**Java Map Implementations**

Since Map is an interface, you need to instantiate a concrete implementation of the Map interface in order to use it. The Java Collections API contains the following Map implementations:

* java.util.HashMap
* java.util.Hashtable
* java.util.EnumMap
* java.util.IdentityHashMap
* java.util.LinkedHashMap
* java.util.Properties
* java.util.TreeMap
* java.util.WeakHashMap

The most commonly used Map implementations are HashMap and TreeMap.

Each of these Map implementations behaves a little differently with respect to the order of the elements when iterating the Map, and the time (big O notation) it takes to insert and access elements in the maps.

HashMap maps a key and a value. It does not guarantee any order of the elements stored internally in the map.

TreeMap also maps a key and a value. Furthermore, it guarantees the order in which keys or values are iterated - which is the sort order of the keys or values. The HashMap implementation is typically the fastest of the two Map implementations, so whenever you don't need to sort the elements in the Map you can just use a HashMap. Otherwise use a TreeMap.

**Creating a MAP**

To create a Java Map you must create an instance of one the classes that implement the Java Map interface. Here are a few examples of how to create a Map instance:

Map mapA = new HashMap();

Map mapB = new HashMap();

**Generic JAVA Map**

From Java 5, Java Generics makes it possible to limit the types of object you can use for both keys and values in a Map. Here is an example:

 Map<String, MyObject> map = **new** HashMap<String, MyObject>();

This Map can now only accept String objects for keys, and MyObject instances for values. You can then access and iterate keys and values without casting them. Here is how it looks:

**for**(MyObject anObject : map.values()){

//do something with anObject...

}

**for**(String key : map.keySet()){

MyObject value = map.get(key);

//do something to value

}

If you know the type of the objects being stored in the Map it is considered good practice to always specify a generic type when declaring and creating a Java Map. The generic type helps you avoid inserting the wrong objects, and makes it easier for people reading your code to understand what kind of objects the Map contains. Throughout the rest of this tutorial, I will be using generic types on all Map examples whenever it makes sense.

Adding elements into JAVA Map

To add elements to a Map you call its put() method. Here are a few examples:

Map<String, String> map = **new** HashMap<>();

map.put("key1", "element 1");

map.put("key2", "element 2");

map.put("key3", "element 3");

The three put() calls maps a string value to a string key. You can then obtain the value using that key.

**Only Objects Can Be Inserted**

Only Java objects can be used as keys and values in a Java Map. In case you pass primitive values (e.g.. int, double, etc.) to a Map as key or value, the primitive values will be auto-boxed before being passed as parameters. Here is an example of auto-boxing primitive parameters passed to the put() method:

map.put("key", 123);

The value passed to the put() method in the above example is a primitive int. Java auto-boxes it inside an Integer instance though, because the put() method requires an Object instance as both key and value. Auto-boxing would also happen if you passed a primitive as the key to the put() method.

If you call put() more than once with the same key, the latest value passed to put() for that key will overwrite what is already stored in the Map for that key. In other words, the latest value replaces the existing value for the given key.

**Null Keys:** Quite surprisingly you can use the value null as a key in a Java Map. Here is an example of using a null key in a Java Map:

Map map = new HashMap();

map.put(null, "value for null key");

To obtain the value stored by the null key you call the get() method with null as parameter value. Here is an example of getting the value for the null key of a Java Map:

Map<String, String> map = new HashMap<>();

String value = map.get(null);

**Inserting All Elements From Another Map**

The Java Map interface has a method called putAll() which can copy all key + value pairs (entries) from another Map instance into itself. In set theory, this is also referred to as the *union* of the two Map instances.

Here is an example of copying all entries from one Java Map into another via putAll():

Map<String, String> mapA = new HashMap<>();

mapA.put("key1", "value1");

mapA.put("key2", "value2");

Map<String, String> mapB = new HashMap<>();

mapB.putAll(mapA);

After running this code the Map referenced by variable mapB will contain both of the key + value entries inserted into mapA at the beginning of the code example.

The copying of entries only goes one way. Calling mapB.putAll(mapA) will only copy entries from mapA into mapB, not from mapB into mapA. To copy entries the other way, you would have to execute the code mapA.putAll(mapB).

**Get Elements from a Java Map**

To get a specific element stored in a Java Map you call the get() method, passing along the key for that element as a parameter. Here is an example of getting a value stored in a Java Map:

Map map = new HashMap();

map.put("key1", "value 1");

String element1 = (String) map.get("key1");

Notice that the get() method returns a Java Object, so we have to cast it to a String (because we know the value is a String). Later in this Java Map tutorial, you will see how to use Java Generics to type the Map so it knows what specific key and value types it contains. This makes type casting unnecessary and makes it harder to insert the wrong values into the Map by accident.

If we had specified a generic type for the key and value of the Map, then it would not have been necessary to cast the object returned by get() method. Here is how that would look:

Map<String, String> map = new HashMap<>();

map.put("key1", "value 1");

String element1 = map.get("key1");

**Get or Default Value**

The Java Map interface has a ***getOrDefault***() method which can return a default value supplied by you - in case no value is stored in the Map by the given key. Here is an example of getting a value from a Java Map with a backup default value:

Map<String, String> map = new HashMap<>();

map.put("A", "1");

map.put("B", "2");

map.put("C", "3");

String value = map.getOrDefault("E", "default value");

This example creates a Map and stores three values in it using the keys A, B and C. Then the example calls the Map getOrDefault() method, passing the String E as key, along with a default value - the String default value. Since the Map does not contain any object stored by the key E the given default value will be returned - which is the String default value passed as the last parameter to the *getOrDefault*() method.

**Checking if Map Contains Key**

You can check if a Java Map contains a specific key using the containsKey() method. Here is how that looks:

boolean hasKey = map.containsKey("123");

After running this code, the hasKey variable will have the value true if a key + value pair was inserted earlier with the String key 123, and false if no such key + value pair was inserted.

**Checking if Map Contains Value**

The Java Map interface also has a method that enables you to check if the Map contains a certain value. The method is called containsValue() . Here is how calling the containsValue() looks:

boolean hasValue = map.containsValue("value 1");

After executing this code the hasValue variable will contain the value true if a key + value pair was inserted earlier with the String value "value 1", and false if not.

**Iterating the Keys of a Java Map**

There are several ways to iterate the keys stored in a Java Map. The most used methods for iterating the keys are:

* Via the key Iterator
* Via the for-each loop
* Via a Stream

All methods will be covered in the following sections.

**Using a Key Iterator**

You can iterate all the keys of a Java Map via its keySet() method. Here is how iterating the keys of a Java Map looks:

Iterator iterator = map.keySet().iterator();

while(iterator.hasNext(){

Object key = iterator.next();

Object value = map.get(key);

}

As you can see, the key Iterator returns every key stored in a Java Map, one by one (one for each call to the next()). Once you have the key, you can obtain the element stored for that key using the Map get() method.

In the example above, the Iterator next() method returns an Object - and so does the get() method. With generic types specified for the Map, these methods would have returned the type of the key and value objects respectively. Here is how that would look:

Map<String, String> map = new HashMap<>();

Iterator<String> iterator = map.keySet().iterator();

while(iterator.hasNext(){

String key = iterator.next();

String value = map.get(key);

}

Notice how a generic type is now also specified for the Iterator obtained from the map.keySet().iterator().

**Using a Key For-Each Loop**

From Java 5 you can also use the for-each loop to iterate the keys stored in a Java Map. Here is how that looks:

for(Object key : map.keySet()) {

Object value = map.get(key);

}

The effect of the above code is pretty similar to the code shown in the previous section.

If you have specified a generic type for the Java Map, then you can use that type inside the for-each loop. Here is how that looks:

Map<String, String> map = new HashMap<>();

for(String key : map.keySet()) {

String value = map.get(key);

}

**Using a Key Stream**

From Java 8 you can use a Java Stream to iterate the keys of a Java Map. The Stream interface is part of the [**Java Stream API**](https://jenkov.com/java-functional-programming/streams.html) which was added in Java 8. You first obtain the key Set from the Map and from that you can get a Stream. Here is an example of iterating the keys of a Java Map via a Stream:

Map<String, String> map = new HashMap<>();

map.put("one", "first");

map.put("two", "second");

map.put("three", "third");

Stream<String> stream = map.keySet().stream();

stream.forEach((value) -> {

System.out.println(value);

});

**Iterating the Values of a Java Map**

It is also possible to just iterate the values stored in a Java Map. You obtain a Collection of the values stored in a Map via the values() method. You can iterate the values in the Collection in following ways:

* Using an Iterator
* Using the for-each Loop
* Using a value Stream

All of these options are covered in the following sections.

**Using a Value Iterator**

The first way to iterate all values stored in a Java Map is to obtain a value Iterator instance from the value Set, and iterate that. Here is how iterating the values stored in a Java Map using a value Iterator:

Map<String, String> map = new HashMap<>();

Iterator<String> iterator = map.values().iterator();

while(iterator.hasNext()) {

String nextValue = iterator.next();

}

Since a Set is unordered, you do not have any guarantees about the order in which the values in the value set are iterated. However, if you are using a TreeSet you can control this order still.

**Using a Value For-Each Loop**

The second method of iterating the values stores in a Java Map is via the Java for-each loop. Here is how iterating the values of a Java Map using the for-each loop looks in code:

Map<String, String> map = new HashMap<>();

for(String value : map.values()){

System.out.println(value);

}

This example will print out all the values stored in the mapA Map variable.

**Using a Value Stream**

The third way to iterate the values stored in a Java Map is by using a value Stream, by using the [**Java Stream API**](https://jenkov.com/java-functional-programming/streams.html). You first obtain the value Set from the Map, and from the value Set you can obtain the Stream. Here is an example of iterating the values of a Java Map via a value Stream:

Map<String, String> map = new HashMap<>();

map.put("one", "first");

map.put("two", "second");

map.put("three", "third");

Stream<String> stream = map.values().stream();

stream.forEach((value) -> {

System.out.println(value);

});

**Iterating the Entries of a Java Map**

It is also possible to iterate all entries of a Java Map. By entries I mean key + value pairs. An entry contains both the key and value for that entry. Earlier we have only iterated either the keys, or the values, but by iterating the entries we iterate both at the same time.

Like with keys and values, there are two ways to iterate the entries of a Map:

* Using an Entry Iterator
* Using the for-each loop

Both of these options will be explained in the following sections.

**Using an Entry Iterator**

The first way to iterate the entries of a Java Map is via an entry Iterator obtained from the entry Set. Here is an example of iterating the entries of Java Map:

Set<Map.Entry<String, String>> entries = map.entrySet();

Iterator<Map.Entry<String, String>> iterator =

entries.iterator();

while(iterator.hasNext()) {

Map.Entry<String, String> entry = iterator.next();

String key = entry.getKey();

String value = entry.getValue();

}

Notice how the key and value can be obtained from each Map.Entry instance.

Keep in mind that the above code can be made a bit nicer using a Map typed with [**Java Generics**](https://jenkov.com/java-generics/index.html) as shown later in this tutorial.

**Using an Entry For-Each Loop**

The second way to iterate the entries of a Java Map is to use a for-each loop. Here is an example of iterating the entries of a Java Map using a for-each loop:

for(Map.Entry<String, String> entry : map.entrySet()){

String key = entry.getKey();

String value = entry.getValue();

}

Notice, that this example too can be made a bit prettier using a generic Map. Generic Map instances are explained a bit later in this Java Map tutorial.

**Removing Entries From a Java Map**

You remove Entries by calling the remove(Object key) method. You thus remove the (key, value) pair matching the key. Here is an example of removing the entry for a given key in a Java Map :

map.remove("key1");

After executing this instruction, the Map referenced by mapA will no longer contain an entry (key + value pair) for the key key1.

**Removing All Entries**

You can remove all entries in a Java Map using the clear() method. Here is how that looks:

map.clear();

**Replacing an Entry in a Java Map**

It is possible to replace an element in a Java Map using the replace() method. The replace() method will only insert the new value if there is already an existing value mapped to the key. If no existing value is mapped to the given key, no value is inserted. This is different from how put() works, which always insert the value no matter what.

Here is an example of replacing one value with another using the Java Map replace() method:

Map map = new HashMap<>();

map.replace("key", "val2"); //no "key" entry, no replace

map.put("key", "val1"); //now contains "key" entry

map.replace("key", "val2"); //now "key" entry replaced

After running this code the Map instance will contain the String value newer value for the String key key .

**Reading Number of Entries in Map**

You can read the number of entries in a Java Map using the size() method. The number of entries in a Java Map is also referred to as the Map *size* - hence the method name size() . Here is an example of reading the number of entries in a Map using the size() method:

int entryCount = map.size();

**Checking if a Java Map is Empty**

The Java Map interface has a special method for checking if a Map is empty. This method is called *isEmpty*() and it returns either true or false. The isEmpty() method will return false if the Map instance contains 1 or more entries. If the Map contains 0 entries, isEmpty() will return true.