CSE 143
Lecture 15

Sets and Maps

reading: 11.1 - 11.3; 13.2

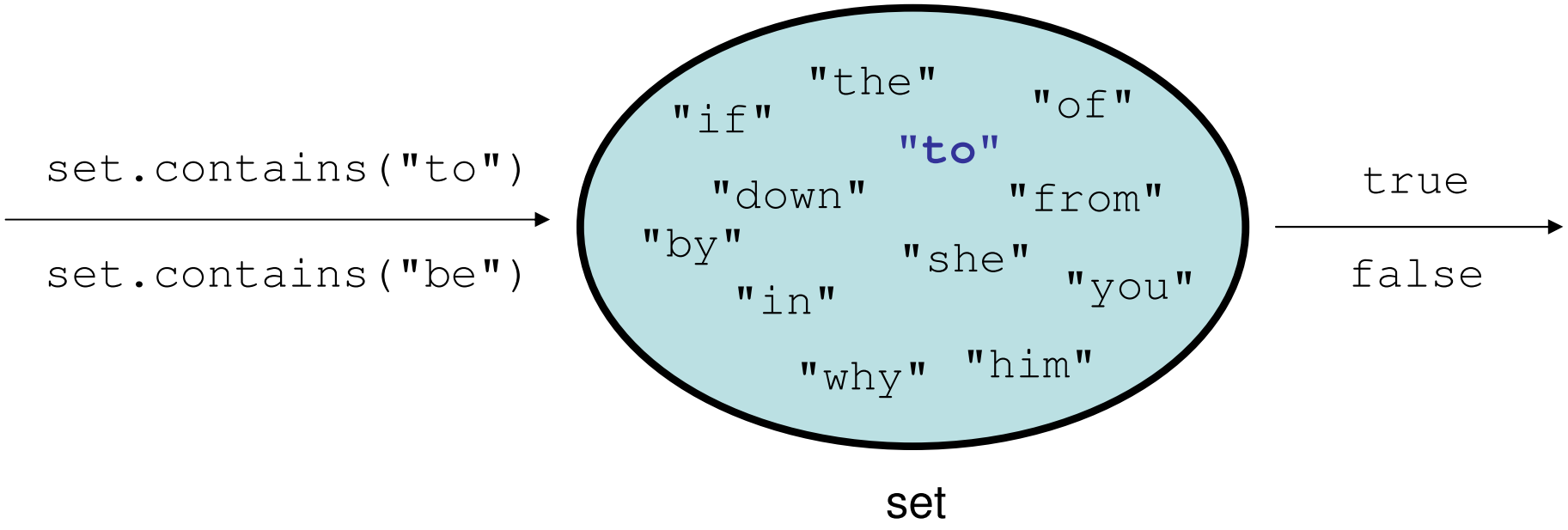
slides created by Marty Stepp
http://www.cs.washington.edu/143/

Exercise

- Write a program that counts the number of unique words in a large text file (say, *Moby Dick* or the King James Bible).
  - Store the words in a collection and report the # of unique words.
  - Once you've created this collection, allow the user to search it to see whether various words appear in the text file.
- What collection is appropriate for this problem?

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` 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" array;
    very fast: **O(1)** for all operations
    elements are stored in unpredictable order

  - `TreeSet`: implemented using a "binary search tree";
    pretty fast: **O(log N)** for all operations
    elements are stored in sorted order

  - `LinkedHashSet`: **O(1)** but stores in order of insertion

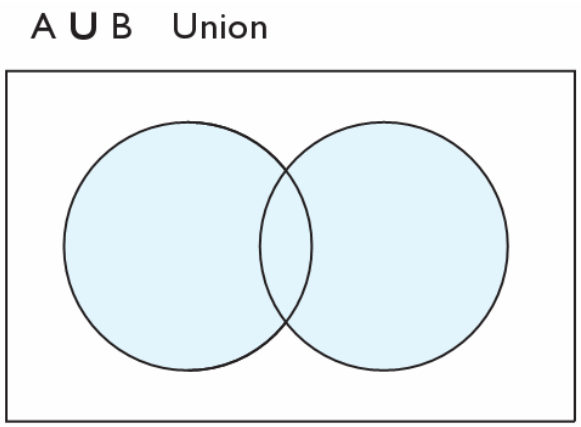
Set methods

```
List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>();      // empty
Set<String> set2 = new HashSet<String>(list);
```

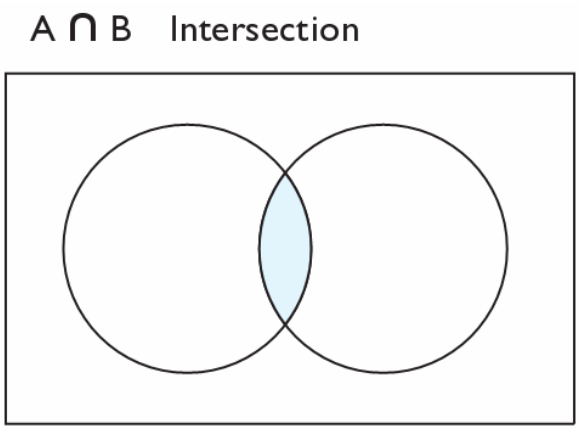
- can construct an empty set, or one based on a given collection

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in this set
`remove(`**value**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the set's size is 0
`toString()`	returns a string such as `"[3, 42, -7, 15]"`

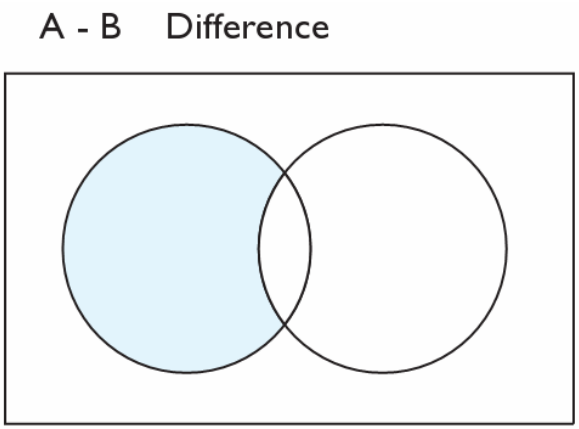
Set operations



`addAll`



`retainAll`



`removeAll`

`addAll(`**collection**`)`	adds all elements from the given collection to this set
`containsAll(`**coll**`)`	returns `true` if this set contains every element from given set
`equals(`**set**`)`	returns `true` if given other set contains the same elements
`iterator()`	returns an object used to examine set's contents *(seen later)*
`removeAll(`**coll**`)`	removes all elements in the given collection from this set
`retainAll(`**coll**`)`	removes elements *not* found in given collection from this set
`toArray()`	returns an array of the elements in this set

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

The "for each" loop (7.1)

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

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

```
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`

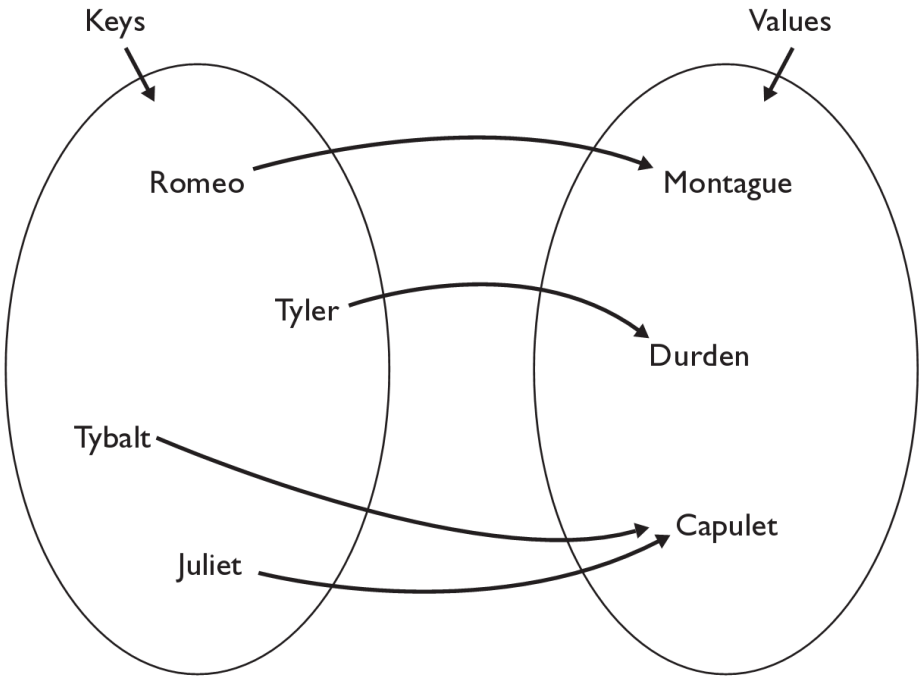
Exercise

- Write a program to count the number of occurrences of each unique word in a large text file (e.g. *Moby Dick*).
  - Allow the user to type a word and report how many times that word appeared in the book.
  - Report all words that appeared in the book at least 500 times, in alphabetical order.
- What collection is appropriate for this problem?

The Map ADT

- **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.

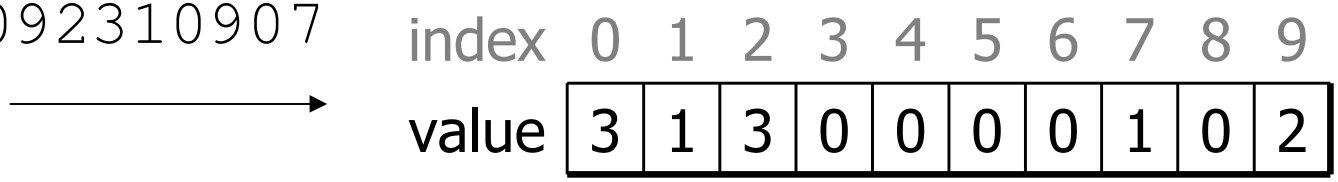


`myMap.get("Juliet")` returns `"Capulet"`

Maps and tallying

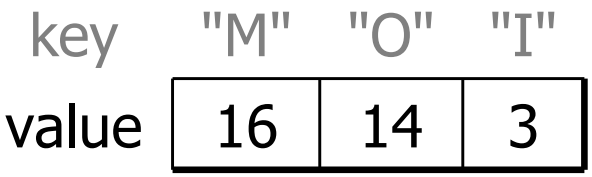
- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`

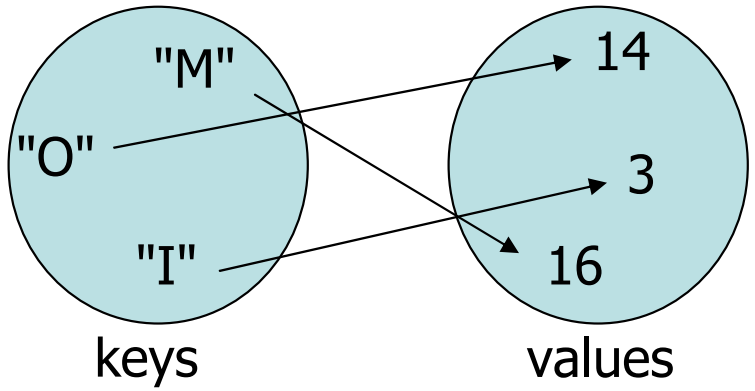
- recall previous tallying examples from CSE 142
  - count digits: `22092310907`




```
// (M)cCain, (O)bama, (I)ndependent
```

  - count votes: `"MOOOOOOMMMMMOOOOOOMOMMIMOMMIMOMMIO"`





`Map` implementation

- in Java, maps are represented by `Map` interface in `java.util`

- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: **O(1)** ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **O(log N)** ; keys are stored in sorted order

  – A map requires 2 type parameters: one for keys, one for values.

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

Map methods

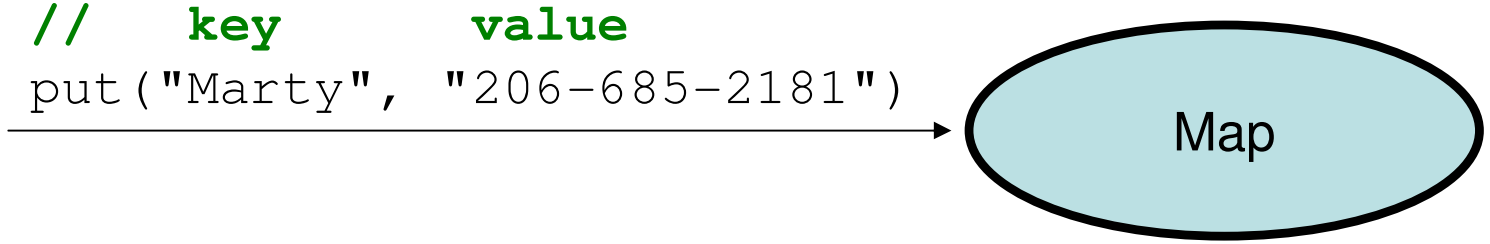
`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

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).

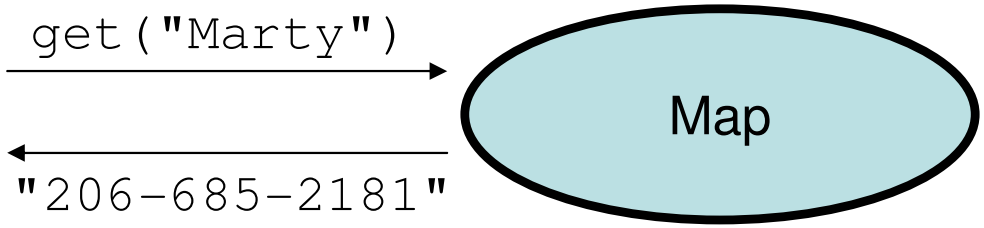
```
//    key          value
put("Marty", "206-685-2181")
```



  - Later, we can supply only the key and get back the related value:

    Allows us to ask: *What is Marty's phone number?*

```
get("Marty")
```

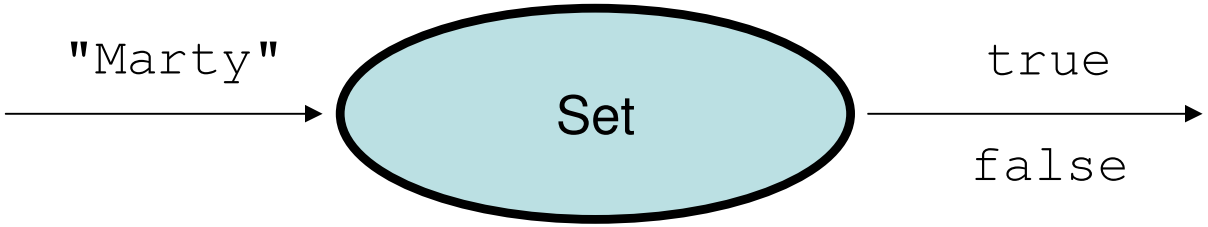


Exercise solution

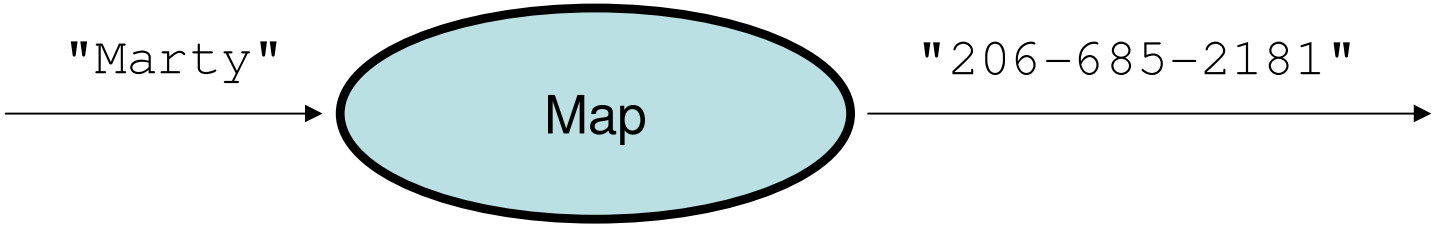
```
// read file into a map of [word --> number of occurrences]
Map<String, Integer> wordCount = new HashMap<String, Integer>();
Scanner input = new Scanner(new File("mobydick.txt"));
while (input.hasNext()) {
    String word = input.next();
    if (wordCount.containsKey(word)) {
        // seen this word before; increase count by 1
        int count = wordCount.get(word);
        wordCount.put(word, count + 1);
    } else {
        // never seen this word before
        wordCount.put(word, 1);
    }
}

Scanner console = new Scanner(System.in);
System.out.print("Word to search for? ");
String word = console.next();
System.out.println("appears " + wordCount.get(word) + " times.");
```

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Marty found in the set? (true/false)*



  - *Map: What is Marty's phone number?*



keySet and values

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a foreach 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.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {              // Geneva -> 2
    int age = ages.get(name);                    // Marty -> 19
    System.out.println(name + " -> " + age);     // 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)

Problem: opposite mapping

- It is legal to have a map of sets, a list of lists, etc.
- Suppose we want to keep track of each TA's GPA by name.

```
Map<String, Double> taGpa = new HashMap<String, Double>();
taGpa.put("Jared", 3.6);
taGpa.put("Alyssa", 4.0);
taGpa.put("Steve", 2.9);
taGpa.put("Stef", 3.6);
taGpa.put("Rob", 2.9);
...
System.out.println("Jared's GPA is " +
                   taGpa.get("Jared"));   // 3.6
```

- This doesn't let us easily ask which TAs got a given GPA.
  - How would we structure a map for that?

Reversing a map

- We can reverse the mapping to be from GPAs to names.

```
Map<Double, String> taGpa = new HashMap<Double, String>();
taGpa.put(3.6, "Jared");
taGpa.put(4.0, "Alyssa");
taGpa.put(2.9, "Steve");
taGpa.put(3.6, "Stef");
taGpa.put(2.9, "Rob");
...
System.out.println("Who got a 3.6? " +
                   taGpa.get(3.6));   // ???
```

- What's wrong with this solution?
  - More than one TA can have the same GPA.
  - The map will store only the last mapping we add.

Proper map reversal

- Really each GPA maps to a *collection* of people.

```
Map<Double, Set<String>> taGpa =
        new HashMap<Double, Set<String>>();
taGpa.put(3.6, new TreeSet<String>());
taGpa.get(3.6).add("Jared");
taGpa.put(4.0, new TreeSet<String>());
taGpa.get(4.0).add("Alyssa");
taGpa.put(2.9, new TreeSet<String>());
taGpa.get(2.9).add("Steve");
taGpa.get(3.6).add("Stef");
taGpa.get(2.9).add("Rob");
...
System.out.println("Who got a 3.6? " +
                   taGpa.get(3.6));   // [Jared, Stef]
```

  - must be careful to initialize the set for a given GPA before adding

Exercises

- Modify the word count program to print every word that appeared in the book at least 1000 times, in sorted order from least to most occurrences.

- Write a program that reads a list of TA names and quarters' experience, then prints the quarters in increasing order of how many TAs have that much experience, along with their names.

```
Allison 5                    1 qtr: [Brian]
Alyssa 8          ------->   2 qtr: ...
Brian 1                      5 qtr: [Allison, Kasey]
Kasey 5
...
```