CSE 143
Lecture 7

Sets and Maps

reading: 11.2 - 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?

Empirical analysis (13.2)

Running a program and measuring its performance

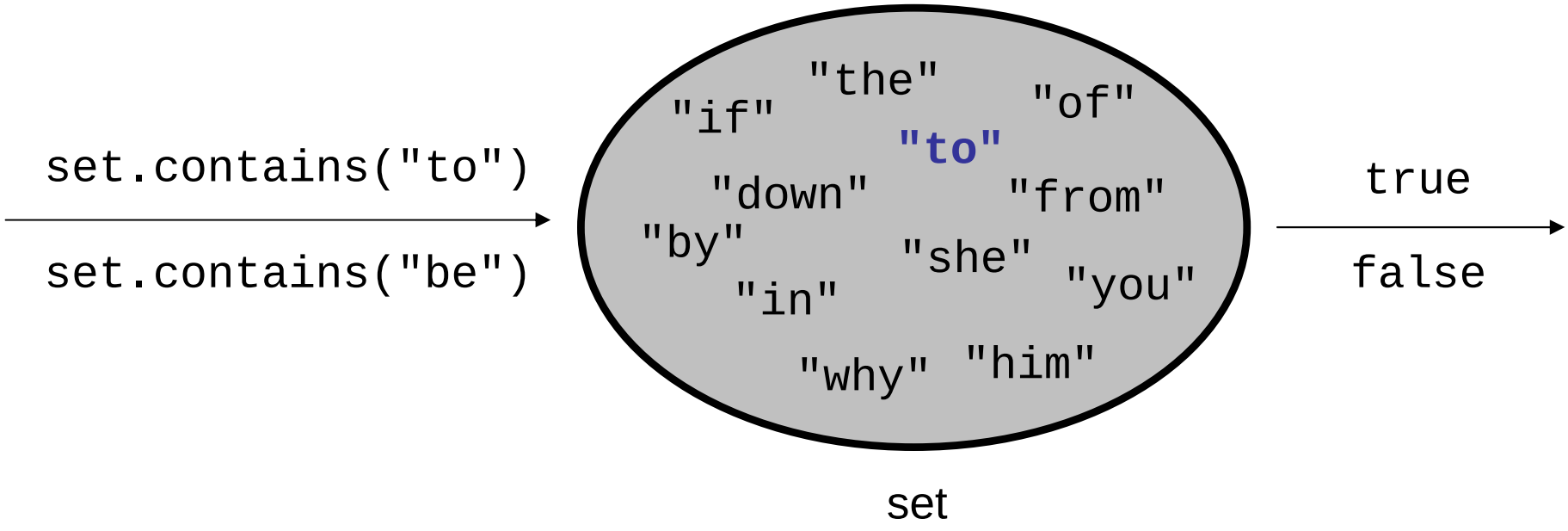
`System.currentTimeMillis()`

- Returns an integer representing the number of milliseconds that have passed since 12:00am, January 1, 1970.
  - The result is returned as a value of type `long`, which is like `int` but with a larger numeric range (64 bits vs. 32).
- Can be called twice to see how many milliseconds have elapsed between two points in a program.

- How much time does it take to store *Moby Dick* into a `List`?

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

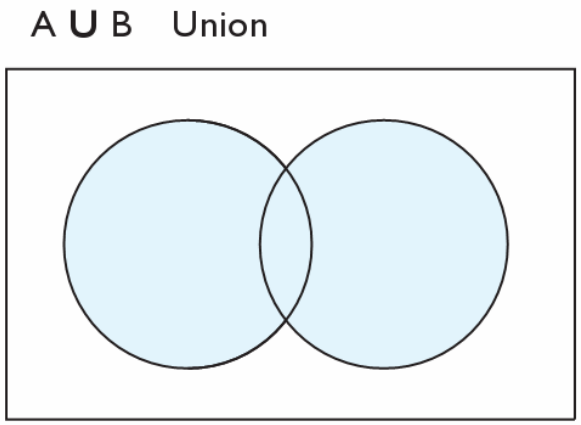
Set methods

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

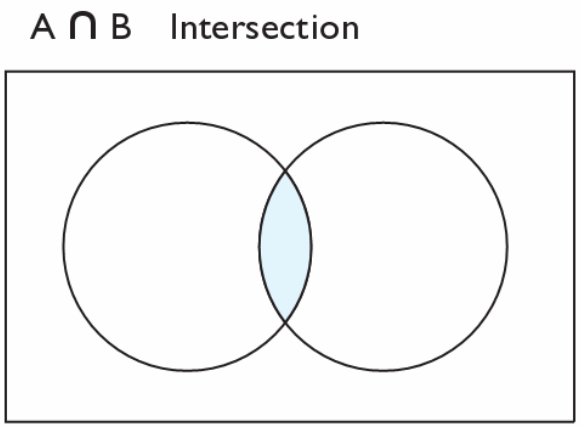
- can construct an empty set, or one based on a given collection

`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]"`

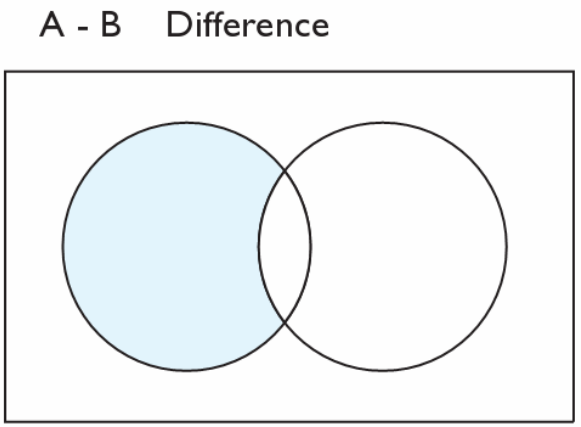
Set operations



`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

- `Sets` do not use indexes; you cannot `get` element `i`

- `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]
```

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

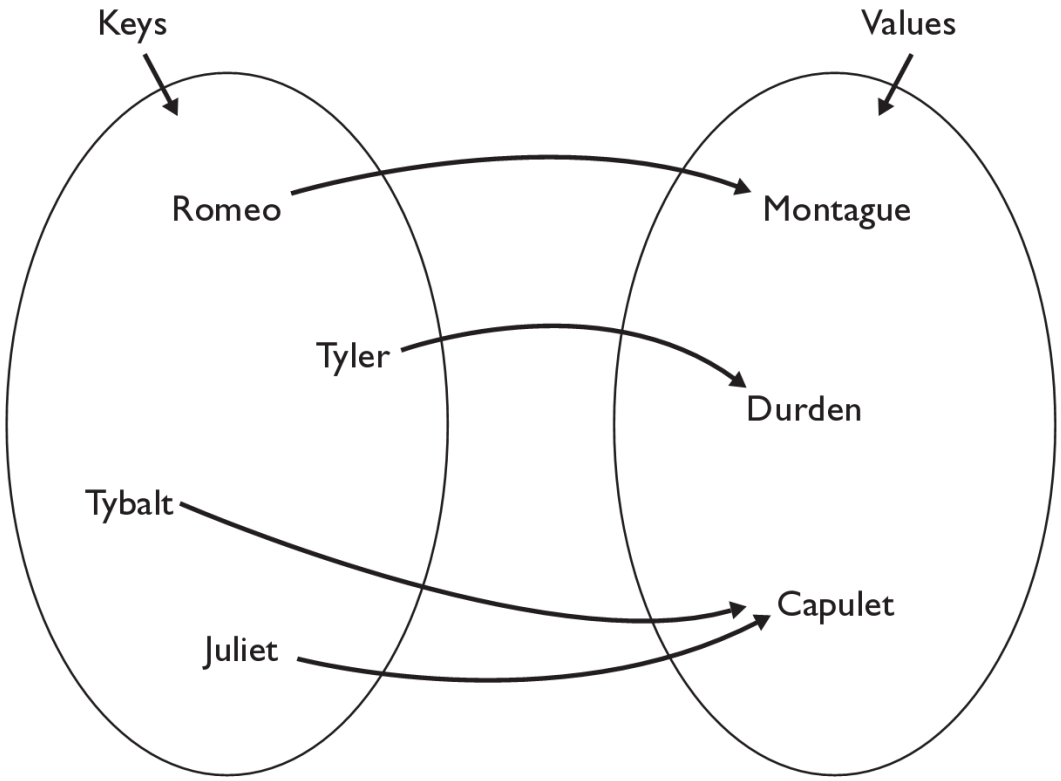
Exercise

- Modify your program to count the number of occurrences of each word in the book.
  - 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?

Maps

- **map**: An ADT holding 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.

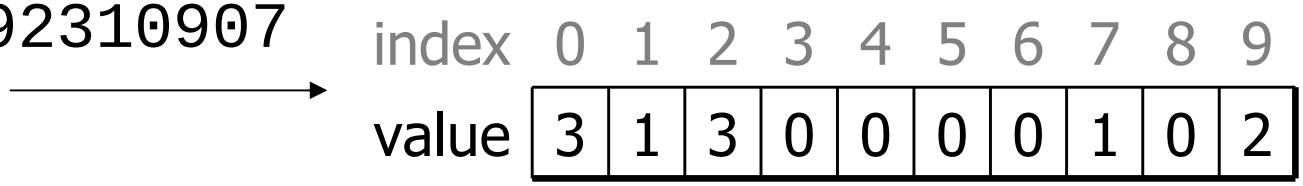


`map.get("Juliet")` returns `"Capulet"`

Maps and tallying

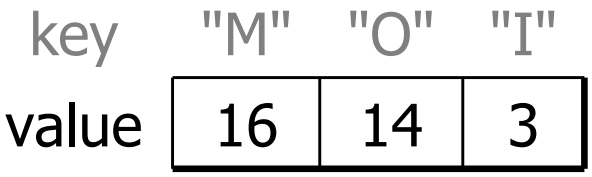
- a map can be thought of as generalization of an array
  - the "index" (key) doesn't have to be an `int`

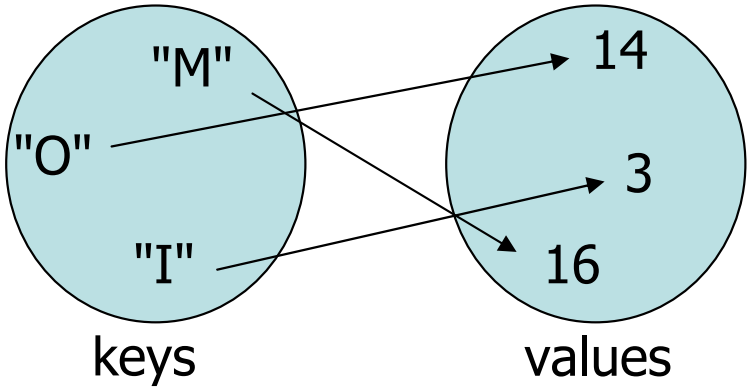
- recall previous tallying examples from CSE 142
  - count digits: `22092310907`




```
// (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 `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using a "hash table" array;
    very fast: **O(1)** ; keys are stored in unpredictable order
  - `TreeMap`: implemented using a "binary search tree";
    pretty 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>();
```

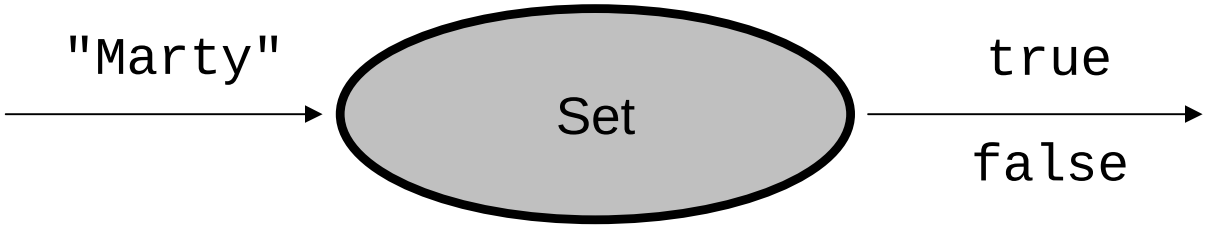
Map methods

`put(`**`key`**`, `**`value`**`)`	adds a mapping from the given key to the given value
`get(`**`key`**`)`	returns the value mapped to the given key (`null` if none)
`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}"`

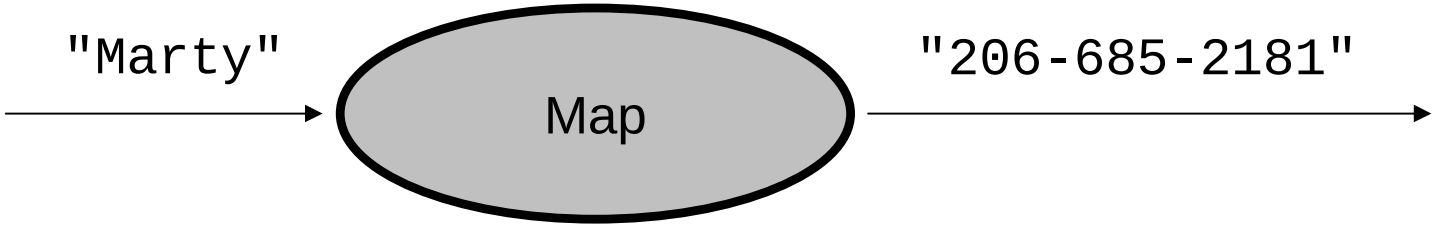
`keySet()`	returns a `Set` of all keys in the map
`values()`	returns a `Collection` of all values in the map
`putAll(`**`map`**`)`	adds all key/value pairs from the given map to this map
`equals(`**`map`**`)`	returns `true` if given map has same mappings as this one

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - We are remembering one related piece of information about every element: *Is "Marty" found in the set? (true/false)*



  - A map allows the related piece of information to be something other than a boolean: *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 HashMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57);
for (String name : ages.keySet()) {              // Geneva -> 2
    int age = ages.get(age);                     // 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)