Sets (11.2)

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)

- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order.
**Set methods**

In Java, Set is an interface that allows you to call the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>adds the given value to the set. If the value is already in the set, nothing happens</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>returns <code>true</code> if the given value is found in this set</td>
</tr>
<tr>
<td><code>remove(value)</code></td>
<td>removes the given value from the set</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the set</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns <code>true</code> if the set's size is 0</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
Set implementation

- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table"; extremely fast for all operations; elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree"; very fast for all operations; elements are stored in sorted order

```java
Set<Integer> numbers = new TreeSet<Integer>();
Set<String> words = new HashSet<String>();
```
The "for each" loop (7.1)

```java
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a Set, List, array, or other collection

```java
Set<Double> grades = new HashSet<Double>();
...

for (double grade : grades) {
    System.out.println("Student's grade: "+ grade);
}
```

- needed because sets have no indexes; can't get element i
Maps (11.3)

- **map**: Holds a set of key-value pairs, where each key is unique
  a.k.a. "dictionary", "associative array", "hash"

```java
map.get("the")
```

```
key   | value
---|---
"at"  | 43
"you" | 22
"in"  | 37
"why" | 14
"me"  | 22
"the" | 56
```
Maps (11.3)

- **map**: Holds a set of unique keys and a collection of values, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- basic map operations:
  - **put**(key, value): Adds a mapping from a key to a value.
  - **get**(key): Retrieves the value mapped to the key.
  - **remove**(key): Removes the given key and its mapped value.

```
myMap.get("Aug") returns 37.3
```
Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

```
// key     value
put("Suzy", "206-685-2181")
```

- Later, we can supply only the key and get back the related value:
  - Allows us to ask: *What is Suzy's phone number?*
Map implementation

- Java provides the Map interface in java.util

- Map is implemented by the HashMap and TreeMap classes
  - HashMap: implemented using a "hash table"; extremely fast: keys are stored in unpredictable order
  - TreeMap: implemented as a linked "binary tree" structure; very fast: keys are stored in sorted order

- Maps require 2 type params: one for keys, one for values.

```java
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();

// maps from Integer keys to String values
Map<Integer, String> words = new TreeMap<Integer, String>();
```
## Map methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>put(key, value)</strong></td>
<td>adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one</td>
</tr>
<tr>
<td><strong>get(key)</strong></td>
<td>returns the value mapped to the given key (null if not found)</td>
</tr>
<tr>
<td><strong>containsKey(key)</strong></td>
<td>returns true if the map contains a mapping for the given key</td>
</tr>
<tr>
<td><strong>remove(key)</strong></td>
<td>removes any existing mapping for the given key</td>
</tr>
<tr>
<td><strong>clear()</strong></td>
<td>removes all key/value pairs from the map</td>
</tr>
<tr>
<td><strong>size()</strong></td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><strong>isEmpty()</strong></td>
<td>returns true if the map's size is 0</td>
</tr>
<tr>
<td><strong>toString()</strong></td>
<td>returns a string such as &quot;{a=90, d=60, c=70}&quot;</td>
</tr>
<tr>
<td><strong>keySet()</strong></td>
<td>returns a set of all keys in the map</td>
</tr>
<tr>
<td><strong>values()</strong></td>
<td>returns a collection of all values in the map</td>
</tr>
<tr>
<td><strong>putAll(map)</strong></td>
<td>adds all key/value pairs from the given map to this map</td>
</tr>
<tr>
<td><strong>equals(map)</strong></td>
<td>returns true if given map has the same mappings as this one</td>
</tr>
</tbody>
</table>
keySet and values

- **keySet** method returns a **Set** of all keys in the map
  - can loop over the keys in a for-each loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57); // ages.keySet() returns Set<String>
for (String name : ages.keySet()) {
    int age = ages.get(name);
    System.out.println(name + " -> " + age); // Marty -> 19, Geneva -> 2, Vicki -> 57
}
```

- **values** method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - no easy way to get from a value to its associated key(s)