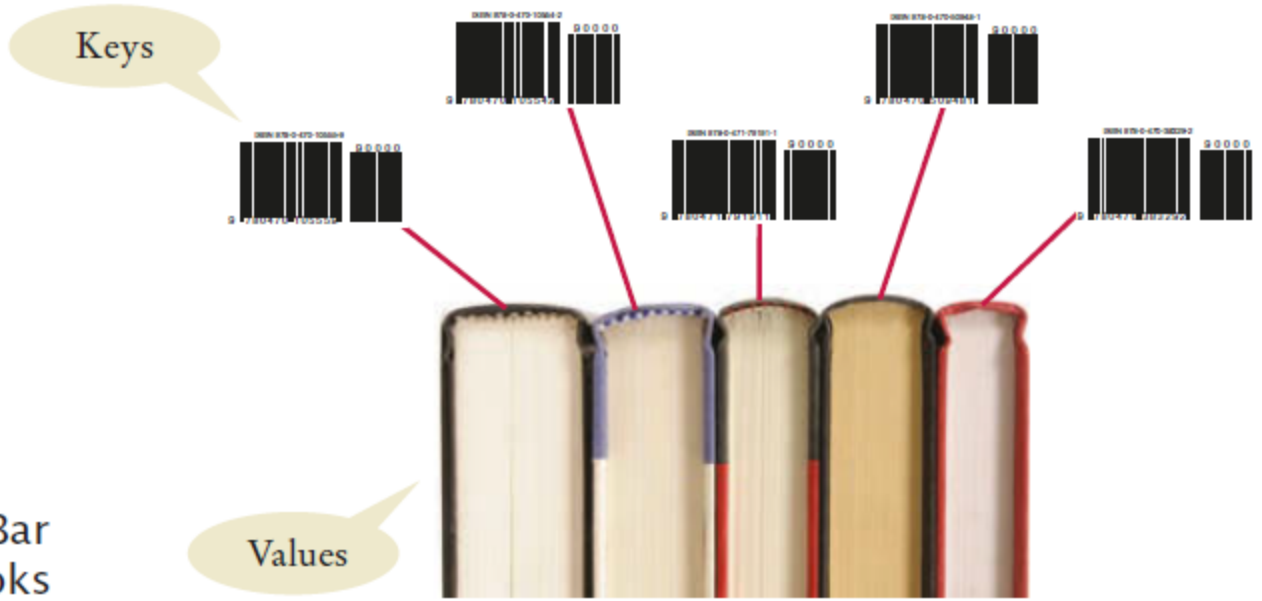


Figure 5
A Map from Bar Codes to Books

A map keeps associations between key and value objects.

The `HashSet` and `TreeSet` classes both implement the `Set` interface.

Set implementations arrange the elements so that they can locate them quickly.

You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

When you construct a `HashSet` or `TreeSet`, store the reference in a `Set` variable.

```
Set<String> names = new HashSet<>();
```

or

```
Set<String> names = new TreeSet<>();
```

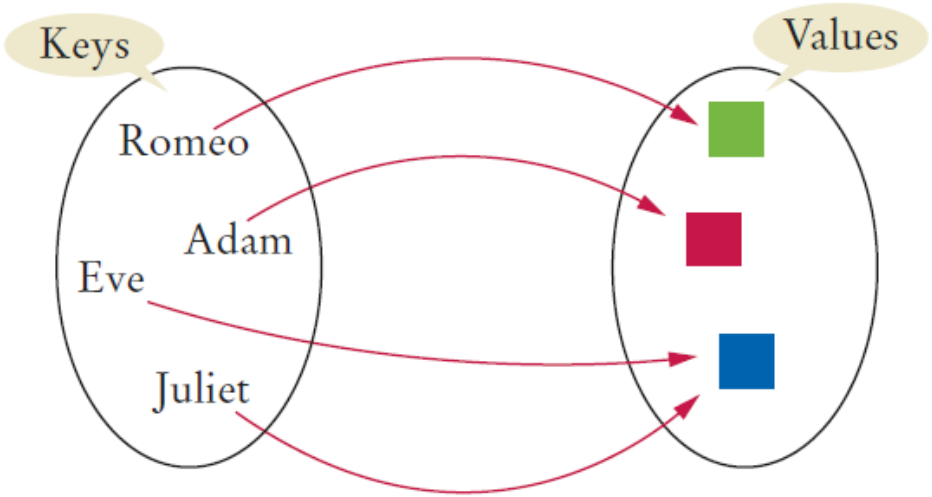
Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names) {`
`  . . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

The HashMap and TreeMap classes both implement the Map interface.



A Map

In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their **keys**. For this to work, it must be possible to compare the keys to one another.

This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys.
(The *Comparator* can be provided as a parameter to the *TreeMap* constructor.)

Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` (or `compare()`) method is used to decide whether two keys are to be considered the same.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*. Then the following are some of the methods that are defined for `map`:

- `map.get(key)` — returns the object of type *V* that is associated by the map to the `key`. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)
- `map.put(key,value)` — Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.
- `map.putAll(map2)` — if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` — if `map` associates a value to the specified `key`, that association is removed from the map.
- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key.

If `map` is a variable of type *Map<K,V>*, then

The value returned by `map.keySet()` is a "view" of keys in the map implements the *Set<K>* interface

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn.

`map.values()` returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements.