The Map interface present in java.util package represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.

**Characteristics of a Map Interface**

* A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the **HashMap** and **LinkedHashMap**, but some do not like the **TreeMap**.
* The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable order, while HashMap does not.
* There are two interfaces for implementing Map in java. They are, Map and SortedMap, and three classes: **HashMap, TreeMap and LinkedHashMap**.

**Why and When to use Maps?**

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some examples are:

* A map of error codes and their descriptions.
* A map of zip codes and cities.
* A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
* A map of classes and students. Each class (key) is associated with a list of students (value).

**1. HashMap:** HashMap is a part of Java’s collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. This class uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches. Let’s see how to create a map object using this class.

**2. LinkedHashMap:** LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Let’s see how to create a map object using this class.

**3. TreeMap:** The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. Let’s see how to create a map object using this class.

1. **import** java.util.\*;
2. **class** MapExample2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Elements can traverse in any order
9. **for**(Map.Entry m:map.entrySet()){
10. System.out.println(m.getKey()+" "+m.getValue());
11. }
12. }
13. }

|  |  |
| --- | --- |
| Method | Description |
| [clear()](https://www.geeksforgeeks.org/map-clear-method-in-java-with-example/) | This method is used to clear and remove all of the elements or mappings from a specified Map collection. |
| [containsKey(Object)](https://www.geeksforgeeks.org/map-containskey-method-in-java-with-examples/) | This method is used to check whether a particular key is being mapped into the Map or not. It takes the key element as a parameter and returns True if that element is mapped in the map. |
| [containsValue(Object)](https://www.geeksforgeeks.org/map-containsvalue-method-in-java-with-examples/) | This method is used to check whether a particular value is being mapped by a single or more than one key in the Map. It takes the value as a parameter and returns True if that value is mapped by any of the key in the map. |
| [entrySet()](https://www.geeksforgeeks.org/map-entryset-method-in-java-with-examples/) | This method is used to create a set out of the same elements contained in the map. It basically returns a set view of the map or we can create a new set and store the map elements into them. |
| [equals(Object)](https://www.geeksforgeeks.org/map-equals-method-in-java-with-examples/) | This method is used to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not. |
| [get(Object)](https://www.geeksforgeeks.org/map-get-method-in-java-with-examples/) | This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key. |
| [hashCode()](https://www.geeksforgeeks.org/map-hashcode-method-in-java-with-examples/) | This method is used to generate a hashCode for the given map containing key and values. |
| [isEmpty()](https://www.geeksforgeeks.org/map-isempty-method-in-java-with-examples/) | This method is used to check if a map is having any entry for key and value pairs. If no mapping exists, then this returns true. |
| [keySet()](https://www.geeksforgeeks.org/map-keyset-method-in-java-with-examples/) | This method is used to return a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. |
| [put(Object, Object)](https://www.geeksforgeeks.org/map-put-method-in-java-with-examples/) | This method is used to associate the specified value with the specified key in this map. |
| [putAll(Map)](https://www.geeksforgeeks.org/map-putall-method-in-java-with-examples/) | This method is used to copy all of the mappings from the specified map to this map. |
| [remove(Object)](https://www.geeksforgeeks.org/map-remove-method-in-java-with-examples/) | This method is used to remove the mapping for a key from this map if it is present in the map. |
| **size()** | This method is used to return the number of key/value pairs available in the map. |
| **values()** | This method is used to create a collection out of the values of the map. It basically returns a Collection view of the values in the HashMap. |

## ****3. Creating a new Map****

### ****Creating a HashMap:****

Always use interface type (Map), generics and diamond operator to declare a new map. The following code creates a HashMap:

|  |
| --- |
| Map<Integer, String> mapHttpErrors = new HashMap<>();   mapHttpErrors.put(200, "OK");  mapHttpErrors.put(303, "See Other");  mapHttpErrors.put(404, "Not Found");  mapHttpErrors.put(500, "Internal Server Error");   System.out.println(mapHttpErrors); |

This maps HTTP status codes to their descriptions. Output:

|  |  |
| --- | --- |
| 1 | {404=Not Found, 500=Internal Server Error, 200=OK, 303=See Other} |

As you can see in the output, a HashMap does not impose any order on its key-value elements.

You can create a new Map that copies elements from an existing map. For example:

|  |  |
| --- | --- |
| 1 | Map<Integer, String> mapErrors = new HashMap<>(mapHttpErrors); |

The map mapErrors is created with initial elements copied from the map mapHttpErrors.

**Creating a LinkedHashMap:**

The following code creates a LinkedHashMap that maps phone numbers with contact names:

|  |
| --- |
| Map<String, String> mapContacts = new LinkedHashMap<>();   mapContacts.put("0169238175", "Tom");  mapContacts.put("0904891321", "Peter");  mapContacts.put("0945678912", "Mary");  mapContacts.put("0981127421", "John");    System.out.println(mapContacts); |

Output:

|  |  |
| --- | --- |
| 1 | {0169238175=Tom, 0904891321=Peter, 0945678912=Mary, 0981127421=John} |

As you can see, the LinkedHashMap maintains its elements by their insertion order.

### ****Creating a TreeMap:****

The following code creates a TreeMap that maps file extensions to programming languages:

|  |
| --- |
| Map<String, String> mapLang = new TreeMap<>();   mapLang.put(".c", "C");  mapLang.put(".java", "Java");  mapLang.put(".pl", "Perl");  mapLang.put(".cs", "C#");  mapLang.put(".php", "PHP");  mapLang.put(".cpp", "C++");  mapLang.put(".xml", "XML");    System.out.println(mapLang); |

Output:

|  |  |
| --- | --- |
| 1 | {.c=C, .cpp=C++, .cs=C#, .java=Java, .php=PHP, .pl=Perl, .xml=XML} |

As you can see, the TreeMap sorts its keys by their [natural ordering](https://www.codejava.net/java-core/collections/understanding-object-ordering-in-java-with-comparable-and-comparator), which is the alphabetical order in this case.

## ****4. Performing Basic Operations on a Map****

The basic operations of a Map are association (**put**), lookup (**get**), checking (**containsKey**and **containsValue**), modification (**remove**and **replace**) and cardinality (**size** and **isEmpty**).

**Associating a value with a key:**

The **put(K, V)** method associates the specified value V with the specified key K. If the map already contains a mapping for the key, the old value is replaced by the specified value:

|  |
| --- |
| Map<Integer, String> mapHttpErrors = new HashMap<>();  mapHttpErrors.put(400, "Bad Request");  mapHttpErrors.put(304, "Not Modified");  mapHttpErrors.put(200, "OK");  mapHttpErrors.put(301, "Moved Permanently");  mapHttpErrors.put(500, "Internal Server Error"); |

### ****Getting a value associated with a specified key:****

The **get(Object key)** method returns the value associated with the specified key, or returns null if the map contains no mapping for the key. Given the map in the previous example:

|  |  |
| --- | --- |
| 1  2 | String status301 = mapHttpErrors.get(301);  System.out.println("301: " + status301); |

Output:

|  |  |
| --- | --- |
| 1 | 301: Moved Permanently |

### ****Checking if the map contains a specified key:****

The method **containsKey(Object key)** returns true if the map contains a mapping for the specified key. For example:

|  |
| --- |
| if (mapHttpErrors.containsKey("200")) {      System.out.println("Http status 200");  } |

Output:

|  |  |
| --- | --- |
| 1 | Found: Http status 200 |

### ****Checking if the map contains a specified value:****

The method **containsValue(Object value)** returns true if the map contains one or more keys associated with the specified value. For example:

|  |  |
| --- | --- |
| 1  2  3 | if (mapHttpErrors.containsValue("OK")) {      System.out.println("Found status OK");  } |

Output:

|  |  |
| --- | --- |
| 1 | Found status OK |

### ****Removing a mapping from the map:****

The **remove(Object key)** method removes the mapping for a key from the map if it is present (we care about only the key, and the value does not matter). This method returns the value to which the map previously associated the key, or null if the map doesn’t contain mapping for the key. Here’s an example:

|  |
| --- |
| String removedValue = mapHttpErrors.remove(500);   if (removedValue != null) {      System.out.println("Removed value: " + removedValue);  } |

Output:

|  |  |
| --- | --- |
| 1 | Removed value: Internal Server Error |

Similarly, the **remove(Object key, Object value)** method removes the mapping of a specified key and specified value, and returns true if the value was removed. This method is useful in case we really care about the key and value to be removed.

I recommend you to read this [well-know Java collection book](https://amzn.to/2JzHwVv) to learn in-depth about Java collections framework.

### ****Replacing a value associated with a specified key:****

The **replace(K key, V value)**method replaces the entry for the specified key only if it is currently mapping to some value. This method returns the previous value associated with the specified key. Here’s an example:

|  |
| --- |
| System.out.println("Map before: " + mapHttpErrors);   mapHttpErrors.replace(304, "No Changes");   System.out.println("Map after: " + mapHttpErrors); |

Output:

|  |  |
| --- | --- |
| 1  2 | Map before: {400=Bad Request, 304=Not Modified, 200=OK, 301=Moved Permanently}  Map after: {400=Bad Request, 304=No Changes, 200=OK, 301=Moved Permanently} |

Similarly, the **replace(K key, V oldValue, V newValue)** method replaces the entry for the specified key only if it is currently mapping to the specified value. This method returns true if the value was replaced. Useful in case we want to replace exactly a key-value mapping.

### ****Getting the size of the map:****

The **size()**method returns the number of key-value mappings in this map. For example:

|  |  |
| --- | --- |
| 1 | int size = mapHttpErrors.size(); |

Output:

|  |  |
| --- | --- |
| 1 | Number of HTTP status code: 5 |

### ****Checking if the map is empty:****

The **isEmpty()** method returns true if the map contains no key-value mappings. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (mapHttpErrors.isEmpty()) {      System.out.println("No Error");  } else {      System.out.println("Have HTTP Errors");  } |

Output:

|  |  |
| --- | --- |
| 1 | Have HTTP Errors |

## ****5. Iterating Over a Map (using Collection views)****

As a Map is not a true collection, there is no direct method for iterating over a map. Instead, we can iterate over a map using its collection views. Any Map’s implementation has to provide the following three Collection view methods:

* **keySet()**: returns a Set view of the keys contained in the map. Hence we can iterate over the keys of the map as shown in the following example:

|  |
| --- |
| Map<String, String> mapCountryCodes = new HashMap<>();   mapCountryCodes.put("1", "USA");  mapCountryCodes.put("44", "United Kingdom");  mapCountryCodes.put("33", "France");  mapCountryCodes.put("81", "Japan");    Set<String> setCodes = mapCountryCodes.keySet();  Iterator<String> iterator = setCodes.iterator();    while (iterator.hasNext()) {      String code = iterator.next();      String country = mapCountryCodes.get(code);        System.out.println(code + " => " + country);  } |

* Output:

|  |  |
| --- | --- |
| 1  2  3  4 | 44 => United Kingdom  33 => France  1 => USA  81 => Japan |

* **values()**: returns a collection of values contained in the map. Thus we can iterate over values of the map like this:

|  |
| --- |
| Collection<String> countries = mapCountryCodes.values();   for (String country : countries) {      System.out.println(country);  } |

* Output:

|  |  |
| --- | --- |
| 1  2  3  4 | United Kingdom  France  USA  Japan |

* **entrySet()**: returns a Set view of the mappings contained in this map. Therefore we can iterate over mappings in the map like this:

|  |
| --- |
| Set<Map.Entry<String, String>> entries = mapCountryCodes.entrySet();   for (Map.Entry<String, String> entry : entries) {      String code = entry.getKey();      String country = entry.getValue();       System.out.println(code + " => " + country);  } |

Output:

|  |  |
| --- | --- |
| 1  2  3  4 | 44 => United Kingdom  33 => France  1 => USA  81 => Japan |

Since Java 8 with Lambda expressions and the **forEach()** statement, iterating over a Map is as easy as:

|  |  |
| --- | --- |
| 1  2 | mapCountryCodes.forEach(      (code, country) -> System.out.println(code + " => " + country)); |

Output:

|  |  |
| --- | --- |
| 1  2  3  4 | 44 => United Kingdom  33 => France  1 => USA  81 => Japan |

For more information about the different methods of collection iteration, read this article: [The 4 Methods for Iterating Collections in Java](https://www.codejava.net/java-core/collections/the-4-methods-for-iterating-collections-in-java).

## ****6. Performing Bulk Operations with Maps****

There are two bulk operations with maps: **clear()** and **putAll()**.

The **clear()** method removes all mappings from the map. The map will be empty after this method returns. For example:

|  |  |
| --- | --- |
| 1  2 | mapHttpErrors.clear();  System.out.println("Is map empty? " + mapHttpErrors.isEmpty()); |

Output:

|  |  |
| --- | --- |
| 1 | Is map empty? true |

The **putAll(Map<K, V> m)** method copies all of the mappings from the specified map to this map. Here’s an example:

|  |
| --- |
| Map<Integer, String> countryCodesEU = new HashMap<>();   countryCodesEU.put(44, "United Kingdom");  countryCodesEU.put(33, "France");  countryCodesEU.put(49, "Germany");   Map<Integer, String> countryCodesWorld = new HashMap<>();   countryCodesWorld.put(1, "United States");  countryCodesWorld.put(86, "China");  countryCodesWorld.put(82, "South Korea");   System.out.println("Before: " + countryCodesWorld);   countryCodesWorld.putAll(countryCodesEU);   System.out.println("After: " + countryCodesWorld); |

Output:

|  |  |
| --- | --- |
| 1  2 | Before: {1=United States, 82=South Korea, 86=China}  After: {1=United States, 33=France, 49=Germany, 82=South Korea, 86=China,  44=United Kingdom} |

## ****7. Concurrent Maps****

Unlike the legacy Hashtable which is synchronized, the HashMap, TreeMap and LinkedHashMap are not synchronized. If thread-safe is priority, consider using ConcurrentHashMap in place of HashMap. Or we can use the **Collections.synchronizedMap()** utility method that returns a synchronized (thread-safe) map backed by the specified map. For example:

|  |  |
| --- | --- |
| 1 | Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>()); |

And remember we have to manually synchronize the map when iterating over any of its collection views:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Set<Integer> keySet = map.keySet();    synchronized (map) {      Iterator<Integer> iterator = keySet.iterator();        while (iterator.hasNext()) {          Integer key = iterator.next();          String value = map.get(key);      }  } |

If you use a kind of SortedMap, e.g. TreeMap, consider using the more specific method **Collections.synchronizedSortedMap()**.

**NOTE:** If you use your own type for the key and value (e.g. Student or Employee), the key class and value class must implement the **equals()** and **hashCode()** methods properly so that the map can look up them correctly.