# Learning goals

Before the next day, you should have achieved the following learning goals:

- Understand how to use interfaces in Java, and use them in your programs.
- Understand how stacks, queues, and maps work.
- Strengthen your understanding of pointers, and how they are used in dynamic data structures.

You should be able to finish most of non-star exercises in the lab. Remember that star exercises are more difficult. **Do not try star-exercises unless the other ones are clear to you**.

# 1 Supermarket queue

Use the interface `PersonQueue` that represents a queue of people waiting at the supermarket and then implement it. You can use the definition and the implementations of `StringStack` as a guide. You can reuse any version of class `Person` from past days. For example, depending on your implementation, you may want to use a version of `Person` with or without internal pointers.

```
public interface PersonQueue {
    /**
     * Adds another person to the queue.
     */
    void insert(Person person);

    /**
     * Removes a person from the queue.
     */
     Person retrieve();
}
```

Then create a class `Supermarket` that has two methods: `addPerson(Person)` and `servePerson()`. These methods must call the appropriate methods of `PersonQueue`.

# 2 Supermarket queue revisited (*)

Implement the interface `PersonQueue` in a different way. Then check that it works exactly the same without changing either the interface or your class `Supermarket`.

# 3 Foreign people, different queues (*)

Get a queue implementation from one of your colleagues. Use it and check that it works exactly the same without changing either the interface or your class `Supermarket`.

If it does not work, why is this?

# 4 Unfair queue (*)

## 4.1 Simple

Implement the interface queue in a way that the person at the end (i.e. the person that is retrieved next time the method `retrieve()` is called) is always the oldest person.

### 4.2 Clustered

Implement the interface queue in a way that the next person retrieved is always a person over 65, if there is any in the queue; if not, it must be a person over 18, if there is any in the queue. Inside each age category, the behaviour of the queue is typical FIFO: first in, first out.

These two queues are examples of *priority queues*, and are not strictly FIFO: old people will always come out of the queue before younger people, even if the youngsters came to the queue first. Priority queues are more difficult to implement, but they are also important in computing. For example, your hard disk uses a priority queue to decide where to move next: if the disk's head is at position 555 and the queue of requests is

$$4, 99, 234, 500, 101, 43, 881, 77$$

your disk may decide to move to position 500 to reduce movement, time, and energy consumption.

## 5 Maps

### 5.1 Hash function

Create a class `HashUtilities` that implements a simple hash function `shortHash(int)` that takes any integer and returns an integer between 0 and 1000. You can use the modulo operator and the static function `Math.abs(int)` for obtaining the absolute value of an integer.

Note that `shortHash(int)` is a pure function (it does not have any side effects), so it should be `static`. Then you can call this method using the name of the class like `HashUtilities.shortHash(int)`.

Every object in Java has the method `hashCode()`, that returns an `int`. Test your hash function by passing the hash codes of different objects and verifying that it always returns a number between 0 and 1000, as in the following example:

```
println "Give me a string and I will calculate its hash code";
String str = System.console().readLine();
int hash = str.hashCode();
int smallHash = HashUtilities.shortHash(hash);
System.out.println("0 < " + smallHash + " < 1000");
```

### 5.2 Simple map

Create a class that implements the following interface of a simple map from integers to strings.

```
/**
 * Map from integer to Strings
 */
public interface SimpleMap {
    /**
     * Puts a new String in the map.
     *
     * If the key is already in the map, nothing is done.
     */
    void put(int key, String name);

     /**
      * Returns the name associated with that key,
      * or null if there is none.
      */
     String get(int key);

     /**
      * Removes a name from the map. Future calls to get(key)
      * will return null for this key unless another
      * name is added with the same key.
```

```
     */
    void remove(int key);

    /**
     * Returns true if there are no workers in the map,
     * false otherwise
     */
    boolean isEmpty();
}
```

## 5.3 Hash table (*)

Create a class that implements the following interface of a simple map from integers to strings: it is a one-to-many mapping. A similar map is used in some countries to classify the citizens into groups for tax purposes (so that each department has a limited number of citizens to examine and process).

```
/**
 * Map from integer to Strings: one to many
 */
public interface SimpleMap {
    /**
     * Puts a new String in the map.
     */
    String put(int key, String name);

    /**
     * Returns all the names associated with that key,
     * or null if there is none.
     */
    String[] get(int key);

    /**
     * Removes a name from the map.
     */
    void remove(int key, String name);

    /**
     * Returns true if there are no workers in the map,
     * false otherwise
     */
    boolean isEmpty();
}
```

Hint: You can implement it with arrays or with linked lists. You do **not** know in advance how many strings you will receive for every key.