Collections

Lists

Lists are collections whose elements are indexed. In other words, elements in these collections can be accessed by an index 0, 1, 2, ..., N, so that they can be sorted, and iterated. In Java, every type of list implements a global interface called `List`, here you can see the API of this interface.

1. List management in Java

If you take a look at `List` API, there are some methods that we can use in any list type, such as:

- `add(element)` or `add(index, element)`: to add an element, either at the end of the list, or at a given index.
- `clear()` to clear the list (remove all of its elements).
- `contains(element)`: to check if a given element already exists in the list (as long as the class overrides `equals` method to know how to check if two elements are equal or not).
- `get(index)`: to get the element at the specified index
- `indexOf(element)`: gets the index of the first occurrence of the given element (as long as the class implements `equals` method). If the element does not exist in the list, then -1 is returned.
- `remove(index)`: to remove the element at the specified index
- `remove(element)`: to remove the first occurrence of the specified element in the list (as long as the class overrides `equals` method)
- `size()` to get the total number of elements stored in the list

The most popular list subtype that we can use in Java is `ArrayList` class (API). As it implements `List` interface, we can use all of the methods from that interface. Internally, an *ArrayList* element is just a list that treats the data as an internal array (although we don't need to care about the size of this array).

We can create an ArrayList by either defining an `ArrayList` variable or a `List` variable (using polymorphism):

```
List myList = new ArrayList();
ArrayList myOtherList = new ArrayList();
```

Then, we can add, get, remove... elements from that list. These elements can be of any type, so we must take care when getting elements from the list, and make sure that they are of the appropriate type.

```
myList.add("Hello");
myList.add(new Person("Nacho", 40));
...
if (myList.get(1) instanceof Person)
{
    Person p = (Person)(myList.get(1));
    System.out.println(p.getName());
}
```

1.1. Using generics

If we use lists as we have seen in previous example, we may have some troubles, since we can add any kind of object in the list, so we need to cast and check the object types before working with them. For instance, in previous example we have typecasted the elements extracted from the list in order to use them. Otherwise, Java treats them as `Object` instances:

```
// Only Object methods can be called
Object o = myList.get(1);

// Every Person method can be called
Person p = (Person)(myList.get(1));
```

In order to avoid these checkings, we can use **generics**. Generics are a way to customize a given object or collection to work with a concrete data type. This data type is expressed between `<` and `>` symbols, after the collection type.

For instance, if we want to work with a list of strings using generics, we initialize the list this way (either using a `List` or an `ArrayList`):

```
List<String> stringList = new ArrayList<>();
ArrayList<String> anotherStringList = new ArrayList<>();
```

From this point on, every element that we add to the collection needs to be a string, and whenever we get an element from the collection, we can be sure that it will be a string, so no typecast is needed.

```
stringList.add("Hello");
stringList.add("Goodbye");
System.out.println(stringList.get(1).toUpperCase()); // GOODBYE
```

Note that we can use **polymorphism** with generics, so that, if we create a list of `Animal` objects, for instance, there can be any type of animal in that list (dogs, ducks, and so on).

```
List<Animal> animals = new ArrayList<>();
animals.add(new Dog(...));
animals.add(new Duck(...));
...
```

1.2. List subtypes

There are other list subtypes available in the Java API, although we are not going to work with them in this unit. Here you can have a quick overview:

- `Vector` class is something similar to `ArrayList`, but it is thread-safe (this is, it is suitable for working with multiple threads), so it is not as efficient as *ArrayList* is.
- `LinkedList` class is another list subtype in which every element is not only linked with the following one, but with previous one, so we can explore the list from any of its edges with the same efficiency

There are some important differences regarding performance between `ArrayList` (or `Vector`) and `LinkedList`:

- `ArrayList`: its main advantage is the complexity of accessing an index to get an element, which is *O(1)*, this is, a constant value, but it's not the best implementation when we want to do many add and remove operations, that have a complexity of *O(n)*, this is, it depends on the list size.
- `LinkedList`: there is no internal array, so nothing has to be resized, elements are inserted and removed only by modifying references in the previous and next object. This is the main advantage of this type of list. The main disadvantage is accessing a random position because it has to travel through all items to get to the specified index (there's no internal index to access a position directly)

1.3. Lists of basic data types

If we want to create a list of some basic data type (such as *integers* or *doubles*) we can't specify this simple type in the generic:

```
List<int> numbers = new ArrayList<>();
```

Alternatively, we can make use of **wrappers**, special classes that are part of the Java API and can replace basic types in some situations. These classes are:

- `Integer` (for `int` type)
- `Float` (for `float` type)
- `Double` (for `double` type)
- `Character` (for `character` type)

- ...

> **NOTE**: `String` is not a basic type, because it's made of a sequence of *chars*, so there's no wrapper for this class. We just use `String` class.

Then, we can create a list (or any collection) of any of these types, and then add elements on it as basic types:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(3);
numbers.add(8);

int aNumber = numbers.get(0);
```

1.4. Sorting lists

In the same way that we use `Comparator` or `Comparable` interfaces to determine how to sort complex objects in an array through `Arrays.sort` method, there is a `Collections.sort` method (in class `java.util.Collections`) that lets us sort a list using a comparator.

All that we need is to define the comparing method (either in the own class to be sorted, or in another class through a `Comparator`), and then call this method to sort the collection. For instance, if we have a list or `Person` objects, we can sort it by age using this comparator:

```
List<Person> people = new ArrayList<>();
... // Fill list
Collections.sort(people, new Comparator<Person>()
{
    @Override
    public int compare(Person p1, Person p2)
    {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
});
```

If the own class (`Person` in this example) has the comparing method in its code, then we can call `Collections.sort` with just one parameter (the collection to be sorted):

```
Collections.sort(people);
```

> **Exercise 1:**
>
> Go back to Exercise 1 of this document. Replace the video game array in `main` method with a generic `ArrayList`. Then, add some video games to the list (either `VideoGame` or `PCVideoGame` objects),

> explore and show the list with a `for` and ask the user to:
>
> - Search video games by title: the user will type a title and then the program will show all the video games whose title contains the typed text (ignoring case).
> - Remove a video game from the list: the user will type the index of the video game to be removed, and if the index is valid, the video game in that index will be removed.

> **Exercise 2:**
>
> Sort the video game list in previous exercise by price in ascending order using `Collections.sort` and the appropriate comparing method. Do this step before printing the list in the screen.

2. Stacks and queues

There are some specific types of lists which are more restrictive regarding how to use add/remove operations. In this section we are going to talk about stacks and queues.

2.1. Stacks

A stack is a collection (or list subtype) in which elements can only be added or removed by one of the edges called *top*. We call these structures LIFO structures, which stands for *Last In, First Out*. Only the element at the top of the stack can be modified (added or removed).

In order to use stacks in Java we have `Stack` class. As you can see in the official API, it's a subtype of `Vector` class, so it's a list. It just adds some specific methods to deal with the top:

- `push(element)` : adds a new element onto the top of the stack
- `pop()` : removes the element at the top of the stack and returns it (or throws an exception if stack is empty)
- `peek()` : returns the element at the top of the stack without removing it (it also throws an exception if stack is empty).
- `empty()` : checks if stack is currently empty

Here you can see an example of how to use it:

```
Stack(Integer) numberStack = new Stack<>();
numberStack.push(3);
numberStack.push(9);
...
while (!stack.empty())
{
    int number = stack.pop();
    System.out.println(number);
}
```

> **Exercise 3:**

> Create a project called **ReversedFigure** that asks the user to draw a bi-dimensional figure (line by line), and then prints the same figure inverted vertically. For instance, if user draws this:

```
   *
  ***
 *****
*******
```

> then the program must output this:

```
*******
 *****
  ***
   *
```

> **NOTE**: as you may notice, stacks are really useful when it comes to inverting orders.

2.2. Queues

Queues are another special type of list in which elements are added by one of the edges and removed from the opposite edge. These structures are also called FIFO structures, which stands for *First In, First Out*. The main basic operations in queues are:

- Adding elements (enqueue)
- Extracting elements (dequeue)

Java provides `Queue` interface in order to deal with queues. You can see in the API that there are some additional, useful methods:

- `add(element):` adds a new element in the tail of the queue
- `remove()` or `poll()`: retrieve and remove the element at the head of the queue. There's a difference between these two methods, depending on whether we want to return *null* if queue is empty or not.
- `peek()`: checks the element at the head of the queue (without removing it)

There are some classes in Java API implementing this interface. One of them is `LinkedList` class, which we have mentioned before. Here you have an example of how to use it as a queue:

```
LinkedList<Integer> numbers = new LinkedList<>();

numbers.add(1);
numbers.add(3);
numbers.add(4);

while(numbers.size() > 0)
{
    System.out.println("We are going to extract " + numbers.peek());
    System.out.println(numbers.remove());
}
```

Exercise 4:

Create a project called **ConcertAttendance**. Define a class called `Person` to represent every person that wants to attend the concert, and store his/her name and age. Then, in the main program, add some `Person` objects to a queue (add them until user enters empty values). Then, filter from this queue only the adult people, and print the final result in the screen, in arrival order.