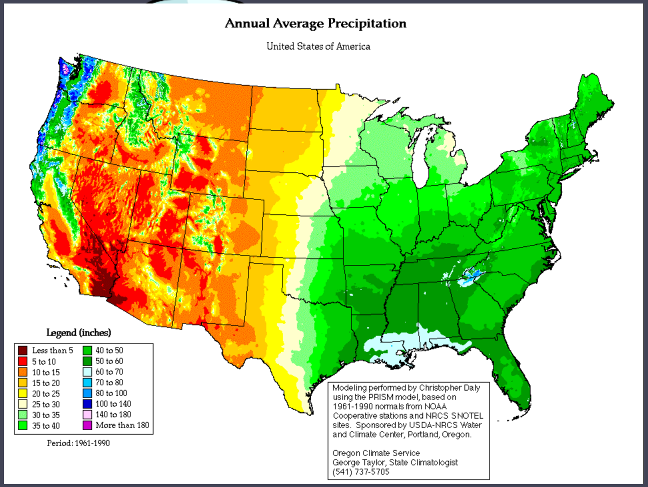
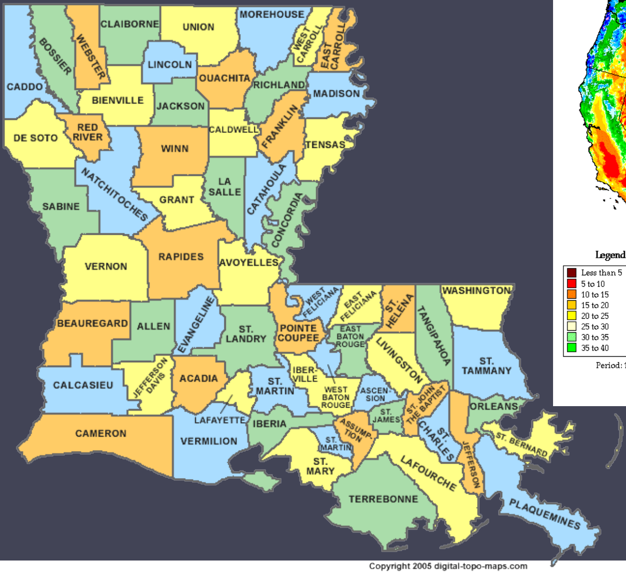




LECTURE 26

Sets and Maps

Assignment

- Read section 7.1 and 7.2
  - Do self-check exercise in section 7.1 (7.1 through 7.4)
  - Ditto for section 7.2 (7.1 through 7.3)

Before…

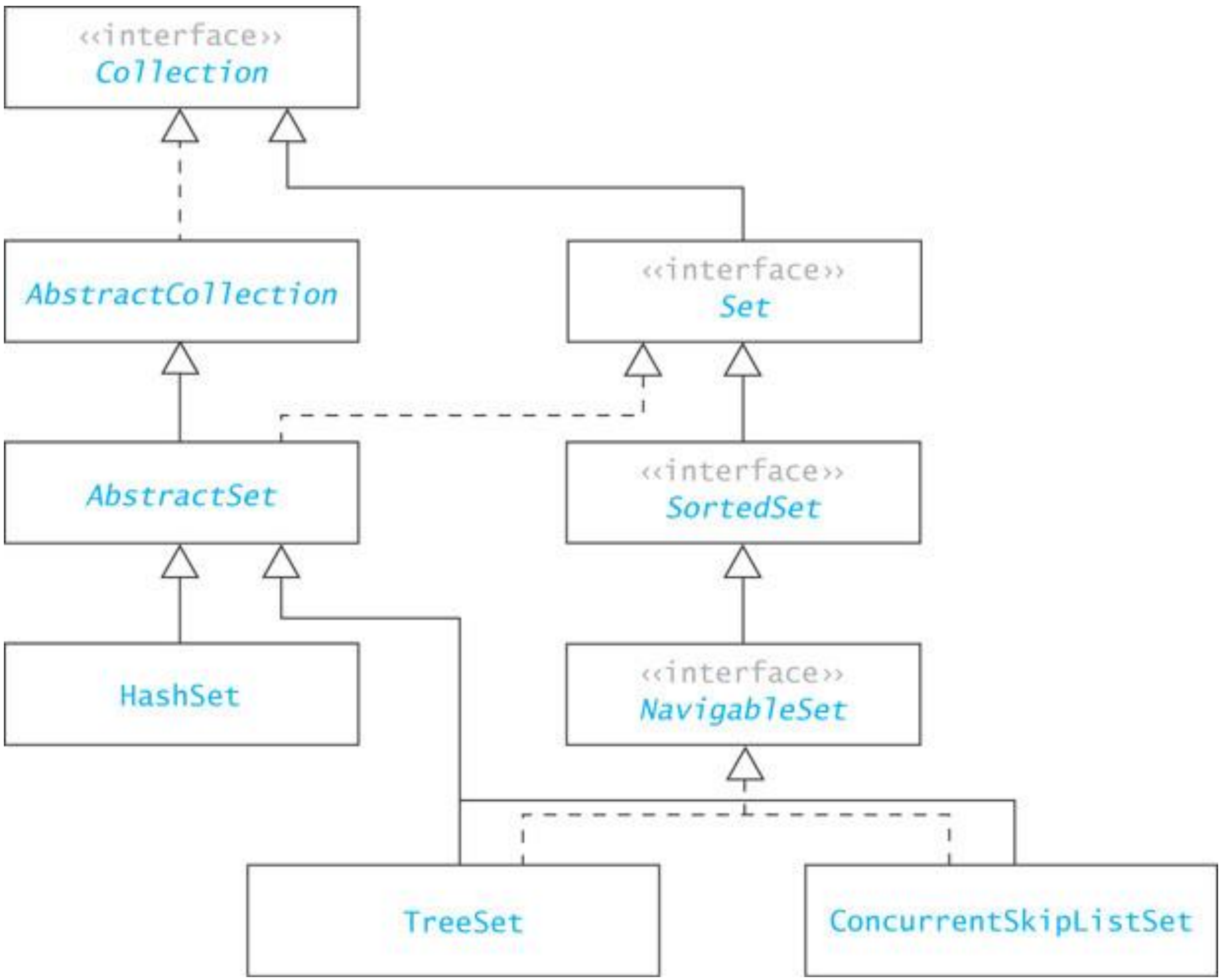
- We learned about part of the Java Collection Framework (`ArrayList` and `LinkedList`)
- The classes that implement the `List` interface are all *indexed* collections
  - An index or subscript is associated with each element
  - The element's index often reflects the relative order of its insertion in the list
  - Searching for a particular value in a list is generally O(*n*)
  - An exception is a binary search of a sorted object, which is O(log *n*)

Now

- We consider another part of the `Collection` hierarchy: the `Set` interface and the classes that implement it
- `Set` objects
  - are *not* indexed
  - do *not* reveal the order
  - do enable efficient search and retrieval of information
  - do allow removal of elements without moving other elements around

Sets and the `Set` Interface

Sets and the `Set` Interface



The `Set` Abstraction

- A set is a collection that contains no duplicate elements and at most one `null` element
  - adding `"apples"` to the set `{"apples", "oranges", "pineapples"}` results in the same set (no change)
- Operations on sets include:
  - testing for membership
  - adding elements
  - removing elements
  - union $A \cup B$
  - intersection $A \cap B$
  - difference $A - B$
  - testing for being a subset $A \subset B$

The `Set` Abstraction

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.

  Example: $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$ is $\{1, 2, 3, 4, 5, 7\}$
- The intersection of sets A, B is the set whose elements belong to both A and B.

  Example: $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$ is $\{3, 5\}$
- The difference of sets A - B is the set whose elements belong to A but not to B.

  Examples: $\{1, 3, 5, 7\} - \{2, 3, 4, 5\}$ is $\{1, 7\}$; $\{2, 3, 4, 5\} - \{1, 3, 5, 7\}$ is $\{2, 4\}$
- Set A is a subset of set B if every element of set A is also an element of set B.

  Example: $\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$ is true

The `Set` Interface and Methods

- Required methods: testing set membership, testing for an empty set, determining set size, and creating an iterator over the set
- Optional methods: adding an element and removing an element
- methods to enforce the “no duplicate members” criterion

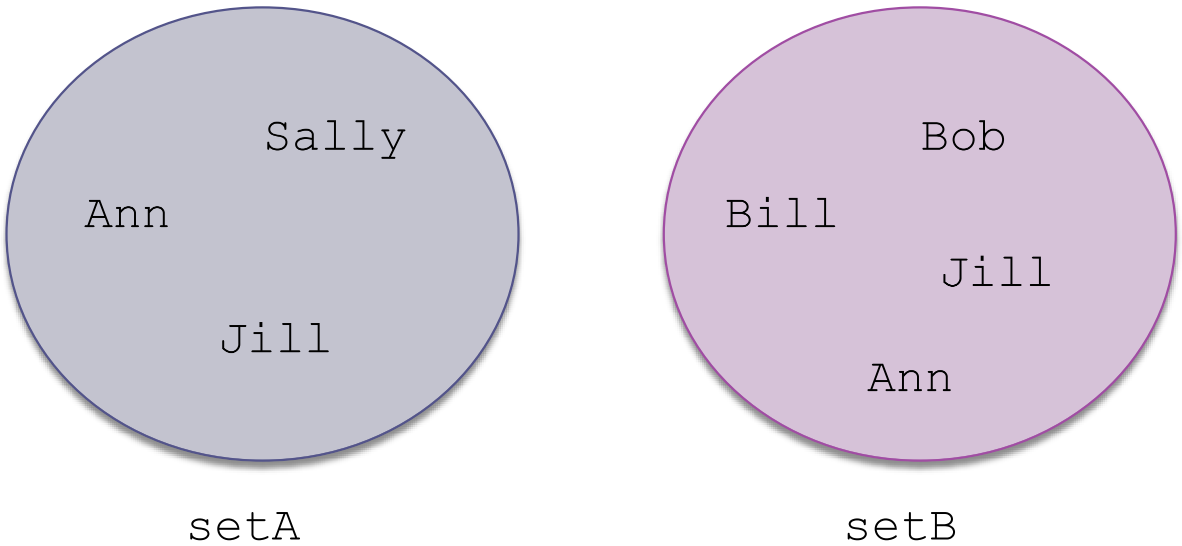
The `Set` Interface and Methods(cont.)

- Required method: `containsAll` tests the *subset* relationship
- Optional methods: `addAll`, `retainAll`, and `removeAll` perform *union*, *intersection*, and *difference* operations, respectively

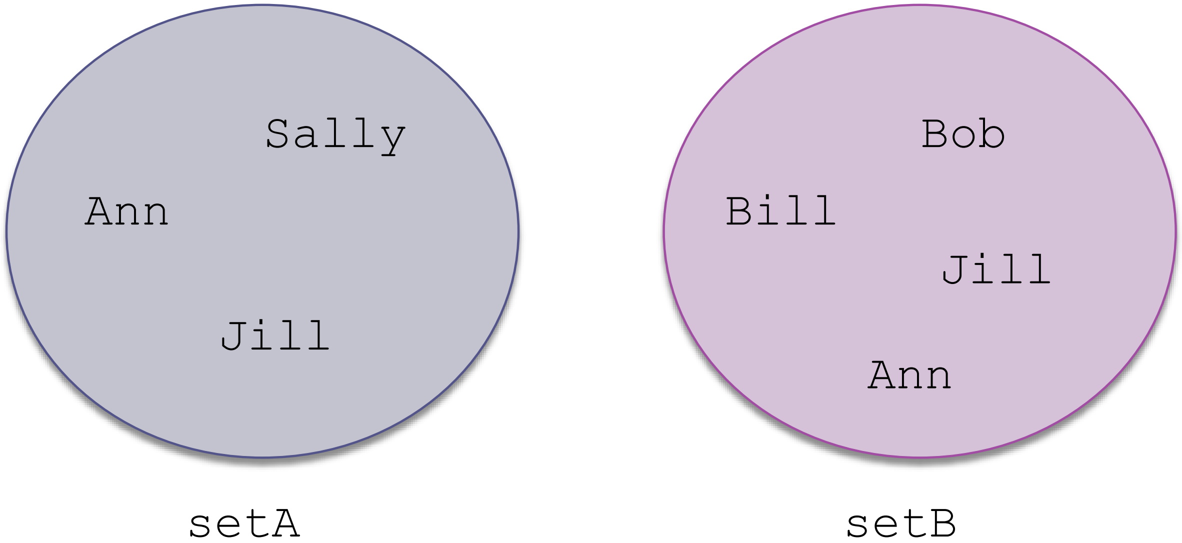
The `Set` Interface and Methods(cont.)

Method	Behavior
`boolean add(E obj)`	Adds item `obj` to this set if it is not already present (optional operation) and returns **`true`**. Returns false if `obj` is already in the set.
`boolean addAll(Collection<E> coll)`	Adds all of the elements in collection `coll` to this set if they're not already present (optional operation). Returns **`true`** if the set is changed. Implements *set union* if `coll` is a `Set`.
`boolean contains(Object obj)`	Returns **`true`** if this set contains an element that is equal to `obj`. Implements a test for *set membership*.
`boolean containsAll(Collection<E> coll)`	Returns **`true`** if this set contains all of the elements of collection `coll`. If `coll` is a set, returns **`true`** if this set is a subset of `coll`.
`boolean isEmpty()`	Returns **`true`** if this set contains no elements.
`Iterator<E> iterator()`	Returns an iterator over the elements in this set.
`boolean remove(Object obj)`	Removes the set element equal to `obj` if it is present (optional operation). Returns **`true`** if the object was removed.
`boolean removeAll(Collection<E> coll)`	Removes from this set all of its elements that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set difference* operation.
`boolean retainAll(Collection<E> coll)`	Retains only the elements in this set that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set intersection* operation.
`int size()`	Returns the number of elements in this set (its cardinality).

The `Set` Interface and Methods(cont.)

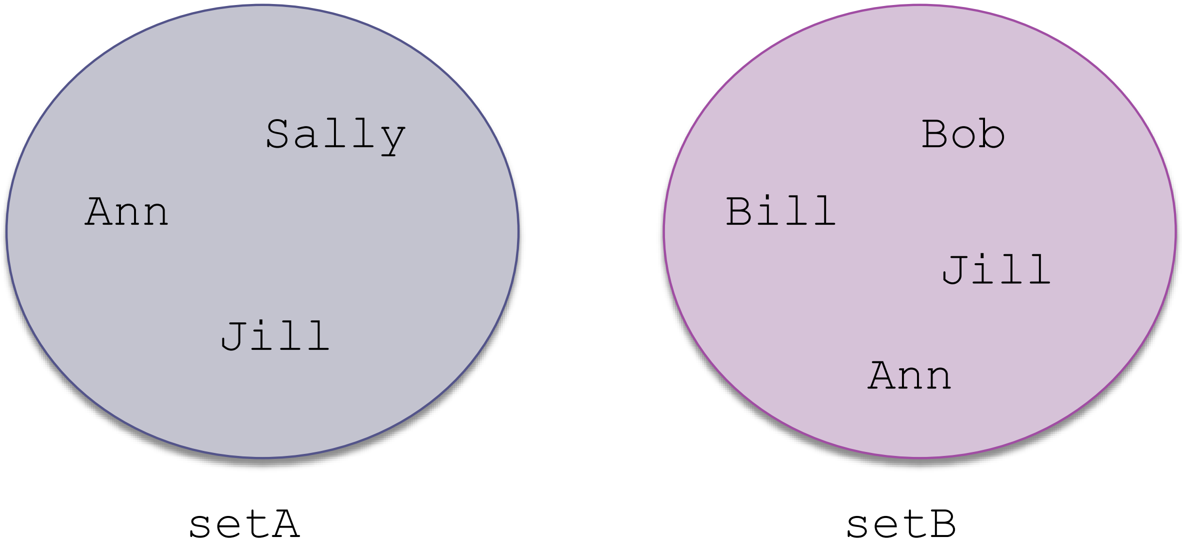


The `Set` Interface and Methods(cont.)




```
setA.addAll(setB);
```

The `Set` Interface and Methods(cont.)



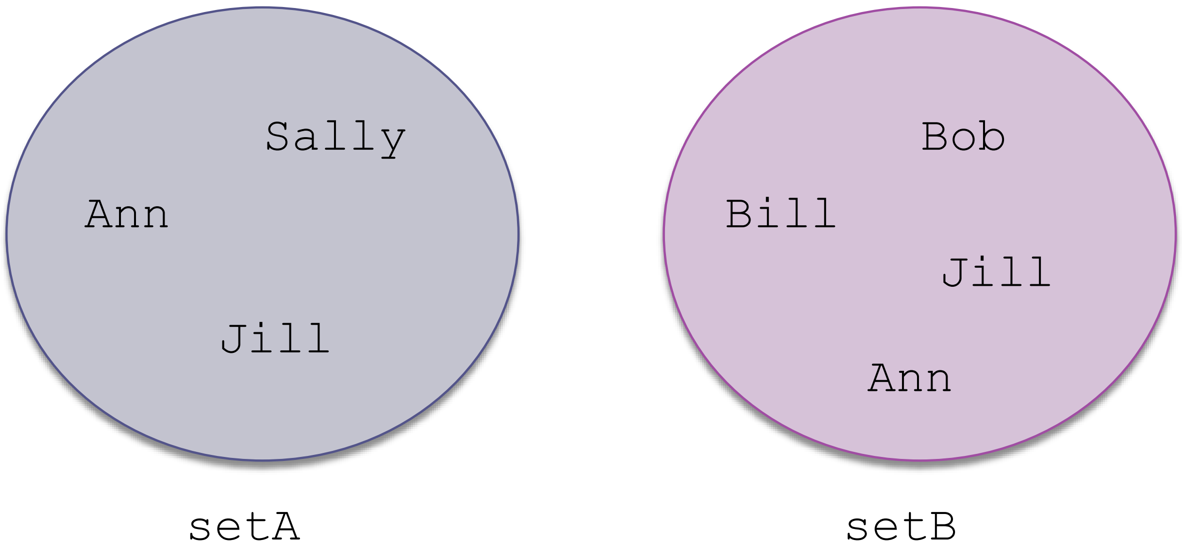

```
setA.addAll(setB);

System.out.println(setA);

Output:
[Bill, Jill, Ann, Sally, Bob]
```

NB: These are names, not people!

The `Set` Interface and Methods(cont.)



If a copy of original `setA` is in `setACopy`, then . . .

The `Set` Interface and Methods(cont.)

- Listing 7.1 (Illustrating the Use of Sets; pages 365-366)

Comparison of Lists and Sets

- Collections implementing the `Set` interface contain only unique elements
- Unlike the `List.add` method, the `Set.add` method returns `false` when inserting an item that would be a duplicate of an item already present
- Unlike a `List`, a `Set` does not have a `get` method—elements cannot be accessed by index

Comparison of Lists and Sets (cont.)

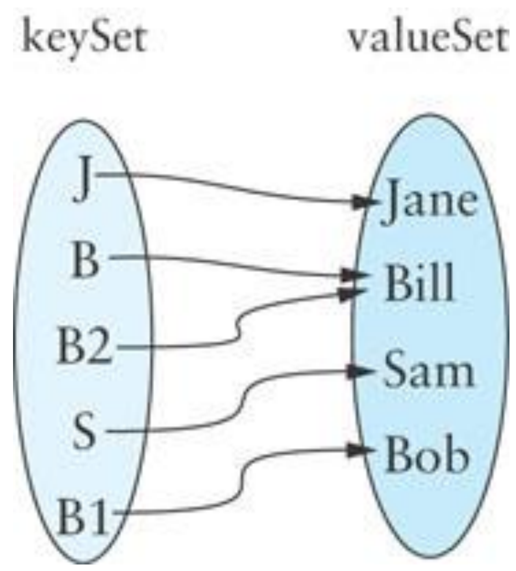
- One can iterate through all elements in a `Set` using an `Iterator` object, but the elements will be accessed in what appears to be an arbitrary order

```
for (String nextItem : setA) {
  //Do something with nextItem
  …
}
```

Maps and the `Map` Interface

Maps and the `Map` Interface

- The `Map` is related to the `Set`
- A `Map` is a set of ordered pairs (key, value)
- Keys are unique, but values need not be unique (remember arrays?)
- Each key is a "mapping" to a particular value
- A map provides efficient storage and retrieval of information in a structure
- A map can have *many-to-one* mapping: (`B, Bill`), (`B2, Bill`)



`{(J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill)}`

Maps and the `Map` Interface(cont.)

- In an *onto* mapping, each element of valueSet has a corresponding member in keySet
- The `Map` interface has methods of the form

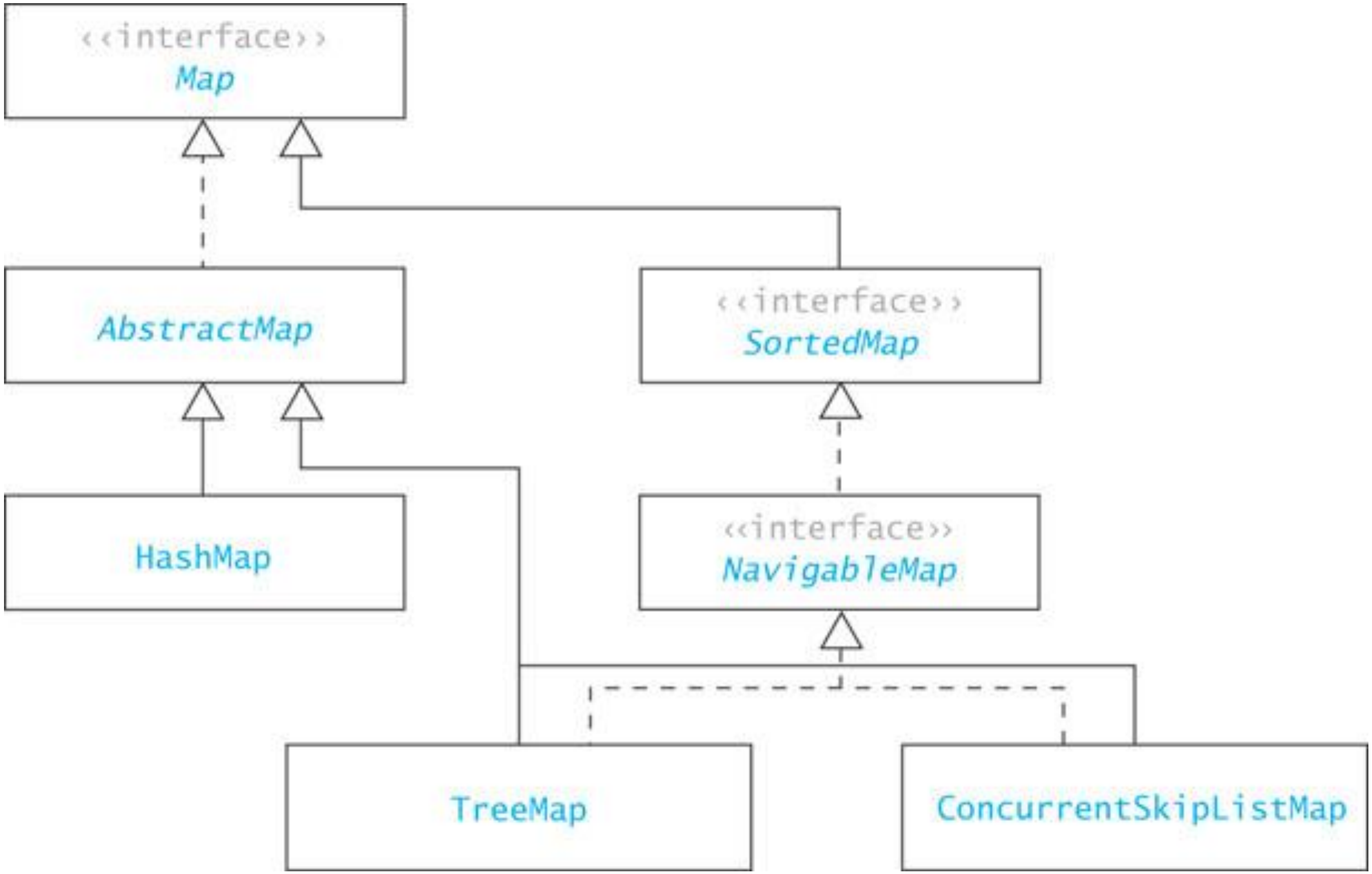
```
V.get (Object key)
V.put (K key, V value)
```

Maps and the `Map` Interface(cont.)

- When information about an item is stored in a table, the information should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

Map **Hierarchy**



Map Interface

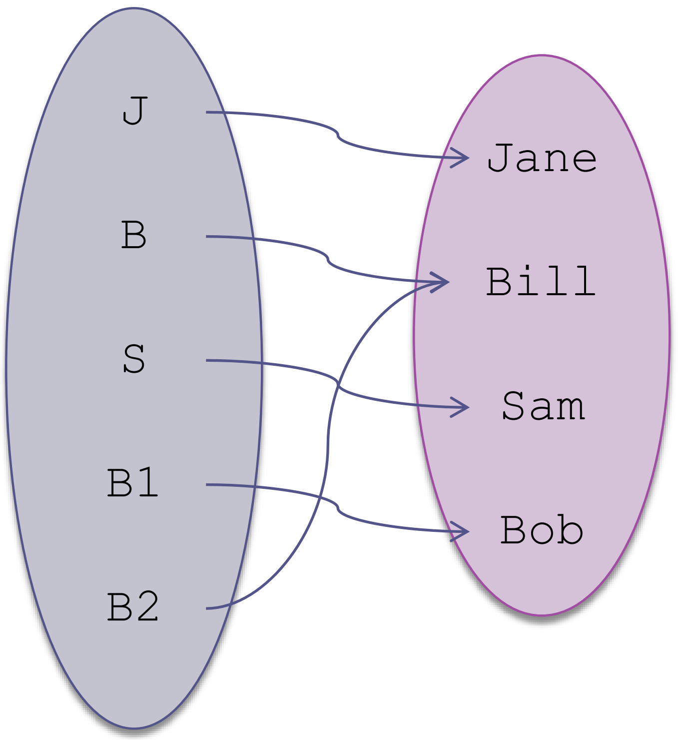
Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of `key-value` mappings in this map.

`Map` Interface (cont.)

- The following statements build a `Map` object:

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

```
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

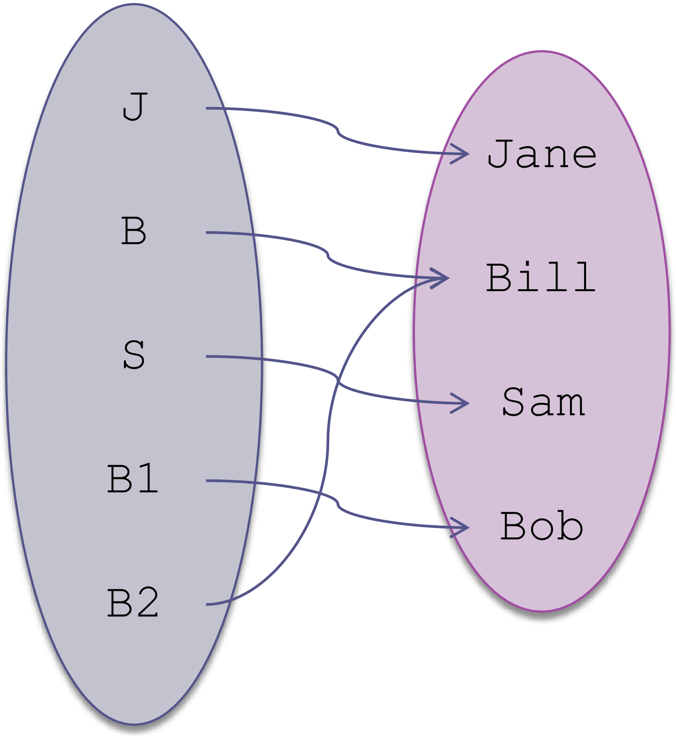


`Map` Interface (cont.)

`aMap.get("B1")`

returns:

`"Bob"`



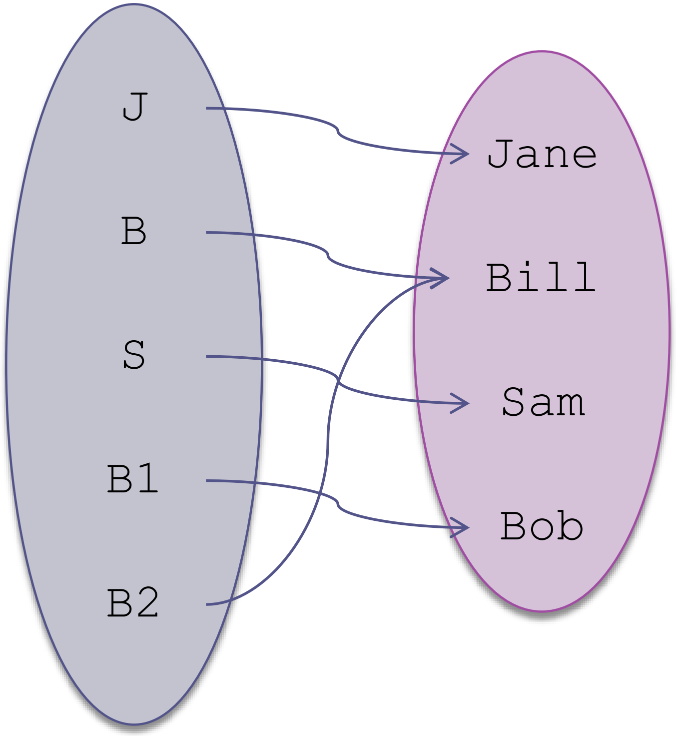
`Map` **Interface** (cont.)

```
aMap.get("Bill")
```

Returns

`null`

(no such key!)



Hash Tables

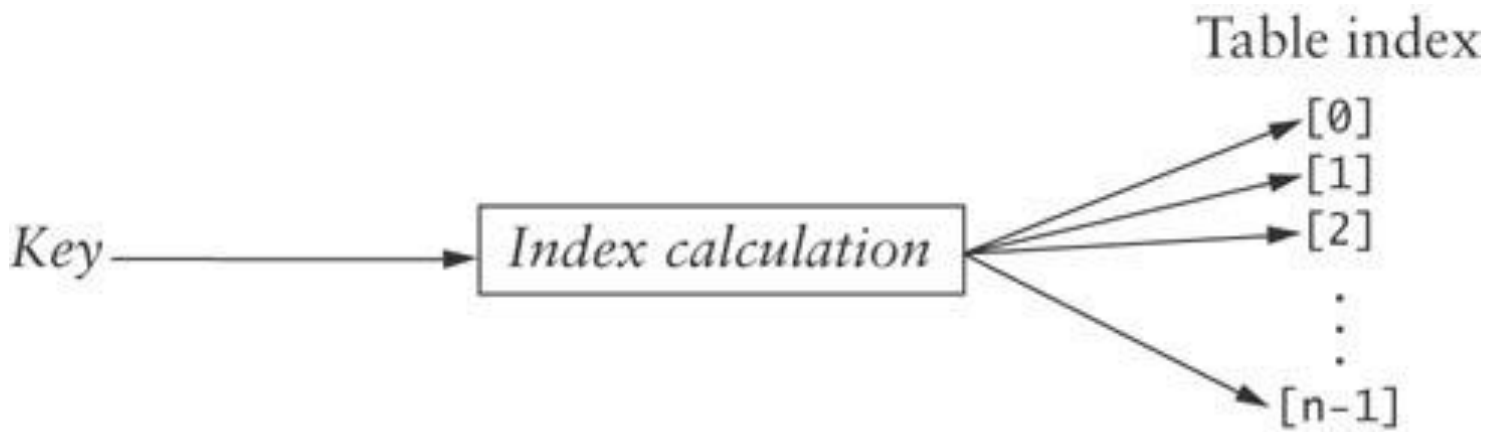
Now something real!..

Hash Tables

- Suppose we want to design a language in which an array could be indexed by… a string: StudentArray[JohnSmith]
- (Such a language exists: *awk* developed in 1970 by Alfred Aho, Peter Weinberger, and Brian Kernighan.*)*
- We could convert each string to an integer that corresponds to its bit string and thus have O(1) access time
- But to support just 9-letter names, an array would need roughly 542950370000 entries, most of which would be empty!
- We have only 42 people in this class, and many names require more than 9 letters…

Hash Codes and Index Calculation

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which then becomes an array index



Hash Codes and Index Calculation (cont.)

- Consider the Huffman code problem from the before
- If a text contains only ASCII characters, which are assigned the first 128 Unicode values, we could use a table of size 128 and let its Unicode value determine the location of a character in the table

Hash Codes and Index Calculation (cont.)

- But what if all 65,536 Unicode characters were allowed?
- If we assume that on average 100 characters were used, we could use a table of 200 characters and compute the index as follows:

```
int index = unicodeValue % 200
```

Hash Codes and Index Calculation (cont.)

- If a text contains the word `mañana`
- Given the following Unicode values:

Hexadecimal	Decimal	Name	Character
0x0029	41	right parenthesis	)
0x00F1	241	small letter n with tilde	ñ

- The indices for letters '`ñ`' and '`)`' are both 41

  `41 % 200 = 41` and `241 % 200 = 41`

- This is called a *collision*; we will discuss how to deal with collisions shortly