Java's Collection Framework

Another use of polymorphism and interfaces

Rick Mercer

Outline

- Java's Collection Framework
  - Unified architecture for representing and manipulating collections
- Collection framework contains
  - Interfaces (ADTs): specification not implementation
  - Concrete implementations as classes
  - Polymorphic Algorithms to search, sort, find, shuffle, ...
- Algorithms are *polymorphic*:
  - the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Collection interfaces in java.util

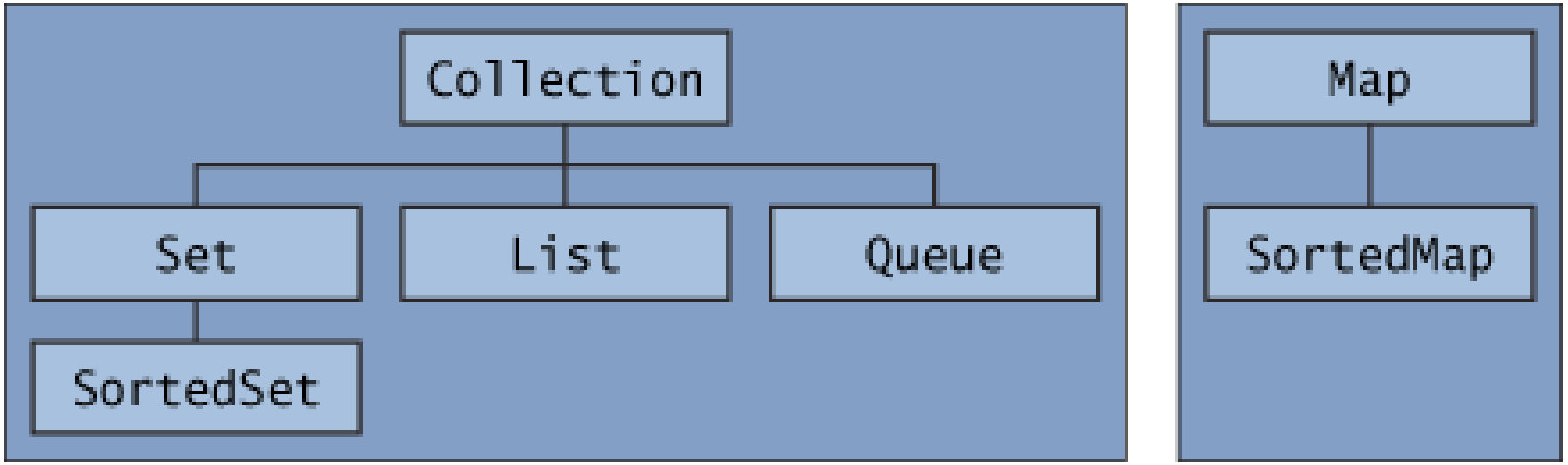

Image from the Java Tutorial

Abstract Data Type

- Abstract data type (ADT) is a specification of the behaviour (methods) of a type
  - Specifies method names to add, remove, find
  - Specifies if elements are unique, indexed, accessible from only one location, mapped,...
  - An ADT shows no implementation
    - no structure to store elements, no implemented algorithms
- What Java construct nicely specifies ADTs?

Collection Classes

- A collection class the can be instantiated
  - implements an interface as a Java class
  - implements all methods of the interface
  - selects appropriate instance variables
- Since Java 5: we have concrete collection classes
  - **`Stack<E>`**
  - **`ArrayList<E>`**, **`LinkedList<E>`**
  - **`LinkedBlockingQueue<E>`**, **`ArrayBlockingQueue<E>`**
  - **`HashSet<E>`**, **`TreeSet<E>`**
  - **`TreeMap<K,V>`**, **`HashMap<K,V>`**

Common Functionality

- Collection classes often have methods for
  - Adding objects
  - Removing an object
  - Finding a reference to a particular object *find*
    - can then send messages to the object still in the collection

List<E>, an ADT written as a Java interface review for some

- List<E>: a collection with a first element, a last element, distinct predecessors and successors
  - The user of this interface has precise control over where in the list each element is inserted
  - duplicates that "**`equals`**" each other are allowed
- The List interface is implemented by these three collection classes
  - ArrayList<E>
  - LinkedList<E>
  - Vector<E>

```
import java.util.*;  // For List, ArrayList, Linked ...
import static org.junit.Assert.*;
import org.junit.Test;

public class ThreeClassesImplementList {

  @Test
  public void showThreeImplementationsOfList() {
    // Interface name: List
    // Three classes that implement the List interface:
    List<String> bigList = new ArrayList<String>();
    List<String> littleList = new LinkedList<String>();
    List<String> sharedList = new Vector<String>();

    // All three have an add method
    bigList.add("in array list");
    littleList.add("in linked list");
    sharedList.add("in vector");

    // All three have a get method
    assertEquals("in array list", bigList.get(0));
    assertEquals("in linked list", littleList.get(0));
    assertEquals("in vector", sharedList.get(0));
  }
}
```

Iterators

- Iterators provide a general way to traverse all elements in a collection

```
ArrayList<String> list = new ArrayList<String>();
list.add("1-FiRsT");
list.add("2-SeCoND");
list.add("3-ThIrD");

Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
  System.out.println(itr.next().toLowerCase());
}
```

Output

```
1-first
2-second
3-third
```

New way to visit elements: Java's Enhanced for Loop

- The for loop has been enhanced to iterate over collections
- General form

  for (*Type element* **:** *collection***) {**

  element is the next thing visited each iteration

  }

```
for (String str : list) {
  System.out.println(str + " ");
}
```

Can't add the wrong type

- Java 5 generics checks the type at compile time
  - See errors early--a good thing
  - "type safe" because you can't add different types

```
ArrayList<GregorianCalendar> dates =
            new ArrayList<GregorianCalendar>();

dates.add(new GregorianCalendar()); // Okay
dates.add("String not a GregorianCalendar"); // Error

ArrayList<Integer> ints = new ArrayList<Integer>();
ints.add(1);  // Okay. Same as add(new Integer(1))
ints.add("Pat not an int")); // Error
```

Algorithms

- Java has *polymorphic* algorithms to provide functionality for different types of collections
  - Sorting (e.g. sort)
  - Shuffling (e.g. shuffle)
  - Routine Data Manipulation (e.g. reverse, addAll)
  - Searching (e.g. binarySearch)
  - Composition (e.g. frequency)
  - Finding Extreme Values (e.g. max)
- Static methods in the Collections class
- Demo a few of these with ArrayList

TreeSet implements Set

- Set<E> An **interface** for collections with no duplicates. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2)
- TreeSet<E>: This **class** implements the Set interface, backed by a balanced binary search tree. This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements as defined by Comparable<T>

Set and SortedSet

- The Set<E> interface
  - add, addAll, remove, size, but no get!
- Two classes that implement Set<E>
  - TreeSet: values stored in order, O(log n)
  - HashSet: values in a hash table, no order, O(1)
- SortedSet extends Set by adding methods E **first**(), SortedSet<E> **tailSet**(E fromElement), SortedSet<E> **headSet**(E fromElement), E **last**(), SortedSet<E> **subSet**(E fromElement, E toElement)

TreeSet elements are in order

```
Set<String> names = new TreeSet<String>();
names.add("Sandeep");
names.add("Chris");
names.add("Kim");
names.add("Chris"); // not added
names.add("Devon");

for (String name : names)
  System.out.println(name);
```

- Output?
- Change to HashSet

The Map Interface (ADT)

- Map describes a type that stores a collection of elements that consists of a *key* and a *value*
- A Map associates (maps) a key the it's value
- The keys must be unique
  - the values need not be unique
  - **`put`** destroys one with same key

Map Operations

- Java's HashMap<K, V>
  - **`public V put(K key, V value)`**
    - associates key to value and stores mapping
  - **`public V get(Object key)`**
    - associates the value to which key is mapped or null
  - **`public boolean containsKey(Object key)`**
    - returns true if the Map already uses the key
  - **`public V remove(Object key)`**
    - Returns previous value associated with specified key, or null if there was no mapping for key.
  - **`Collection<V> values()`**
    - get a collection you can iterate over

Code Demo

Rick: Put in a file named HashMapDemo.java

- Add some mappings to a HashMap and iterate over all elements with **`Collection<V> values()`** and all keys with **`Set<K> keySet()`**

Queue<E>

`boolean add(E e)` Inserts e into this queue

`E element()` Retrieves, but does not remove, the head of this queue

`boolean offer(E e)` Inserts e into this queue

`E peek()` Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty

`E poll()` Retrieves and removes the head of this queue, or returns null if this queue is empty

`E remove()` Retrieves and removes the head of this queue

ArrayBlockingQueue<E> a FIFO queue

```
ArrayBlockingQueue<Double> numberQ =
          new ArrayBlockingQueue<Double>(40);
numberQ.add(3.3);
numberQ.add(2.2);
numberQ.add(5.5);
numberQ.add(4.4);
numberQ.add(7.7);

assertEquals(3.3, numberQ.peek(), 0.1);
assertEquals(3.3, numberQ.remove(), 0.1);
assertEquals(2.2, numberQ.remove(), 0.1);
assertEquals(5.5, numberQ.peek(), 0.1);
assertEquals(3, numberQ.size());
```