Map is an ADT (abstract data type). The values of this ADT are objects that map *keys* to *values*. The best way to understand this is to look at an example. To the right is part of an index of a book. Given a word, like “abort”, it tells you the page number, 114, where that word is defined or used. The index *maps* words to page numbers.

abort, 114

abs, 314

abstraction, 149

ambiguity, 308

Here is how we would declare a variable index that can contain such a map:

Map <String, Integer> index;

The first type parameter gives the type of the keys; the second gives the type of the values.

After building index to contain the four key-value pairs in the box above, we say that “abort maps to 114” and that “index contains the pair (“abort”, 114).”

Note that the keys form a *set*; the same key cannot map to two different values.[[1]](#footnote-1) Secondly, unless specifically prohibited by an implementation, **null** can be used as both a key and a value.

**Implementation HashMap of Map**

Java provides over twenty different implementations of interface Map! We mention only one of them here, HashMap, since you will use it most often. HashMap should generally be used like this:

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

Here, a HashMap object is created and is immediately cast to Map and stored in index. From then on, only operations declared in Map can be used, even though HashMap may provide others. Always cast to the interface for maximum flexibility in choosing an implementation later on.

HashMap has expected constant time for many operations, which is why we use it, although the worst case is linear in the size of the map. The implementation is based on a technique called *hashing*. You don’t have to know how it works to use it, although you will learn about hashing in your data structure course if you haven’t already.

**Major operations on a Map**

A type consists of a bunch of values and a bunch of operations on them. We introduce the major operations defined in interface Map:

1. **put(k, v)** . Use this call to add the key-value pair (k, v) to the map. If a pair (k, v1) is already in the map, the new pair replaces. *You don’t have to delete the old pair* (k, v1) *before adding the new pair* (k, v)*.*   
   The call put(k, v) actually returns a value. If a pair (k, v1) is already in the map, v1 is returned, otherwise **null** is returned. We rarely use this feature and just use the call as a statement.
2. **size()** . This tells you how many key-value pairs are in the map.
3. **isEmpty()** . This tells you whether the map contains any key-value pairs.
4. **get(k)** . If there is a key-value pair (k, v) in the map, return v; otherwise, return **null**.
5. **remove(k)**  . If there is a key-value pair in the map with key k, remove the pair.   
   If a pair (k, v1) was already in the map, return v1; otherwise return **null**. We rarely use this feature and just use the call as a statement.
6. **clear()**  . Remove all the pairs from the map; change the size of the map to 0.

Interface Map more operations. For example, you can ask whether a map contains a key, or contains a value. You can get the set of all keys, and a collection of all values. There are also ways to process all keys, or all values, using a foreach statement. If you feel you need them, look them up in the Java API documentation for interface Map.

**Example code**

The code snippet in the box to the right does the following:

Map<String, Integer> index=

new HashMap<>();

index.put("abort", 114);

index.put("abs", 314);

index.put("abstraction", 149);

index.put("ambiguity", 308);

1. Create a new HashMap and store it in variable index;
2. Add four key-value pairs to the map.

The second box contains JUnit testing code that illustrates the following:

Line 1 shows that method size returns the size of the map.  
Line 2 uses method get to retrieve the value corresponding to a key.  
Line 3 illustrates that **null** is returned when the argument of a call on  
 get is not in the set of keys.  
Lines 4..5 illustrate that put returns the old value corresponding to its  
 key parameter. The previous pair was ("abs", 314), and 314   
 was stored in d.  
Lines 4 and 6 show that put replaced the old value, 314, with the new  
 value, 80.

1. assertEquals(4, index.size());

2. assertEquals(314, index.get("abs"));

3. assertEquals(**null**, index.get(55));

4. int d= index.put("abs", 80);

5. assertEquals(314, d);  
6. assertEquals(80, index.get("abs"));

**Useful examples of Maps**

Most of the examples of Map and HashMap that you see as you look for them on the web have keys and values that are of types String or Integer. However, others can be just as useful, as our second example below shows. Whenever you have pairings of keys and values that require fast, efficient changing of pairs and efficient lookup, think of using a Map, with implementation HashMap or another one.

Map<String, String>

**1. zip codes to cities**. For example, the pair ("14853", "Ithaca") could be in it.

**2. Dates to times.** What about mapping a date to the time of day when the sun sets on that date (at a particular location)?

Map<Date, Time>

**3. Making priority-change in a heap efficient**

This example shows how an operation in a data structure can be made more efficient by introducing a HashMap. We assume knowledge of the data structure *heap*. Read this only if you have that knowledge. To the right, is a barebones view of a class that implements a max-heap. The values in the heap are of type E, and a priority, a double, is associated with each heap value. ArrayList b contains the values (and their priorities) in the heap.

class MaxHeap<E> {

class Element {

E value;

double priority;

…

}

ArrayList<Element> b;

Map<E, Integer> m;

}

For example, suppose E is Integer and ArrayList b contains the following, which represents the heap shown to the right under class MaxHeap.

[(4, 6.1), (9, 4.2), (7, 3.6)]

Now consider implementing a method changePriority(v, p) , which is to change the priority of value v in the heap to p. The implementation of this method   
must fine the node in the heap with value v, which can take time proportional to the   
size of the heap. NO GOOD! We must find a way to implement this method in   
expected logarithmic (in heap size) time, just like insert and poll are logarithmic.

m contains the pairs

(4, 0), (9, 1), (7, 3.6)

(4, 6.1)

(9, 4.2)

(7, 3.6)

To do this, we add an additional data structure, Map m, shown in red in the box   
above. Here’s its definition:

For each value v in the heap, m contains the pair (v, i) where i is the index of v in ArrayList b.

Therefore, to find the index of v in the heap, use m.get(v). That operation takes constant expected time.

Many methods in class MaxHeap are now more complicated, because the definition of m has to be maintained.

1. In our example of an index, we cannot have these two key-value pairs in the map: (abort, 114) and (abort, 205). To allow several page numbers for the same word, redefine the values to be lists, like this:  
    Map <String, List<Integer>> index. [↑](#footnote-ref-1)