by Nacho Iborra

Development Environments
Block 3
Unit 1: Using collections

Whenever we have needed to work with a group of elements of objects of a given type, we have used arrays so far. However, arrays are *static*, this is, their size can't be modified, so they are not suitable for applications where the amount of objects that we want to store may vary as the application runs. Then, we should use a collection.

3.1.1. Introduction to collections

A collection is a **dinamic** data structure that contains objects which we want to store and operate with in a particular way. So, the size of a collection can change during the execution of a program, as we add or remove elements to/from the collection.

In Java, there are several types of collections, although we are going to focus on three subtypes:

- **Lists:** collections in which every element has assigned a numeric index (as in arrays), and we can explore the whole collection by going from one index to the following.
- **Maps:** also called *hash tables* or *dictionaries*, they store key-value pairs, in which, given a key, we can get the corresponding value associated to it. The key can be of any type (numeric, string or even complex objects), so we don't have any numeric index to explore the collection
- **Sets:** a collection of elements with no duplicates. Elements have no numeric index associated, and we usually explore the set through an iterator.

Depending on the application we are developing, we may choose one of these types (or some of them). In the following subsections we are going to know the most popular classes that we can use in Java to represent and work with each type of collection. Almost all of these classes belong to `java.util` package, so we will need to import it in our projects.

3.1.2. Lists

Lists are collections whose elements are indexed, 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.

If you take a look at that 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

3.1.2.1. ArrayList

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.

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());
}
```

3.1.2.2. 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.

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).

3.1.2.3. Other 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
- `Stack` class is a special list subtype to define a stack, this is, a list in which elements are always added and removed from the same edge. It is a LIFO structure (which stands for *Last In, First Out*).
- *Queues* are another speciel list subtype to define queues, this is, a list in which elements are added only by one of the edges, and removed from the opposite one. It is a FIFO structure (which stands for *First In, First Out*). Actually, `Queue` is not a list subclass in Java, but another interface, but the meaning and behavior is really similar to a list.

3.1.2.4. Sorting lists automatically

In the same way that we used `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 any type of collections 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 like the array that we used in previous unit, we can sort it using the same comparator class:

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

Proposed exercises:

3.1.2.1. Go back to exercise 2.6.4.1 of unit 6. 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.

3.1.2.2. Sort previous list by price in ascending order using `Collections.sort` and the appropriate comparing method. Do this step before printing the list in the screen.

3.1.3. Maps

Maps are a type of dynamic collection in which every element or value is referenced by a key. They are also called hash tables or *dictionaries*, because they work like a dictionary: if we know the word we are looking for, we can "jump" to it (without exploring all previous words in the dictionary) and check its meaning.

In Java, every type of map implements a global interface called `Map`, here you can see the API of this interface.

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

- `clear()` to clear the map (remove all of its elements).
- `containsKey(key)` to check if the map contains a given key.

- `get(key)` : to get the value associated to the given key.
- `put(key, value)` : to add the specified key-value pair to the map
- `remove(key)` : to remove the key-value pair identified by the given key.
- `size()` to get the total number of elements stored in the map

3.1.3.1. HashMap

The most popular map subtype that we can use in Java is `HashMap` class (API). As it implements `Map` interface, we can use all of the methods from that interface.

We can create a HashMap by either defining a `HashMap` variable or a `Map` variable (using polymorphism). In both cases, we should use generics (although it is not compulsory), to establish the data type of both the key and the value. In the following example, we define a map whose keys are strings, and whose values are `Person` objects. Then we add some elements to the map (see how we specify both the key and value when putting the element in the map), and look for a given element by its key:

```
Map<String, Person> myMap = new HashMap<>();
myMap.put("11223344A", new Person("Nacho", 40));
myMap.put("22334455B", new Person("Arturo", 35));
...
// Print the name of the Person with key = 11223344A
System.out.println(myMap.get("11223344A").getName());
```

3.1.3.2. Exploring maps

If we want to explore a map, we can't use a traditional `for` and increase an index to go to each position, since there is no numeric index in maps. Instead of this, we need to get the set of keys (with `keySet` method), and explore it with a "foreach":

```
for(String key : myMap.keySet())
{
    System.out.println(myMap.get(key).getName());
}
```

Proposed exercises:

3.1.3.1. Create a project called *Library*, with a main class called `Library` and another class called `Book`, in which we are going to store some information about each book: the *id* (a string), the title and the author's name. In the main application, define a map of books, whose keys will be their corresponding *ids*. Manually add some books to the map, and then explore it and show the books in the screen (override the `toString` method of `Book` class to properly show book info).

3.1.4. Sets

A set is a collection with no duplicate elements. In Java, every type of set implements a global interface called `Set` (here you can see the API of this interface).

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

- `add(element)`: to add an element to the set, as long as it is not already present. Notice that this method returns a **boolean**, indicating if the new element could be added or not.
- `clear()` to clear the set (remove all of its elements).
- `contains(element)`: to check if a given element already exists in the set (as long as the class overrides `equals` method to know how to check if two elements are equal or not).
- `iterator()`: to get an iterator and explore the elements of the set with it.
- `remove(element)`: to remove the element from the set (as long as the class overrides `equals` method). This method also returns a boolean.
- `size()` to get the total number of elements stored in the set

As you can see, there is no index to specify the position of the elements in the set, so we must explore them with an iterator. Let's suppose that we have a set of strings... We can explore it this way:

```
Set<String> mySet = ...;
Iterator<String> it = mySet.iterator();
while(it.hasNext())
{
    String s = it.next();
    ...
}
```

3.1.4.1. HashSet

One of the most popular classes to work with sets is `HashSet` class (API). It works like *HashMap* class seen before, but we only specify the keys of the map (not the values). As it implements `Set` interface, we can use all of the methods from that interface.

We can create a HashSet by either defining a `HashSet` variable or a `Set` variable (using polymorphism). In both cases, we should use generics (although it is not compulsory), to establish the data type of the keys. In the following example, we define a set of strings and add some strings on it. Then we remove a given string from the set:

```
Set<String> mySet = new HashSet<>();
mySet.add("Hello");
mySet.add("Hello"); // Will not work, "Hello" already exits
mySet.add("Goobye");
...
mySet.remove("Goodbye");
```

Proposed exercises:

3.1.4.1. Create a project called *FairyTaleSet* with a main program that stores a set of fairy tales. For each fairy tale, we are going to store its title and the number of pages, so define a *FairyTale* class with these two attributes, a constructor and the corresponding getters and setters.
Also, override `equals` method to determine when two fairy tales will be considered the same: they will be the same whenever they have the same title. Then, in the main application, define a set of `FairyTale` objects, and add some of them to the list. Try to add some fairy tales with the same title, to see how many of them are finally included in the set. Then, explore the set with an iterator and print the data in the screen (you can also override `toString` method for this).

NOTE: in order to determine if two fairy tales are the same by title, besides overriding `equals` method, you need to override `hashcode` method to generate a *hash code* with its title (not the number of pages), so that Java can compare two hash codes of two different fairy tales and determine that the titles are the same. `hashcode` method can be automatically generated by IntelliJ, as getters, setters and other common methods.