Using Maps to Store Key Value Pairs

Introducing the Map Hierarchy

The second main structure offered by the Collections Framework is an implementation of a very classic data structure: the hashmap structure. This concept is not new and is fundamental in structuring data, whether in-memory or not. How does it work and how has it been implemented in the Collections Framework?

A hashmap is a structure able to store key-value pairs. The value is any object your application needs to handle, and a key is something that can represent this object.

Suppose you need to create an application that has to handle invoices, represented by instances of an `Invoice` class. Then your values are these `Invoice` instances, and your keys could be the invoice numbers. Each invoice has a number, and that number is unique among all your invoices.

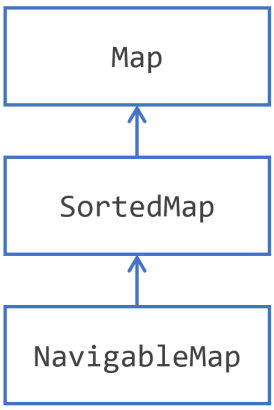
Generally speaking, each value is bound to a key, just as an invoice is bound to its invoice number. If you have a given key, you can retrieve the value. Usually a key is a simple object: think of a string of several characters or a number. The value, on the other hand, can be as complex as you need. This is what hashmaps have been made for: you can manipulate keys, move them from one

part of your application to another, transmit them over a network, and when you need the full object, then you can retrieve it with its key.

Before you see all the details of the `Map` interface, here are the notions you need to have in mind.

- A hashmap can store key-value pairs
- A key acts as a symbol for a given value
- A key is a simple object, a value can be as complex as needed
- A key is unique in a hashmap, a value does not have to be unique
- Every value stored in a hashmap has to be bound to a key, a key-value pair in a map forms an *entry* of that map
- A key can be used to retrieve its bound value.

The Collections Framework gives you a `Map` interface that implements this concept, along with two extensions, `SortedMap` and `NavigableMap`, as shown on the following figure.



The Map Interface Hierarchy

This hierarchy is very simple and looks like the `Set` hierarchy, with `SortedSet` and `NavigableSet`. Indeed, a `SortedMap` shares the same kind of semantics as the `SortedSet`: a `SortedMap` is a map that keeps its key-value pairs sorted by their keys. The same goes for `NavigableMap`: the methods added by this

interface are the same kind of methods than the ones added by `NavigableSet` to `SortedSet`.

The JDK gives you several implementations of the `Map` interface, the most widely used is the `HashMap` class.

Here are the two other implementations.

- `LinkedHashMap` is a `HashMap` with an internal structure to keep the key-value pairs ordered. Iterating on the keys or the key-value pairs will follow the order in which you have added your key-value pairs.
- `IdentityHashMap` is a specialized `Map` that you should only be used in very precise cases. This implementation is not meant to be generally used in application. Instead of using `equals()` and `hashCode()` to compare the key objects, this implementation only compares the references to these keys, with an equality operator (`==`). Use it with caution, only if you are sure this is what you need.

You may have heard of multimaps. Multimap is a concept where a single key can be associated to more than one value. This concept is not directly supported in the Collections Framework. This feature may be useful though, and you will see later in this tutorial how you can create maps with values that are in fact lists of values. This pattern allows you to create multimap-like structures.

Using the Convenience Factory Methods for Collections to Create Maps

As you already saw, Java SE 9 added methods to the `List` and `Set` interfaces to create immutable lists and sets.

There are such methods on the `Map` interface that create immutable maps and immutable entries.

You can create a Map easily with the following pattern.

```
Map<Integer, String> map =
    Map.of(
        1, "one",
        2, "two",
        3, "three"
    );
```

There is one caveat though: you can only use this pattern it if you have no more than 10 key-value pairs.

If you have more, then you need to use another pattern:

```
Map.Entry<Integer, String> e1 = Map.entry(1, "one");
Map.Entry<Integer, String> e2 = Map.entry(2, "two");
Map.Entry<Integer, String> e3 = Map.entry(3, "three");

Map<Integer, String> map = Map.ofEntries(e1, e2, e3);
```

You can also write this pattern in this way, and use static imports to further improve its readability.

```
Map<Integer, String> map3 =
    Map.ofEntries(
        Map.entry(1, "one"),
        Map.entry(2, "two"),
        Map.entry(3, "three")
    );
```

There are restrictions on these maps and entries created by these factory methods, as for the sets:

- The map and the entries you get are immutable objects
- Null entries, null keys, and null values are not allowed
- Trying to create a map with duplicate keys in this way does not make sense, so as a warning you will get an `IllegalArgumentException` at map

creation.

Storing Key/Value Pairs in a Map

The relationship between a key and its bound value follows these two simple rules.

- A key can be bound to only one value
- A value can be bound to several keys.

This leads to several consequences for the content of the map.

- The set of all the keys cannot have any duplicates, so it has the structure of a `Set`
- The set of all the key/value pairs cannot have duplicates either, so it also has the structure of a `Set`
- The set of all the values may have duplicates, so it has the structure of a plain `Collection`.

Then, you can define the following operations on a map:

- Putting a key/value pair in the map. This may fail if the key is already defined in the map
- Getting a value from a key
- Removing a key from a map, along with its value.

You can also define the classic, set-like operations:

- Checking if the map is empty or not
- Getting the number of key-value pairs contained in the map
- Putting all the content of another map in this map

- Clearing the content of a map.

All these operations and concepts are implemented in the `Map` interface, along with some others that you are going to see in the following.

Exploring the Map interface

The `Map` interface is the base type that models the notion of map in the JDK.

You should be extremely careful when choosing the type of the keys for your maps. In a nutshell, choosing a mutable key is not prohibited but is dangerous and discouraged. Once a key has been added to a map, mutating it may lead to changing its hash code value, and its identity. This may make your key-value pair unrecoverable or may get you a different value when querying your map. You will see this later on an example.

The `Map` defines a member interface: `Map.Entry` to model a key-value pair. This interface defines three methods to access the key and the values:

- `getKey()`: to read the key;
- `getValue()` and `setValue(value)`: to read and update the value bound to that key.

The `Map.Entry` objects you can get from a given map are views on the content of the map. Modifying the value of an entry object is thus reflected in the map and the other way round. This is the reason why you cannot change the key in this object: it could corrupt your map.

Last update: *September 14, 2021*