the "previous" node.



```
public void insert(Object o, int index)
{
    // stop if the index is not correct >> throw an exception
    // warning: adding to the size() position is allowed and is the same
    // as adding to the tail!
    if(index<0 || index>=size()+1)
        throw new ArrayIndexOutOfBoundsException(index);

    // create a new node
    Node n = new Node(o);
    // handling the special case
    if(index==0)
    {
        // point the new node at the first node in the list
        n.setNext(root);
        // the new node becomes the base node
        root=n;
    }
    else
    {
        // search for the previous node
        Node prev = getNode(index-1);
        // connect the chain tail to the new node
        n.setNext(prev.getNext());
        // connect the new node
        prev.setNext(n);
    }
}
```

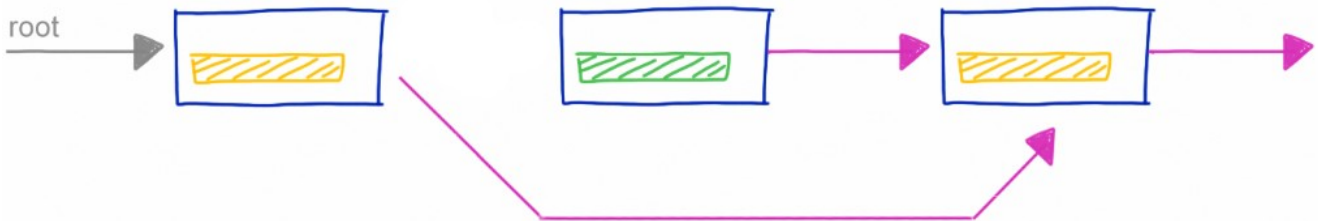
### 5.2.2.6. Replacement

To replace the information in a node at a given position, simply search for the node in question, then replace its contents. Note that the **getNode(...)** method is used to test whether the indicated position is within the limits.

```
public void set(Object o, int index)
{
    getNode(index).setData(o);
}
```

### 5.2.2.7. Delete

To delete a node at a given position, all you must do is search for the node before it, then change its pointer so that it points to the next node. In this way, the node to be deleted is 'by-passed' and no longer forms part of the chain.



The following code deletes the node at a given position:

```

public void remove(int index)
{
    // stop if the index is not correct >> throw an exception
    if(index<0 || index>=size()){
        throw new ArrayIndexOutOfBoundsException(index);
    }
    // handling the special case of the first node
    if(index==0)
    {
        // point the base node to the second node in the chain
        root=root.getNext();
    }
    else
    {
        // find the previous node
        Node prev = getNode(index-1);
        // skip the node to be deleted
        prev.setNext(prev.getNext().getNext());
    }
}
  
```

To remove a given object from the list, first find its position, then delete the node relating to it.

```

public void remove(Object o)
{
    remove(indexOf(o));
}
  
```

Question: What happens if we try to delete data from the list that isn't there?

### 5.2.2.8. Size

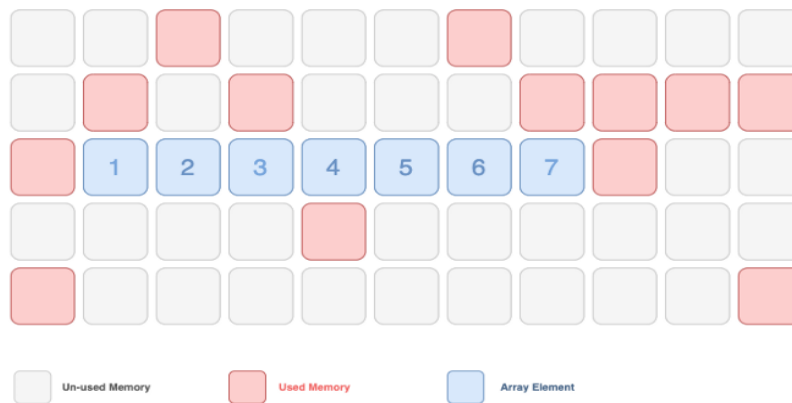
To determine the size of a linked list, count the elements one by one:

```
public int size()
{
    // handling the special case if the list is empty
    if (root == null)
        return 0;

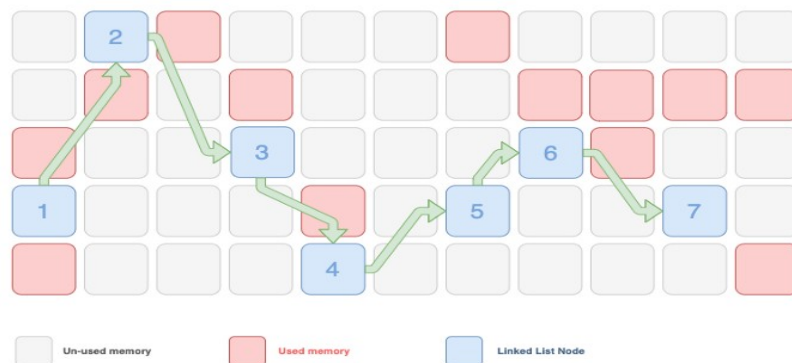
    // start at the base node
    Node actual = root;
    // initialise the competitor
    int count = 1;
    // progress as far as possible
    while (actual.getNext() != null)
    {
        // increment the counter
        count++;
        // jump to the next node
        actual = actual.getNext();
    }
    // return the result
    return count;
}
```

### 5.2.2.9. Comparison<sup>10</sup> with an array

In memory, an array can be represented as follows. The different cells are positioned next to each other. Their "address" can therefore be calculated using their position (index). This is why access can be made in  $O(1)$ .



In the linked list, the different nodes are not necessarily next to each other, but are distributed at random locations in the memory.

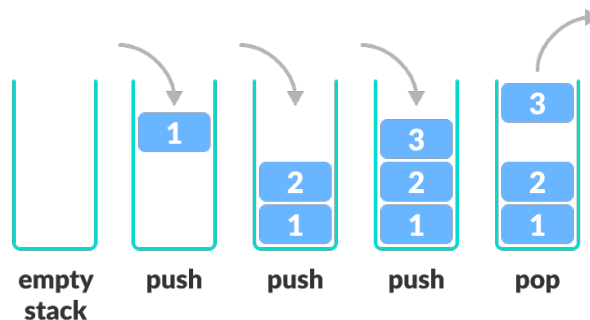


<sup>10</sup><https://levelup.gitconnected.com/array-vs-linked-list-data-structure-c5c0ff405f16>

### 5.2.3. Stack

In computing, a stack is a data structure based on the "last in, first out" (LIFO) principle, which means that, in general, the last element added to the stack will be the first to leave it.<sup>11</sup>

So a stack works internally like a linked list, but it also has **push(...)** and **pop()** methods for adding and removing elements.



```
public class MyStack extends MyList {

    public void push(Object o)
    {
        // add an item to the end of the list
        add(o);
    }

    public Object pop()
    {
        // deal with the special case of an empty list
        if(root==null) return null;
        // handle the case of a list containing only one element
        if(root.getNext()==null)
        {
            Node last = root;
            root = null;
            return last.getData();
        }

        // search for the penultimate node
        Node actual = root;
        while(actual.getNext().getNext()!=null)
            actual=actual.getNext();
        // obtain a reference to the last element
        Node last = actual.getNext();
        // delete the last element
        actual.setNext(null);
        // return the last element
        return last.getData();
    }
}
```

<sup>11</sup>[https://en.wikipedia.org/wiki/Stack\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) - <https://www.programiz.com/dsa/stack>



Alternatively, you could use the following **pop** method:

```
public Object pop()
{
    // deal with the special case of an empty list
    if(size()==0) return null;

    // obtain the index of the last node
    int lastIndex = size()-1;
    // get the last node
    Node last = getNode(lastIndex);
    // delete the last node
    remove(lastIndex);
    // return the data for the last node
    return last.getData();
}
```

What is the advantage of the first method? What is the advantage of the second method?

By saying that **MyStack** inherits from **MyList**, all of **MyList**'s methods are also exposed in **MyStack**, which is not necessarily what you want. To "hide" these methods, the simplest thing to do is to make the **MyStack** class **use** the **MyList** class instead of being a descendant of it.

```
public class MyStack {

    private MyList ml = new MyList();

    public void push(Object o)
    {
        ml.add(o);
    }

    public Object pop()
    {
        // deal with the special case of an empty list
        if(ml.size()==0) return null;

        // obtain the index of the last node
        int lastIndex = ml.size()-1;
        // get the last node
        Node last = ml.getNode(lastIndex);
        // delete the last node
        ml.remove(lastIndex);
        // return the data for the last node
        return last.getData();
    }
}
```

In this way, the **MyStack** class can decide "itself" which **MyList** methods it wants to make accessible and delegate (in NetBeans: right-click, "Insert Code...", "Delegate Method...").

### 5.2.4. Queue

In computing, a queue is a data structure based on the "first in, first out" (FIFO) principle, which means that the first elements added to the queue will be the first to be removed.<sup>12</sup>

A queue therefore functions internally in a similar way to a linked list, but it also has **push(...)** ("enqueue" on the graphical representation) and **pop()** ("dequeue") methods for adding and deleting an element. Given its resemblance to a stack, it could even be seen as a specialisation of the latter.



```
public class MyQueue extends MyStack {

    @Override
    public Object pop()
    {
        // deal with the special case of an empty list
        if(root==null) return null;

        // obtain a reference to the first element
        Node first = root;
        // delete the first element
        root = first.getNext();
        // return the first element
        return first.getData();
    }
}
```

As with the **MyStack** class, the **pop** method could also take the following form:

```
public Object pop()
{
    // deal with the special case of an empty list
    if(size()==0) return null;

    // obtain the first node
    Node first = getNode(0);
    // delete the first node
    remove(0);
    // return the data for the first node
    return first.getData();
}
```

The comment made earlier for the **MyStack** class about revealing the other methods is also true for the **MyQueue** class.

<sup>12</sup>[https://en.wikipedia.org/wiki/Queue\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) - <https://www.programiz.com/dsa/queue>

## 6. Recursion

### 6.1. Definition

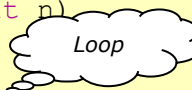
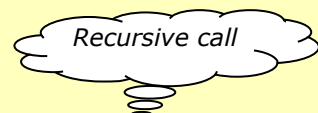
**recursion:** the fact that a method (or, more generally, a sub-program) calls itself directly or indirectly.

If a method calls itself directly, it is said to be **recursive**.

If the recursion concerns a group of methods, we speak of **indirect recursion** or we say that they are **mutually recursive**.



#### 6.1.1. Example

Calculating the factorial of a natural number	
Iterative solution	Recursive solution
Mathematical definition: $n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \quad (\forall n \in \mathbb{N}) \quad 0! = 1$	Mathematical definition: $\forall n \in \mathbb{N}:$ $\begin{cases} 0! = 1 & n = 0 \\ n! = n \cdot (n-1)! & n > 0 \end{cases}$
Programming: <pre>public int factorial(int n) {     int result = 1;     for (int i=1; i&lt;=n; i++){         result = result * i;     }     return result; }</pre> 	Programming: <pre>public int factorial(int n) {     if (n==0)         return 1;     else         return n * factorial(n-1); }</pre> 
Evaluation for $n=4$ $n=1:$ 1 $n=2:$ $1 \cdot 2 = 2$ $n=3:$ $1 \cdot 2 \cdot 3 = 6$ $n=4:$ $1 \cdot 2 \cdot 3 \cdot 4 = 24$	Evaluation for $n=4$ $4! = 4 \cdot 3!$ $= 4 \cdot (3 \cdot 2!)$ $= 4 \cdot (3 \cdot (2 \cdot 1!))$ $= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!)))$ $= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1)))$ $= 4 \cdot (3 \cdot (2 \cdot 1))$ $= 4 \cdot (3 \cdot 2)$ $= 4 \cdot 6$ $= 24$

### 6.2. The termination condition

When coding a recursive method, it is essential to include a termination condition, in other words a way of ending the calculation. Without this termination condition, the sub-programme

will continue to call itself indefinitely, consuming more and more memory and time, until an error occurs, and the calculation stops.

Let's look at the **factorial** method:

- When  $n$  is 0, the method returns 1. This is the only case that does not initiate a recursive call. The  $n=0$  condition is therefore the termination condition for the **factorial** method.
- For all positive values of  $n$ , a sequence of recursive calls is generated, starting with **factorial( $n-1$ )**. With each recursive call,  $n$  decreases, finally arriving at **factorial(0)**, i.e. the case that satisfies the termination condition. The results of all the recursive calls are then calculated 'backwards', starting with the last recursive call. Finally, the expression  $n * \text{factorial}(n-1)$  is evaluated, and the value returned as the result of the initial call.
- For all negative values of  $n$ , a sequence of recursive calls is generated, starting with **factorial( $n-1$ )**. With each recursive call,  $n$  decreases without ever reaching a call that satisfies the termination condition. This generates an 'infinite' sequence of recursive calls.

**Note:** As in mathematics the factorial function is defined only for natural numbers, the reaction of the method for a negative number can be neglected. In principle, it is the responsibility of the user of the method to ensure that the data corresponds to the definition domain of the function.

### A practical guide:

In conclusion, you should ensure that the following points are observed when constructing a recursive method:

- conditions and values for base cases/termination condition (i.e., cases that do not lead to recursive calls) must be found,
- the recursive call must be defined conditionally: each time it is called, the stop condition(s) must be checked,
- at each recursive call, one or more of the parameters passed to the sub-program must be close to the stop condition,
- the number of recursive calls to reach a result must be finite (if it were infinite, we'd have an endless loop),
- in a recursive sub-program, the complexity of the problem must be reduced with each new recursive call.

*"To understand recursion, you first have to understand recursion ..."*

*"This sentence is false" (Epimenides' paradox)*

### 6.3. Why use recursion?




The question is: why use recursion when there are non-recursive methods that are just as effective at performing the same task?

- Recursive methods tend to be more compact and often more elegant than iterative solutions.
- Some mathematical problems are defined recursively and can be solved very elegantly using recursion.
- Some problems are solved by applying recursion directly and would be insolvent (or difficult to solve) without recursion.
- Recursion is particularly well suited to processing recursively defined data structures.

#### Disadvantages of recursion

- Recursion requires a minimum of practice to master. While iteration comes naturally, recursion does not.
- Recursion is resource intensive. Each call to a recursive method generates as much system time as any call to a non-recursive method.
- The *stack* is used to store the state of the calling method, so that when the called method has finished, processing is returned to the caller. A stack overflow error occurs when an attempt is made to save the state of the calling method but there is no space left on the stack.

#### Please note:

- To follow the sequence of recursive calls in NetBeans, you can proceed as follows: Place a *breakpoint* (  ) next to the recursive call or inside the recursive method.
- Start the program using 'Debug Project' (  ).
- After the programme has stopped at the breakpoint, press the F7 key (or the  button) to step through the code.
- On the left, in the 'Debugging' window, you can see the sequence of recursive calls as they are stored on the **stack**. At the bottom, in the 'Variables' window, you can follow the contents of variables and parameters. You can even double-click on the various calls to see the parameter values during recursive calls.

## 7. Connecting to a database

### 7.1. Introduction to JDBC

Nowadays, there are a multitude of different database management systems (abr. DBMS) that respect the standard defined by the SQL structured query language, such as PostgreSQL, DB2, MSSQL or MySQL. The Java language must be able to interact with each of these DBMSs, regardless of the implementation used by the system in question.

However, instead of offering classes capable of directly exploiting all existing DBMSs, which would be far too laborious, Java only defines several interfaces in the **JDBC** API, which can be found in the `java.sql` package. These interfaces define all the attributes and methods that an implementation must have, without proposing one.

#### **Example:**

The `java.sql.Connection` interface defines the attributes and methods required to manage a connection to a DBMS, while `java.sql.Statement` manages the execution of SQL queries.

If a DBMS wants Java software to be able to interact with its system, it must provide an implementation of all the interfaces defined in the `java.sql` package. In the case of MySQL, this implementation is called **MySQL Connector/J**. Developers must therefore install this connector before they can start developing Java software capable of interacting with a MySQL server. Connector/J is available at the following address:

<https://dev.mysql.com/downloads/connector/j/>

Once the connector has been installed, you will find a JAR file in the installation directory which must be included in the NetBeans project. It is best to create a `lib` sub-directory in the project and place it there. This way, when you copy the project, it will always be included.

### 7.2. Connection to a MySQL server

A MySQL server uses port 3306 by default to always listen for new connections. As a DBMS is generally highly secure because of the sensitive data stored in the databases, a connection usually requires a username and password. What's more, a high-performance access rights management system defines exactly what operations are permitted for each user.

The steps involved in creating a connection are:

1. Loading the driver included in Connector/J.
2. Construction of a connection URL respecting the schema defined by JDBC.
3. Creating the connection using the `java.sql.DriverManager` class.

Here is the source code needed to create a connection to a DBMS:

```
import java.sql.*;

try {
    // Definition of connection parameters
    String server = "localhost"; // ip or dns of mysql server
    int port = 3306; // port where server is listening
    String username = "alice"; // username as defined in DBMS
    String password = "security"; // password as defined in DBMS
    String database = "school"; // database used for this project

    // Load the JDBC driver included in Connector/J
    Class.forName("com.mysql.cj.jdbc.Driver");

    // Create the connection URL with our parameters
    String url = "jdbc:mysql://" + server + ":" + port + "/" +
        database + "?user=" + username + "&password=" + password;

    // Create the connection to the server
    try (Connection connect = DriverManager.getConnection(url)) {

        // Connection established

    } catch (SQLException ex) {
        // Connection problem
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }

} catch (ClassNotFoundException ex) {
    // Driver problem
    Logger.getLogger(DatabaseTest.class.getName()).log(Level.SEVERE,
null, ex);
}
```

The source code above includes two catch blocks that deal with the various problems that may arise:

1. The first block deals with **SQLException** exceptions that occur when a connection has failed. There are many reasons for such failures: server not available, incorrect port, non-existent database, insufficient access rights, etc.
2. The second block deals with exceptions of type **ClassNotFoundException** which occur when the driver has not been loaded. This is generally due to incorrect configuration of the project in Netbeans, more specifically the absence of the Connector/J JAR file.

### 7.3. Executing an SQL query and analysing the results

Once a connection to the DBMS has been created, it can be used to send SQL queries. This is done using the `executeQuery()` method of the `java.sql.Statement` interface. The result of a query is an object of type `ResultSet` which contains a temporary table with the data returned by the query.

A `ResultSet` object contains zero or more lines which can be iterated over using the `next()` method. This advances an internal pointer as in the case of text files. To access the various columns of the temporary table, we use the accessors defined in the `ResultSet` interface by specifying the name of the column. The most common accessors are `getInt()`, `getDouble()` and `getString()`, but there are also more specific accessors for processing dates or times, for example.

The steps involved in sending a request are:

1. Creating a `Statement` object.
2. Use this object's `executeQuery()` method to specify the query to be sent.
3. Iterate over the rows of the returned `ResultSet` and exploit the data in such a row using accessors by specifying the column names.

Here is the source code needed to execute such an SQL query:

```
// Creation of statement object able to send queries
try (Statement statement = connect.createStatement()) {

    // create and execute SQL query
    ResultSet resultSet = statement.executeQuery("SELECT * FROM people");

    // iterate over the rows of the resultSet
    while (resultSet.next()) {
        // access the String contained in column "name"
        String name = resultSet.getString("name");
        System.out.println(name);
    }

} catch (SQLException ex) {
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
```

The source code above includes a `catch` block that also handles `SQLException` type exceptions. However, in this case, these exceptions occur when the query being executed has failed, for example in the case of a badly formed query, a non-existent table, etc.



**Attention**

Once a **ResultSet**, **Statement** or **Connection** has been closed, it can no longer be used. This is why:

- **ResultSet** and **Statement** are generally closed as soon as they are no longer used,
- the **Connection** is closed at the end of the programme.

**7.4. Full source code**

The following source code contains a complete and compact example summarising the chapters above:

```
import java.sql.*;

try {
    String server = "localhost";
    int port = 3306;
    String username = "alice";
    String password = "security";
    String database = "school";

    Class.forName("com.mysql.cj.jdbc.Driver");

    String url = "jdbc:mysql://" + server + ":" + port + "/" +
        database + "?user=" + username + "&password=" + password;

    try (Connection connect = DriverManager.getConnection(url)) {
        try (Statement statement = connect.createStatement()) {
            ResultSet resultSet =
                statement.executeQuery("SELECT * FROM people");

            while (resultSet.next()) {
                String name = resultSet.getString("name");
                System.out.println(name);
            }
        } catch (SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("VendorError: " + ex.getErrorCode());
        }
    } catch (SQLException ex) {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
} catch (ClassNotFoundException ex) {
    Logger.getLogger(DatabaseTest.class.getName()).log(Level.SEVERE,
null, ex);
}
```

## 7.5. Statement vs. PreparedStatement (for advanced users)

Using a simple Statement is not very efficient, flexible, or secure. This is why we generally work with **PreparedStatement**.

```
// PreparedStatement can use variables and are more efficient
PreparedStatement preparedStatement =
    connect.prepareStatement("SELECT * FROM persons WHERE name LIKE ?");

// Parameters start with 1
preparedStatement.setString(1, "Jos");

// Result set get the result of the SQL query
ResultSet resultSet = preparedStatement.executeQuery();

// ResultSet is initially before the first data set
while (resultSet.next())
{
    String name = resultSet.getString("name");
    // do something with all the data ...
}
```

A **PreparedStatement** uses question marks as placeholders in queries. These wildcards are then replaced using the **set...** methods with the actual data. **PreparedStatement**s are not only faster than regular **Statements**, but also prevent SQL code injection attacks.