Collections

Maps, sets and trees

Apart from lists, there are some other important collection types in Java that we can use in our programs. Let's have a look at them in this document.

1. 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.

1.1. Map management in Java

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

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());
```

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());
}
```

1.2. Hash function

How does Java place each element of a map in a position so that it can be automatically retrieved by its key? Well, it applies a **hash function**, this is, a function that converts the key into a numeric value, which represents an index in the map. Then, the element is automatically placed at that index (user doesn't need to care about it, obviously).

Every object in Java has (and can override) an inherited function called `hashCode` that returns an integer associated to this object. Java uses this hashing function to determine the number associated to each key in the map, and this function can also be used, along with `equals` method, to automatically check if two objects are equal or not, as we will see in later examples.

1.3. Map subtypes and performance

`HashMap` and `Hashtable` are `Map` implementations using a hash function for distributing keys. The main difference between them is that a `Hashtable` is synchronized, so it's thread safe. Another difference is that with `HashMap` you can use `null` as a key.

The main advantage of this type of structures is that, when you want to search for a value, it's very fast to find (compared to a list for example), since it only has to calculate a hash function to find out where the desired object is placed. When you want to insert a lot of entries and search a lot too, instead of iterating, a `Map` implementation is usually better suited than a `List`.

> **Exercise 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).

2. 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();
    ...
}
```

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");
```

Exercise 2:

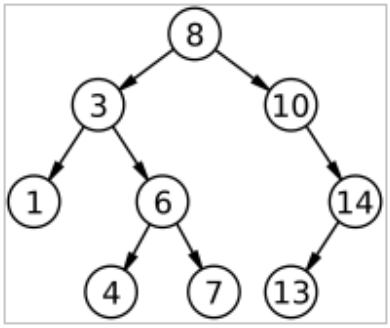
Create a project called **FairyTaleSet** with a main program that stores a set of fairy tales. For each fairy tale, we are going to store its title and the number of pages, so define a `FairyTale` class with these two attributes, a constructor and the corresponding getters and setters.
Also, override `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.

3. Trees

A tree is a hierarchical structure with a root value from which we can find a set of linked nodes. Each of these nodes is a subtree indeed, with all of the nodes linked from it. There are some restrictions: no node is duplicated, there are no cycles and no node points to the root.

This is an example of a tree. Root node is represented by number 8, and it has two subtrees: one represented by number 3 and another one represented by number 10. Subtree number 3 has two more subtrees inside: numbers 1 and 6, and so on...



We can define a tree exploring its nodes recursively (it can be useful, for instance, for exploring our directory structure), and we can also define a tree as an ordered structure, so that all the nodes at the left of a given node are lower than this node, and all the nodes at the right of a node are higher. The tree shown in the image above is an ordered tree (all the nodes of the left subtree are lower than 8, and all the nodes of the right subtree are higher than 8, and we can say the same with any subtree). The main advantage of this ordered trees, assuming that they are balanced (i.e., all the branches have more or less the same depth), is that the complexity of searching a given node is simpler than in a list. If a list has N elements, the complexity of searching an element is O(N); it means that we could explore up to N elements before finding the desired one. However, if we have a balanced tree with N nodes, where each node has up to M branches, the complexity of searching an element in this tree is $O(\log_M N)$, which is lower than O(N).

There are several types of tree structures, such as binary trees (each node has 2 branches), red-black trees (balanced binary tree), and so on. We will not focus on implementing these types of trees, but in using the ones provided by Java. Let's see how they work with an example.

Some of the implementations for maps and sets use trees to store the values, such as `TreeMap` or `TreeSet`. In general, trees are a structure not very usual in our programs, although they have some advantages.

Let's see an example: we have an interface called `Shape` to represent different geometrical shapes (circles, squares and so on). They all have a `getArea` method, provided by the interface, that calculates the area of each shape. We could override the toString `method` in each subclass (`Circle`, `Square` ...) to make them print the shape type and the area. For instance, in `Square` class we would have this method:

```
@Override
public String toString()
{
    return String.format("Square -> Area: %.2f", getArea());
}
```

From this class structure, we can create a `TreeSet` that stores a set of shapes, ordering them automatically by their area. Our main program would look like this:

```
Set<Shape> set = new TreeSet<>(new Comparator<Shape>()
{
    @Override
    public int compare(Shape s1, Shape s2)
    {
        return Double.compare(s1.getArea(), s2.getArea());
    }
});

set.add(new Triangle(6.25, 8));
set.add(new Triangle(7.2, 6.78));
set.add(new Circle(4.3));
set.add(new Circle(1.88));
set.add(new Rectangle(9.25, 7.6));
set.add(new Rectangle(9, 2.55));

for(Shape s: set)
    System.out.println(s.toString());
```

Note that we create a new `Comparator` object with an anonymous class, and implement the method compare, that compares two shapes by their area. Then, we add some shapes to the tree (unordered), and if we run the program, we would get this:

```
Circle -> Area: 11,10
Rectangle -> Area: 22,95
Triangle -> Area: 24,41
Triangle -> Area: 25,00
Circle -> Area: 58,09
Rectangle -> Area: 70,30
```

You can see that shapes are listed ordered by their area automatically. This is one of the strengths of trees.