

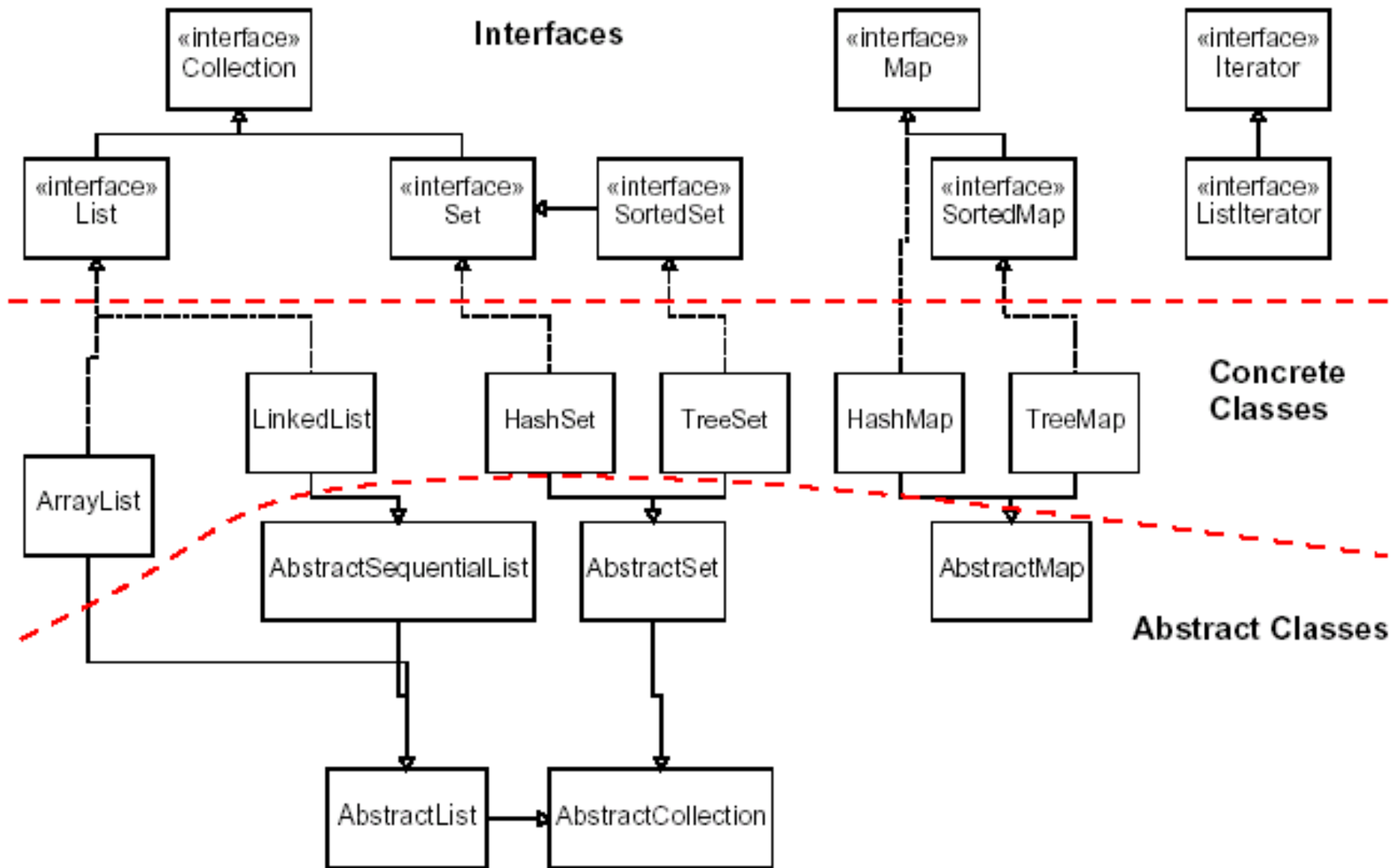
Building Java Programs

Chapter 11

Java Collections Framework

Copyright (c) Pearson 2013.
All rights reserved.

Java collections framework



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

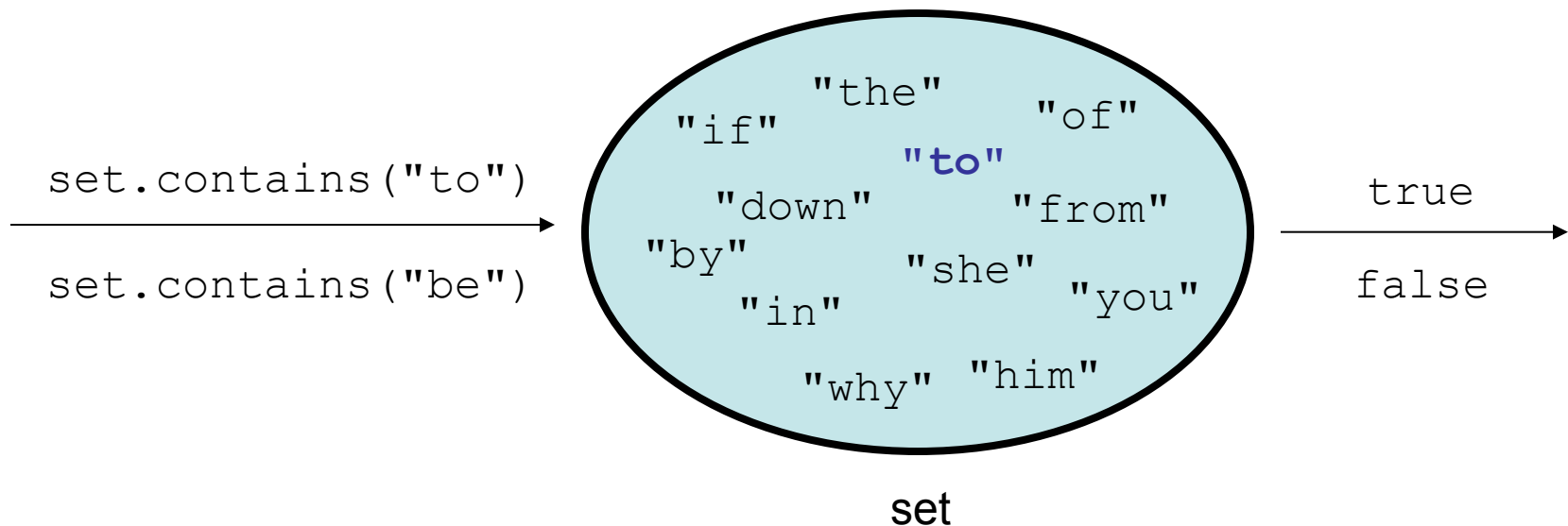
Running a program and measuring its performance

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

Set methods

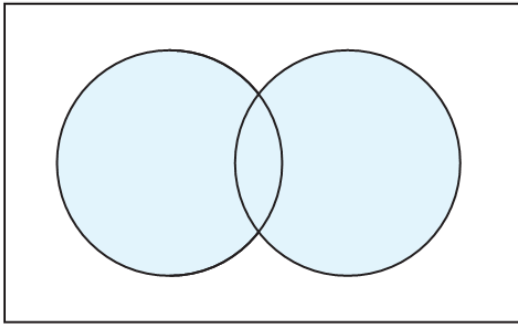
```
List<String> list = new ArrayList<String>();  
...  
Set<Integer> set = new TreeSet<Integer>(); // empty  
Set<String> set2 = new HashSet<String>(list);
```

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

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

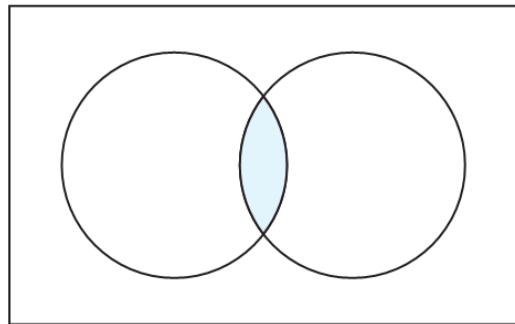
Set operations

$A \cup B$ Union



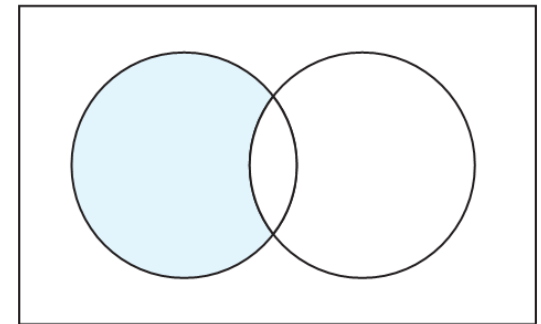
`addAll`

$A \cap B$ Intersection



`retainAll`

$A - B$ Difference



`removeAll`

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

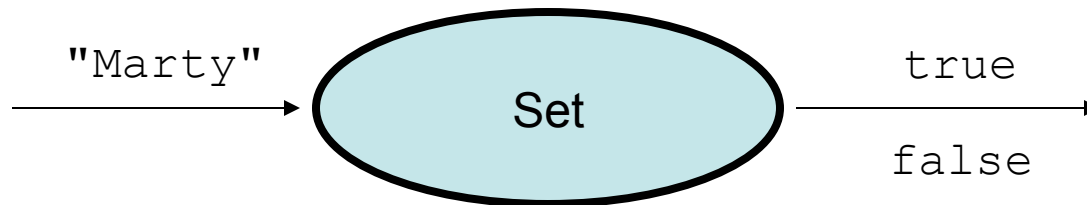
```
Set<Double> grades = new HashSet<Double>();  
...
```

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

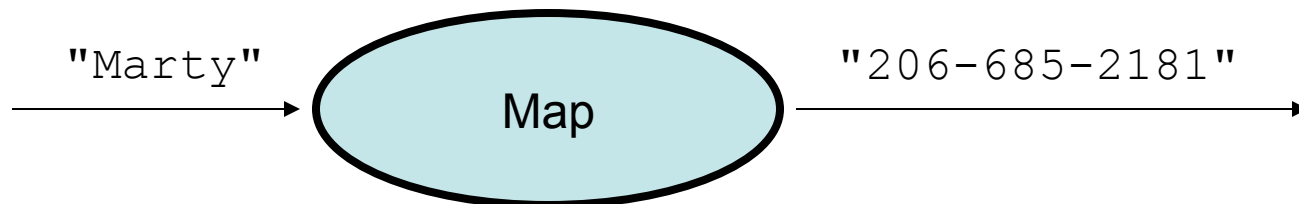
– needed because sets have no indexes; can't `get` element `i`

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.

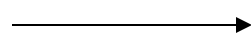
```
Allison 5
```

```
Alyssa 8
```

```
Brian 1
```

```
Kasey 5
```

```
...
```



```
1 qtr: [Brian]
```

```
2 qtr: ...
```

```
5 qtr: [Allison, Kasey]
```


Iterators

reading: 11.1; 15.3; 16.5

Examining sets and maps

- elements of Java `Set`s and `Map`s can't be accessed by index
 - must use a "foreach" loop:

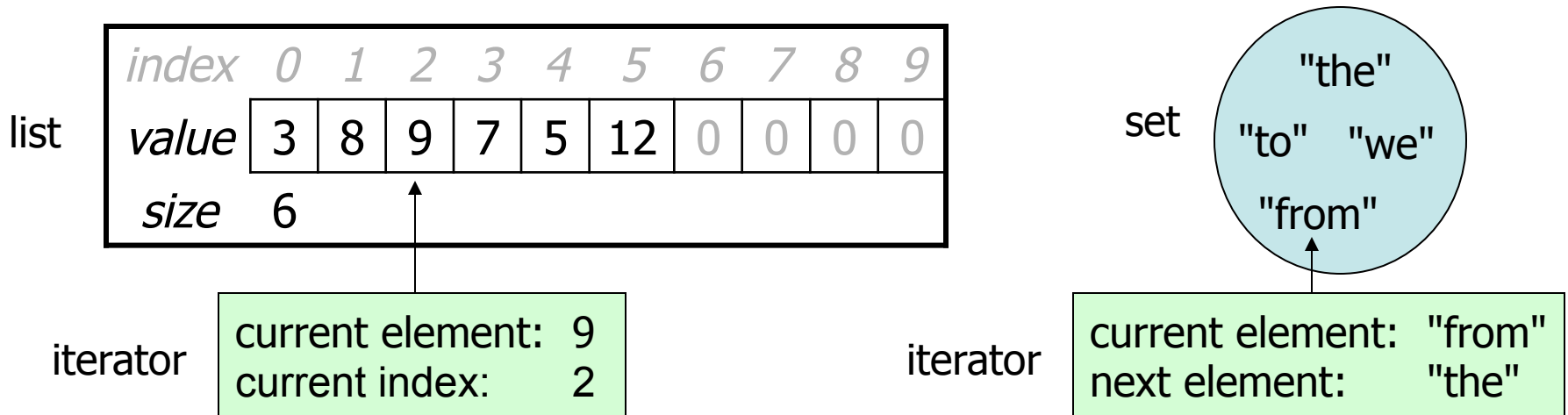
```
Set<Integer> scores = new HashSet<Integer>();  
for (int score : scores) {  
    System.out.println("The score is " + score);  
}
```

- Problem: foreach is read-only; cannot modify set while looping

```
for (int score : scores) {  
    if (score < 60) {  
        // throws a ConcurrentModificationException  
        scores.remove(score);  
    }  
}
```

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
 - Remembers a position, and lets you:
 - get the element at that position
 - advance to the next position
 - remove the element at that position



Iterator methods

<code>hasNext()</code>	returns <code>true</code> if there are more elements to examine
<code>next()</code>	returns the next element from the collection (throws a <code>NoSuchElementException</code> if there are none left to examine)
<code>remove()</code>	removes the last value returned by <code>next()</code> (throws an <code>IllegalStateException</code> if you haven't called <code>next()</code> yet)

- Iterator interface in `java.util`
 - every collection has an `iterator()` method that returns an iterator over its elements

```
Set<String> set = new HashSet<String>();  
...  
Iterator<String> itr = set.iterator();  
...
```

Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);    // Kim
scores.add(87);
scores.add(43);   // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);    // [72, 87, 94]
```

Iterator example 2

```
Map<String, Integer> scores = new TreeMap<String, Integer>();
scores.put("Kim", 38);
scores.put("Lisa", 94);
scores.put("Roy", 87);
scores.put("Marty", 43);
scores.put("Marisa", 72);
...
```

```
Iterator<String> itr = scores.keySet().iterator();
while (itr.hasNext()) {
    String name = itr.next();
    int score = scores.get(name);
    System.out.println(name + " got " + score);

    // eliminate any failing students
    if (score < 60) {
        itr.remove(); // removes name and score
    }
}
System.out.println(scores); // {Lisa=94, Marisa=72, Roy=87}
```

Exercise

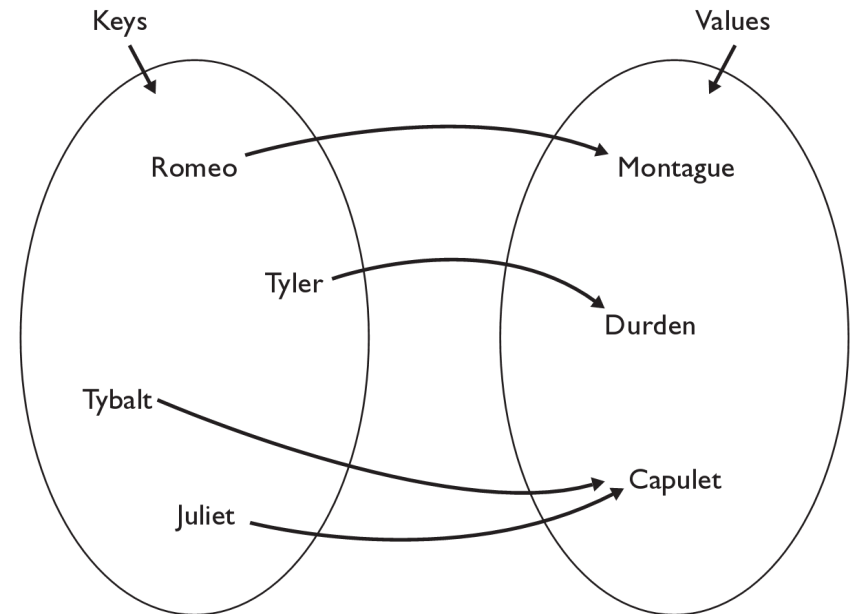
- Modify the Book Search program from last lecture to eliminate any words that are plural or all-uppercase from the collection.
- Modify the TA quarters experience program so that it eliminates any TAs with 3 quarters or fewer of experience.

Exercise

- Write a program to count the occurrences of each word in a large text file (e.g. *Moby Dick* or the King James Bible).
 - 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.
- How will we store the data to solve 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.

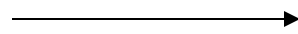


`myMap.get("Juliet")` returns "Capulet"

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

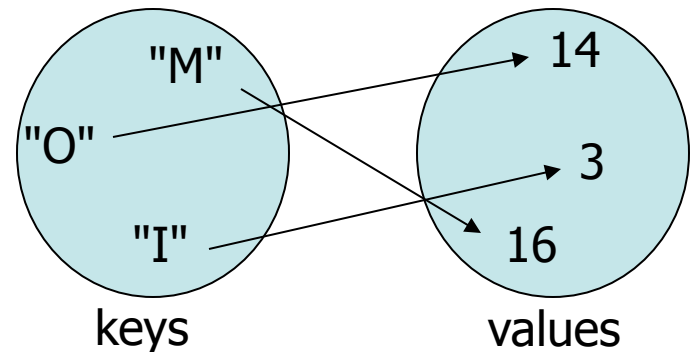


index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

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

– count votes: "MOOOOOOMMMMMMOOOOOOOMOMMMIMOMMMIMOMMMIO"

key	"M"	"O"	"I"
value	16	14	3



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>();
```

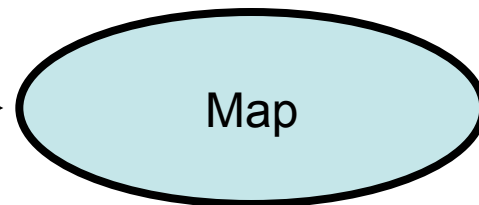
Map methods

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

Using maps

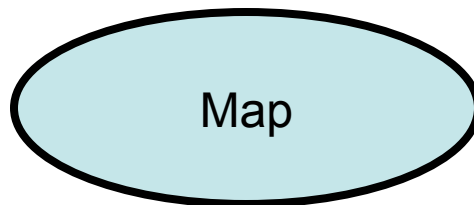
- A map allows you to get from one half of a pair to the other.
 - Remembers one piece of information about every index (key).

```
// 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")  
←  
"206-685-2181"
```



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.");
```

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()) {
    int age = ages.get(name);
    System.out.println(name + " -> " + age);
}
```

// Geneva -> 2
// Marty -> 19
// Vicki -> 57

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

Languages and Grammars

Languages and grammars

- (formal) **language**: A set of words or symbols.
- **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
 - describes language *syntax* (rules) but not *semantics* (meaning)
 - can be used to generate strings from a language, or to determine whether a given string belongs to a given language

Backus-Naur (BNF)

- **Backus-Naur Form (BNF)**: A syntax for describing language grammars in terms of transformation *rules*, of the form:

<symbol> ::= <expression> | <expression> ... | <expression>

- **terminal**: A fundamental symbol of the language.
- **non-terminal**: A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.
- developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

An example BNF grammar

`<s> ::= <n> <v>`

`<n> ::= Marty | Victoria | Stuart | Jessica`

`<v> ::= cried | slept | belched`

- Some sentences that could be generated from this grammar:

Marty slept

Jessica belched

Stuart cried

BNF grammar version 2

```
<s> ::= <np> <v>  
<np> ::= <pn> | <dp> <n>  
<pn> ::= Marty | Victoria | Stuart | Jessica  
<dp> ::= a | the  
<n> ::= ball | hamster | carrot | computer  
<v> ::= cried | slept | belched
```

- Some sentences that could be generated from this grammar:

```
the carrot cried  
Jessica belched  
a computer slept
```

BNF grammar version 3

```
<s> ::= <np> <v>
<np> ::= <pn> | <dp> <adj> <n>
<pn> ::= Marty | Victoria | Stuart | Jessica
<dp> ::= a | the
<adj> ::= silly | invisible | loud | romantic
<n> ::= ball | hamster | carrot | computer
<v> ::= cried | slept | belched
```

- Some sentences that could be generated from this grammar:

```
the invisible carrot cried
Jessica belched
a computer slept
a romantic ball belched
```

Grammars and recursion

```
<s> ::= <np> <v>
<np> ::= <pn> | <dp> <adjp> <n>
<pn> ::= Marty | Victoria | Stuart | Jessica
<dp> ::= a | the
<adjp> ::= <adj> <adjp> | <adj>
<adj> ::= silly | invisible | loud | romantic
<n> ::= ball | hamster | carrot | computer
<v> ::= cried | slept | belched
```

- Grammar rules can be defined *recursively*, so that the expansion of a symbol can contain that same symbol.
 - There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.

Grammar, final version

`<s> ::= <np> <vp>`

`<np> ::= <dp> <adjp> <n> | <pn>`

`<dp> ::= the | a`

`<adjp> ::= <adj> | <adj> <adjp>`

`<adj> ::= big | fat | green | wonderful | faulty | subliminal`

`<n> ::= dog | cat | man | university | father | mother | child`

`<pn> ::= John | Jane | Sally | Spot | Fred | Elmo`

`<vp> ::= <tv> <np> | <iv>`

`<tv> ::= hit | honored | kissed | helped`

`<iv> ::= died | collapsed | laughed | wept`

- Could this grammar generate the following sentences?

Fred honored the green wonderful child

big Jane wept the fat man fat

- Generate a random sentence using this grammar.

Sentence generation

