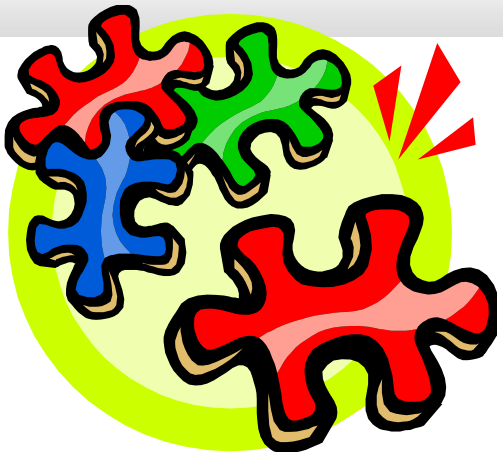
Sets and Maps



Motivations

The "**No-Fly**" **list** is a list, created and maintained by the U.S. government's Terrorist Screening Center, of people who are not permitted to board a commercial aircraft for travel in or out of the United States. Suppose we need to write a program that checks whether a person is on the No-Fly list. You can use a list to store names in the No-Fly list. However, a more efficient data structure for this application is a **set**.

Suppose your program also needs to store detailed information about terrorists in the No-Fly list. The detailed information such as gender, height, weight, and nationality can be retrieved using the name as the key. A **map** is an efficient data structure for such a task.

Set abstract data type

- A ***set*** is a collection of distinct elements. A set ***add*** operation adds an element to the set, provided an equal element doesn't already exist in the set. A set is an unordered collection. Ex: The set with integers 3, 7, and 9 is equivalent to the set with integers 9, 3 and 7.
- A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.

1

Which of the following is not a valid set?

- { 78, 32, 46, 57, 82 }
- { 34, 8, 92 }
- { 78, 28, 91, 28, 15 }

Correct

A set does not contain duplicates. { 78, 28, 91, 28, 15 } contains 28 twice and is therefore not a valid set.

- Elements keys
  - Elements may be:
    - Primitive data type (wrapped as an object)
    - Strings
    - Objects (distinguish elements based on an element's **key value**)

Set ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e\text{: } e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e\text{: } e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e\text{: } e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

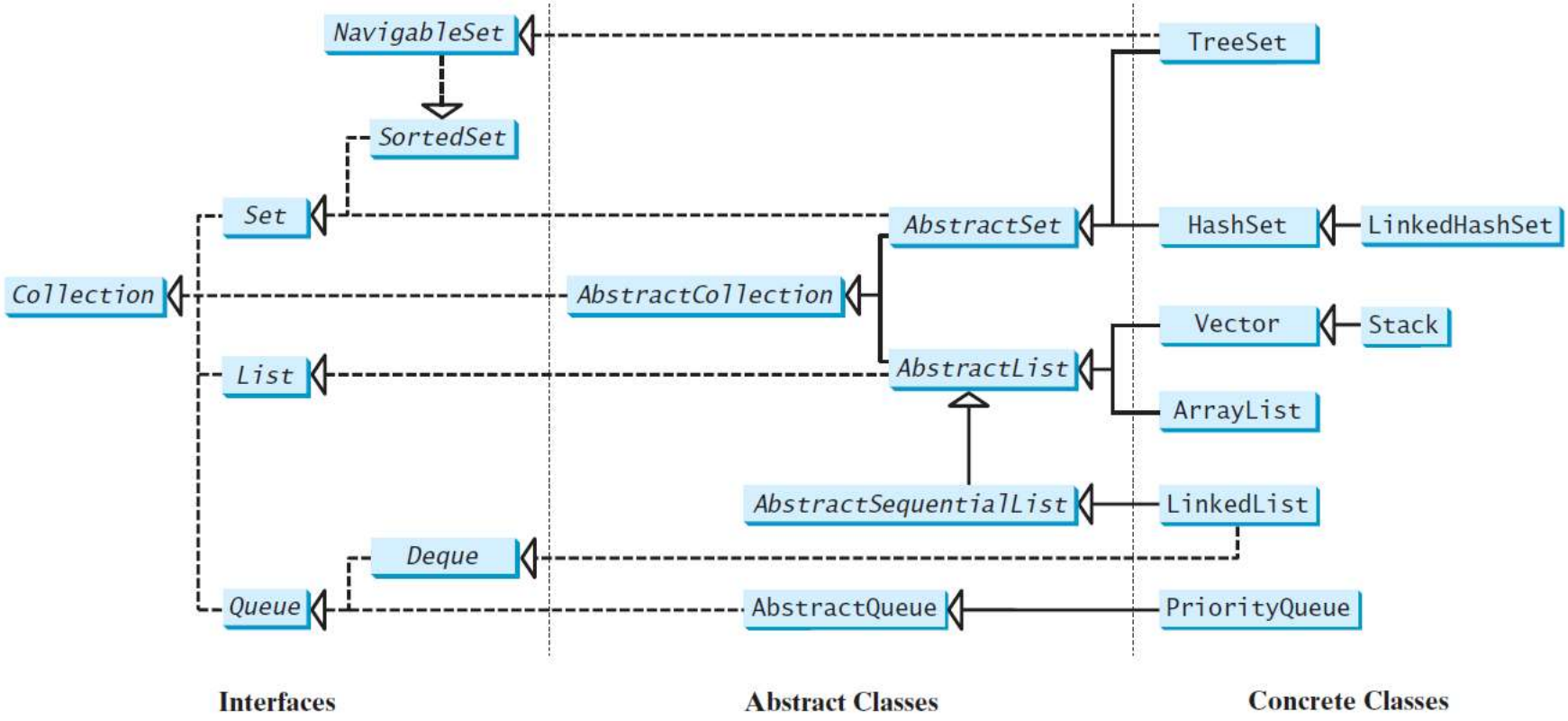
removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Set operations

- The ***union*** of sets X and Y, denoted as X ∪ Y, is a set that contains every element from X, every element from Y, and no additional elements.
  - Ex: { 54, 19, 75 } ∪ { 75, 12 } = { 12, 19, 54, 75 }.
- The ***intersection*** of sets X and Y, denoted as X ∩ Y, is a set that contains every element that is in both X and Y, and no additional elements.
  - Ex: { 54, 19, 75 } ∩ { 75, 12 } = { 75 }.
- The ***difference*** of sets X and Y, denoted as X \ Y, is a set that contains every element that is in X but not in Y, and no additional elements.
  - Ex: { 54, 19, 75 } \ { 75, 12 } = { 54, 19 }.
- The union and intersection operations are commutative, so X ∪ Y = Y ∪ X and X ∩ Y = Y ∩ X. The difference operation is not commutative.

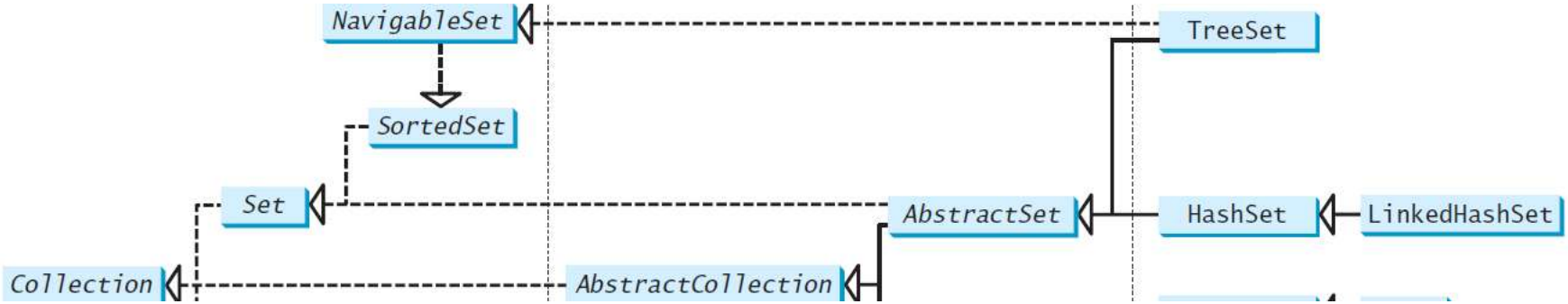
Review of Java Collection Framework hierarchy

Set and List are subinterfaces of Collection.



The HashSet Class

The HashSet class is a concrete class that implements Set. It can be used to store duplicate-free elements. For efficiency, objects added to a hash set need to implement the hashCode method in a manner that properly disperses the hash code.



Example: Using HashSet and Iterator

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

TestHashSet

Example: Using LinkedHashSet

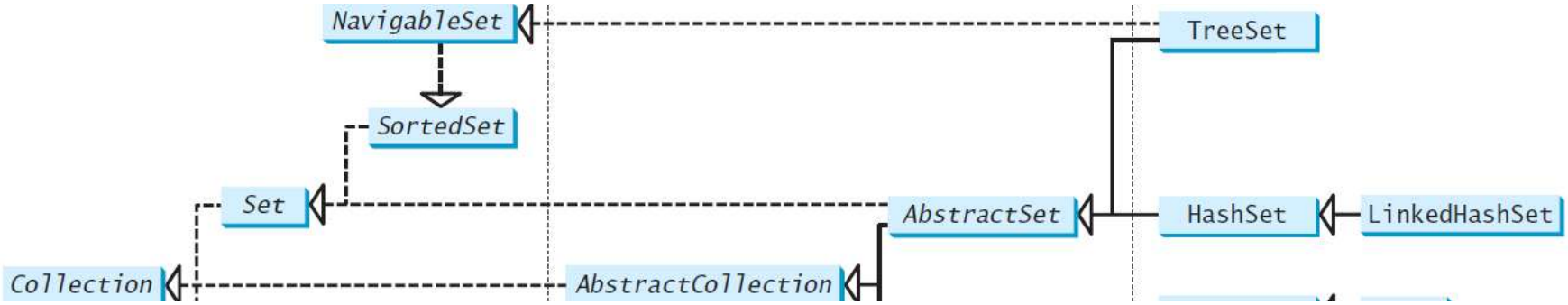
This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

TestLinkedHashSet

The SortedSet Interface and the TreeSet Class

(

SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.



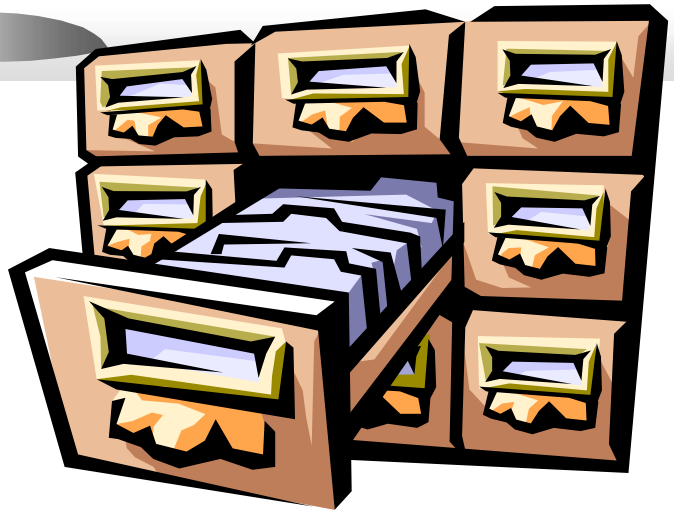
TestTreeSetSet

Case Study: Counting Keywords

This section presents an application that counts the number of the keywords in a Java source file.

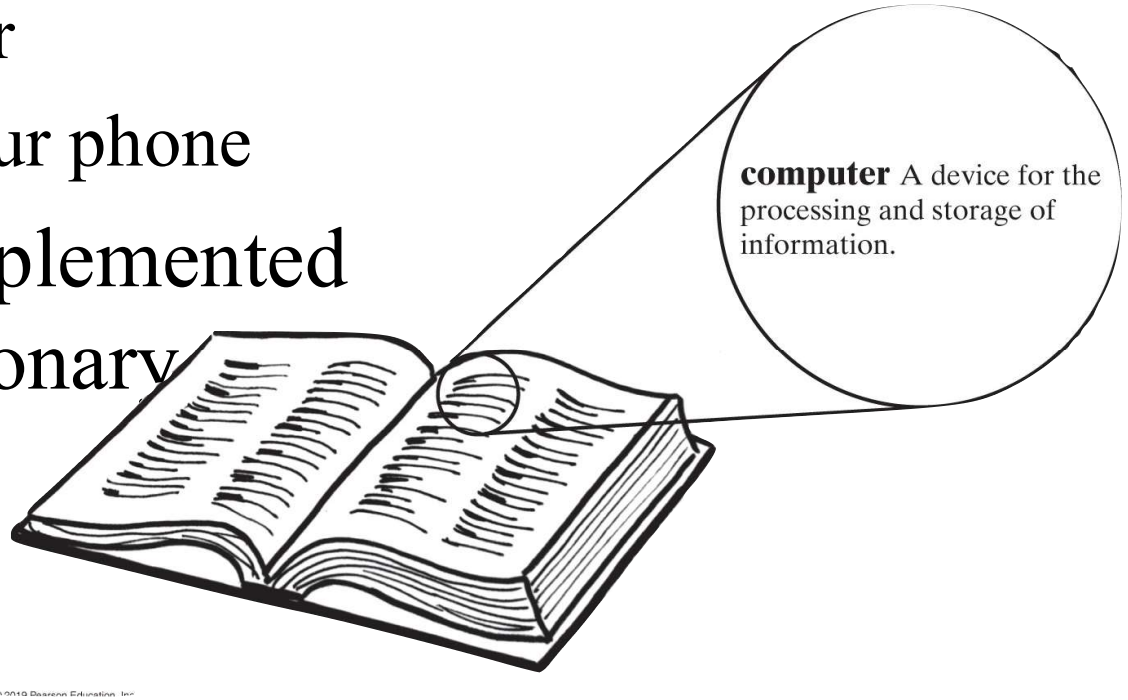
CountKeywords

Maps



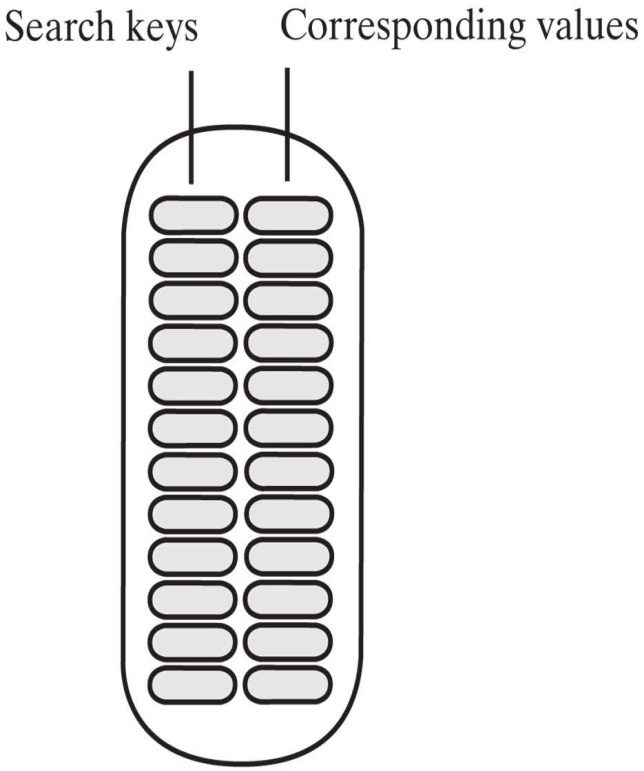
Dictionaries

- When you want to look up …
  - The meaning of a word
  - An address
  - A phone number
  - A contact on your phone
- These can be implemented in an ADT Dictionary



Specifications for the ADT

- Dictionary Data Dictionary
  - *Collection of pairs* `(k, v)` of objects `k` and `v`,
    - `k` is the search key
    - `v` is the corresponding value
  - *Number of pairs* in the collection



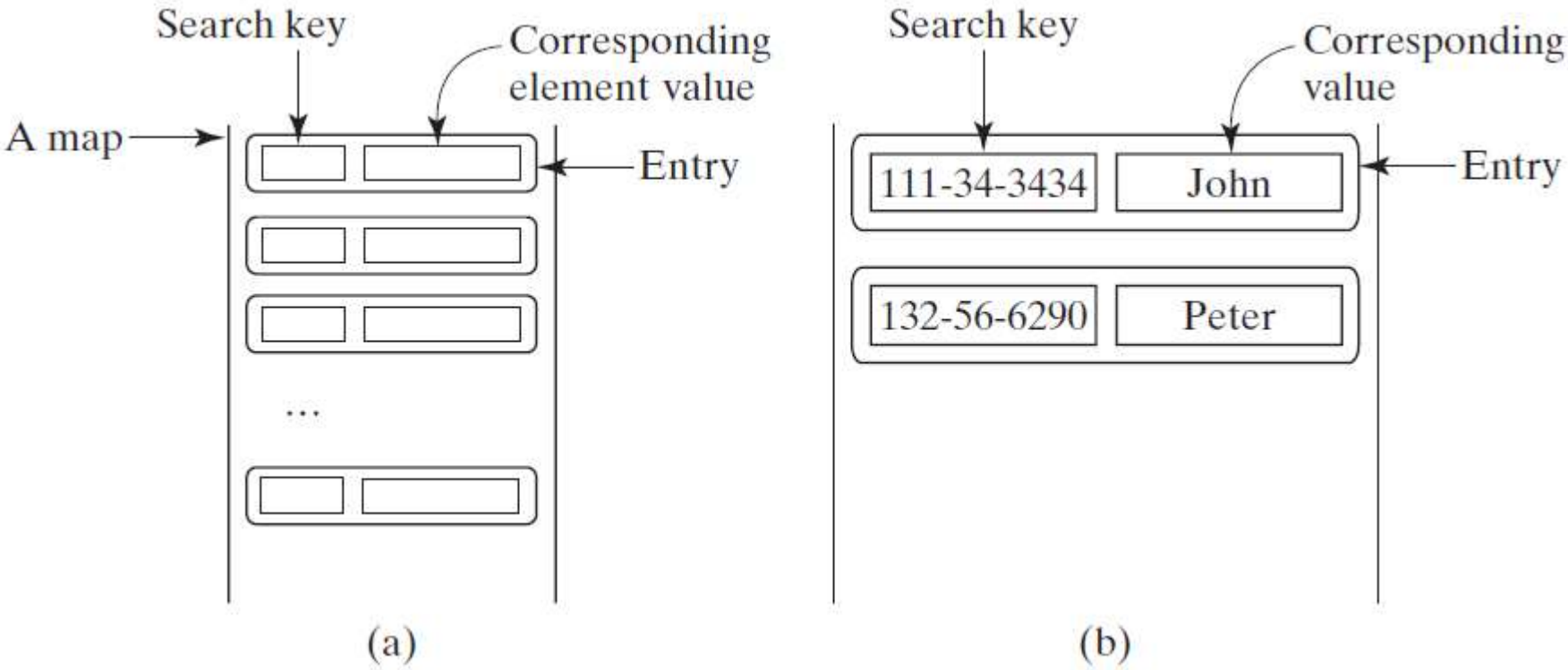
A dictionary object

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

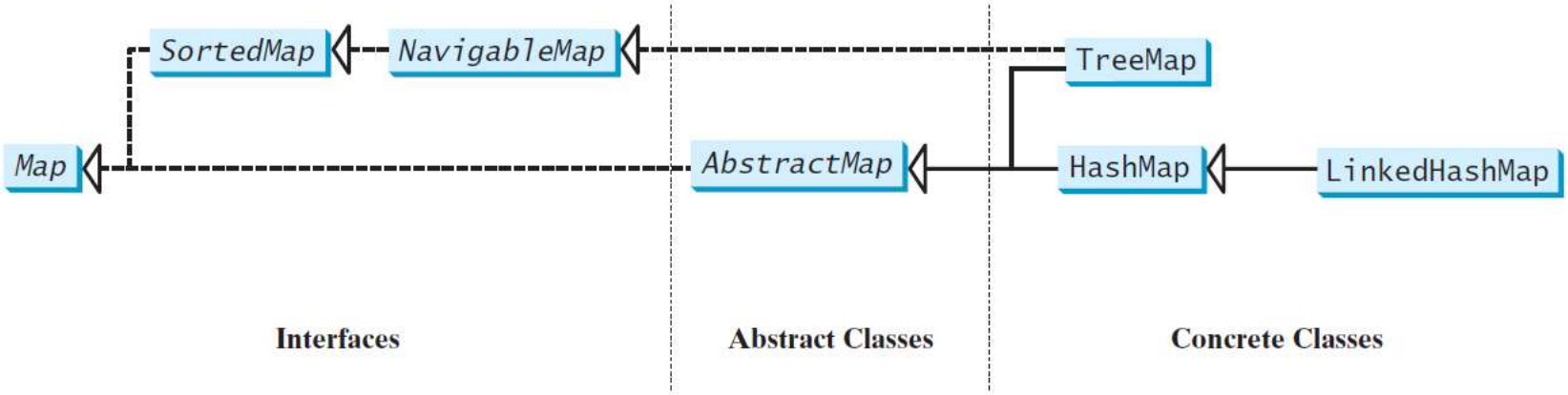
The Map Interface

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.

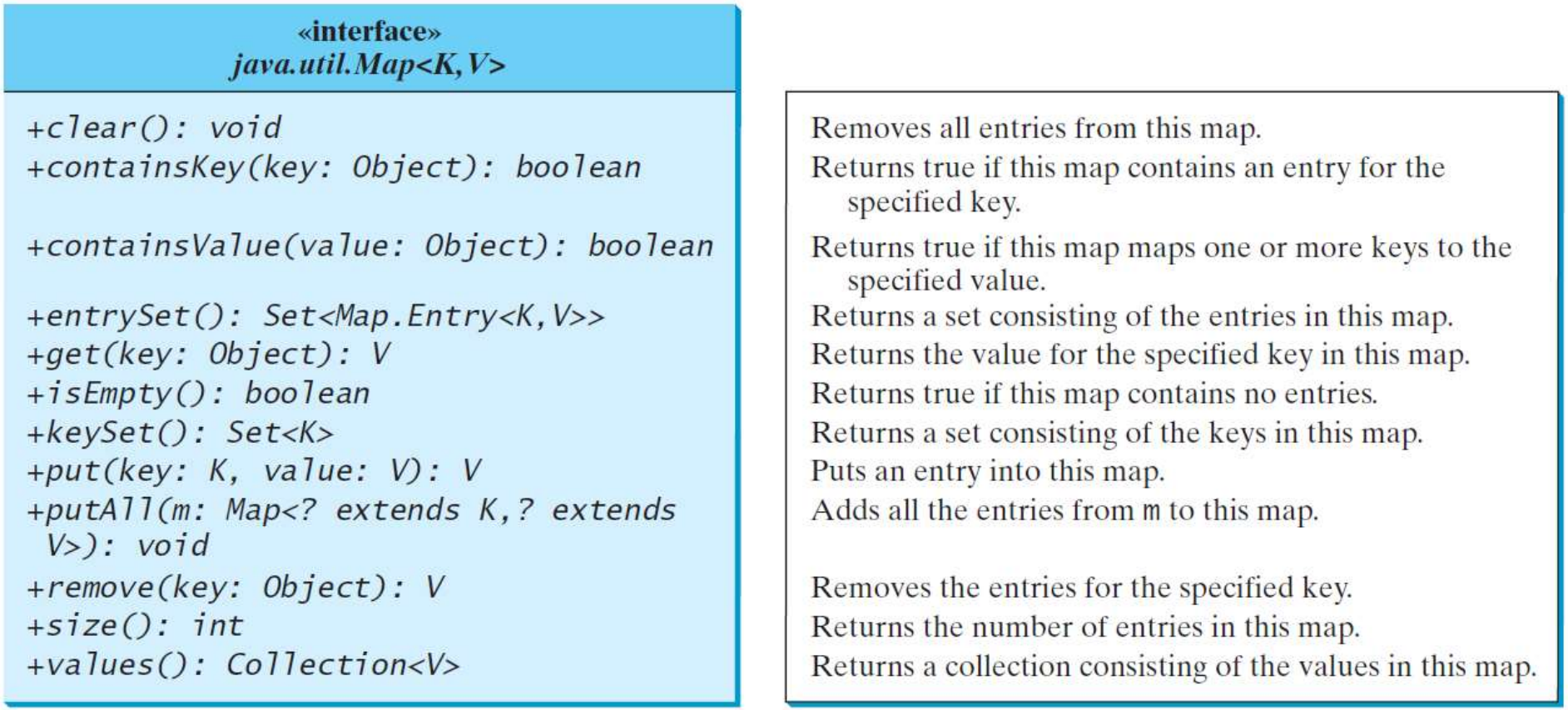


Map Interface and Class Hierarchy

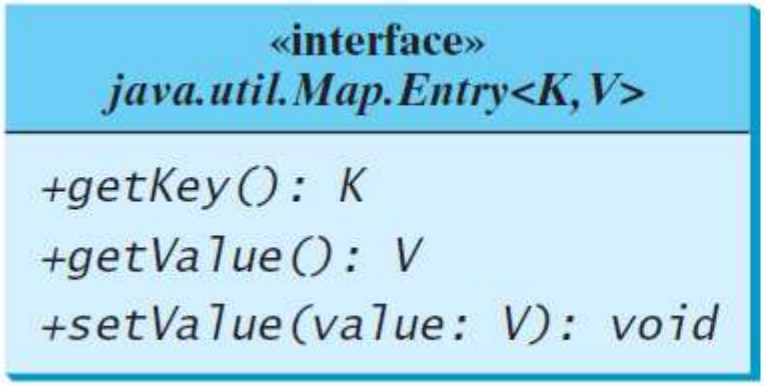
An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.



The Map Interface UML Diagram



Entry



Returns the key from this entry.

Returns the value from this entry.

Replaces the value in this entry with a new value.

HashMap and TreeMap

The HashMap and TreeMap classes are two concrete implementations of the Map interface. The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping. The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

Example: Using HashMap and TreeMap

This example creates a hash map that maps borrowers to mortgages. The program first creates a hash map with the borrower's name as its key and mortgage as its value. The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

TestMap

Case Study: Counting the Occurrences of Words in a Text

This program counts the occurrences of words in a text and displays the words and their occurrences in ascending order of the words. The program uses a hash map to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase the value for the word (key) by 1 in the map. To sort the map, convert it to a tree map.

CountOccurrenceOfWords