Principles of Software Construction: Objects, Design and Concurrency

Java Collections

15-214

toad

Fall 2013

Jonathan Aldrich **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Midterm exam in class next Tuesday, 15 Oct
- Midterm review session Sunday 1 – 3 p.m., Hamburg Hall 1000
  - Review sheet will be released before the review session
  - Will also release a sample midterm exam soon

Key concepts from Tuesday

Key concepts from Tuesday

- GUIs
  - The Model-View-Controller pattern
  - The Observer pattern
- Java Swing architecture
  - Threading architecture
  - Swing components
  - Using the Observer pattern in Swing

Design patterns we have seen so far:

Composite

Template Method

Strategy

Observer

Decorator

Model-View-Controller

Key concepts for today

- A tour of the Java Collections Framework
  - Some of the features
  - Some of the common usage patterns
  - Some of the design patterns in use
    - *Iterator*, *Marker Interface*, *Factory Method*, *Adapter*, Strategy, Decorator, Template Method

The philosophy of the Collections framework

- Powerful and general
- Small in size and conceptual weight
  - Only include fundamental operations
  - "Fun and easy to learn and use"

The java.util.Collection<E> interface

```
boolean      add(E e);
boolean      addAll(Collection<E> c);
boolean      remove(E e);
boolean      removeAll(Collection<E> c);
boolean      retainAll(Collection<E> c);
boolean      contains(E e);
boolean      containsAll(Collection<E> c);
void         clear();
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

The java.util.List<E> interface

- Defines order of a collection
- Extends java.util.Collection<E>:

```
boolean add(int index, E e);
E       get(int index);
E       set(int index, E e);
int     indexOf(E e);
int     lastIndexOf(E e);
List<E> sublist(int fromIndex, int toIndex);
```

The java.util.Set<E> interface

- Enforces uniqueness of each element in collection
- Extends java.util.Collection<E>:

- Aside: The *Marker Interface* pattern
  - Problem: You want to define a behavioral constraint not enforced at compile time.
  - Solution: Define an interface with no methods.

The java.util.Queue<E> interface

- Extends java.util.Collection<E>:

```
boolean add(E e);     // These three methods
E       remove();     // might throw exceptions
E       element();

boolean offer(E e);
E       poll();       // These two methods
E       peek();       // might return null
```

The java.util.Map<K,V> interface

- Does not extend java.util.Collection<E>

```
V        put(K key, V value);
V        get(Object key);
V        remove(Object key);
boolean  containsKey(Object key);
boolean  containsValue(Object value);
void     putAll(Map<K,V> m);
int      size();
boolean  isEmpty();
void     clear();
Set<K>                  keySet();
Collection<V>           values();
Set<Map.Entry<K,V>>     entrySet();
```

One problem: Java arrays are not Collections

- To convert a Collection to an array
  - Use the toArray method

```
List<String> arguments = new LinkedList<String>();
…                    // puts something into the list
String[] arr = (String[]) arguments.toArray();
String[] brr = arguments.toArray(new String[0]);
```

- To view an array as a Collection
  - Use the java.util.Arrays.asList method

```
String[] arr = {"foo", "bar", "baz", "qux"};
List<String> arguments = Arrays.asList(arr);
```

One problem: Java arrays are not Collections

- To convert a Collection to an array
  - Use the toArray method

```
List<String> arguments = new LinkedList<String>();
…                      // puts something into the list
String[] arr = (String[]) arguments.toArray();
String[] brr = arguments.toArray(new String[0]);
```

- To view an array as a Collection
  - Use the java.util.Arrays.asList method

```
String[] arr = {"foo", "bar", "baz", "qux"};
List<String> arguments = Arrays.asList(arr);
```

- Aside: The *Adapter* pattern
  - Problem: Existing library or class does not match the interface you want
  - Solution: Expose the functionality of the object in a different form

What do you want to do with your Collection today?

Traversing a Collection

- Old-school Java for loop for ordered types

```
List<String> arguments = …;
for (int i = 0; i < arguments.size(); ++i) {
  System.out.println(arguments.get(i));
}
```

- Modern standard Java for-each loop

```
List<String> arguments = …;
for (String s : arguments) {
  System.out.println(s);
}
```

- Use an Iterator

The *Iterator* pattern

```
public interface java.util.Iterator<E> {
  boolean hasNext();
  E next();
  void remove();  // removes previous returned item
}                 // from the underlying collection
```

- To use, e.g.:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  System.out.println(s);
}
```

Using a java.util.Iterator<E>

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();  // removes previous returned item
}                 // from the underlying collection
```

- To use to remove items, e.g.:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Charlie"))
    it.remove();
}
// The next line will always print false
System.out.println(arguments.contains("Charlie"));
```

Using a java.util.Iterator<E>: A warning

- The default Collections implementations are mutable
- The java.util.Iterator assumes the Collection does not change while the Iterator is being used
  - You will get a `ConcurrentModificationException`

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Charlie"))
    arguments.remove("Charlie"); // runtime error
}
```

Aside: The *Factory Method* pattern

```
public interface Collection<E> {
  boolean     add(E e);
  boolean     addAll(Collection<E> c);
  boolean     remove(E e);
  boolean     removeAll(Collection<E> c);
  boolean     retainAll(Collection<E> c);
  boolean     contains(E e);
  boolean     containsAll(Collection<E> c);
  void        clear();
  int         size();
  boolean     isEmpty();
  Iterator<E> iterator();
  Object[]    toArray()
  E[]         toArray(E[] a);
  …
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.

Sorting a Collection

- Use the Collections.sort method:

```
public static void main(String[] args) {
  List<String> lst = Arrays.asList(args);
  Collections.sort(lst);
  for (String s : lst) {
    System.out.println(s);
  }
}
```

- Abuse the SortedSet:

```
public static void main(String[] args) {
  SortedSet<String> set =
        new TreeSet<String>(Arrays.asList(args));
  for (String s : set) {
    System.out.println(s);
  }
}
```

Sorting your own types of objects

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- General contracts:
  - `a.compareTo(b)` should return:
    - `<0` if `a` is less than `b`
    - `0` if `a` and `b` are equal
    - `>0` if `a` is greater than `b`
  - Should define a total order
    - If `a.compareTo(b)` < 0 and `b.compareTo(c)` < 0, then `a.compareTo(c)` should be < 0
    - If `a.compareTo(b)` < 0, then `b.compareTo(a)` should be > 0
  - Should usually be consistent with `.equals`
    - `a.compareTo(b)` == 0 iff `a.equals(b)`

Comparable objects – an example

```
public class Integer implements Comparable<Integer> {
  private int val;
  public Integer(int val) { this.val = val; }
  …
  public int compareTo(Integer o) {
    if (val < o.val) return -1;
    if (val == o.val) return 0;
    return 1;
  }
}
```

- Aside: Why did I not just `return val – o.val`?

Comparable objects – another example

- Make Name comparable:

```
public class Name                                          {
  private String first;
  private String last;
  public Name(String first, String last) {   // should
    this.first = first;  this.last = last;   // check
  }                                          // for null
  …




}
```

- Hint: Strings implement `Comparable<String>`

Comparable objects – another example

- Make Name comparable:

```
public class Name implements Comparable<Name> {
  private String first;
  private String last;
  public Name(String first, String last) {   // should
    this.first = first;  this.last = last;   // check
  }                                          // for null
  …
  public int compareTo(Name o) {
    int lastComparison = last.compareTo(o.last);
    if (lastComparison != 0) return lastComparison;
    return first.compareTo(o.first);
  }
}
```

Alternative comparisons

```
public class Employee implements Comparable<Employee> {
  protected Name name;
  protected int  salary;
  …
}
```

- What if we want to sort Employees by name, usually, but sometimes sort by salary?

Alternative comparisons

```
public class Employee implements Comparable<Employee> {
  protected Name name;
  protected int  salary;
  …
}
```

- What if we want to sort Employees by name, usually, but sometimes sort by salary?
- Answer: There's a Strategy pattern interface for that

```
public interface Comparator<T> {
  public int     compare(T o1, T o2);
  public boolean equals(Object obj);
}
```

Writing a Comparator object

```
public class Employee implements Comparable<Employee> {
  protected Name name;
  protected int  salary;
  public int compareTo(Employee o) {
    return name.compareTo(o.name);
  }
}

public class EmpSalComp implements Comparator<Employee> {
  public int compare (Employee o1, Employee o2) {
    return o1.salary – o2.salary;  // Why is this OK?
  }
  public boolean equals(Object obj) {
    return obj instanceof EmpSalComp;
  }
}
```

Using a Comparator

- Order-dependent classes and methods take a Comparator as an argument

```
public class Main {
  public static void main(String[] args) {
    SortedSet<Employee> empByName =   // sorted by name
          new TreeSet<Employee>();

    SortedSet<Employee> empBySal =  // sorted by salary
          new TreeSet<Employee>(new EmpSalComp());
  }
}
```

Aside: The java.util.SortedSet<E> interface

- Extends java.util.Set<E>:

```
Comparator<E> comparator();
E             first();
E             last();
SortedSet<E>  subSet(E fromElement, E toElement);
SortedSet<E>  headSet(E toElement);
SortedSet<E>  tailSet(E fromElement);
```

- The `comparator` method returns null if the natural ordering is being used

The java.util.Collections class

- Standard implementations of common algorithms
  - binarySearch, copy, fill, frequency, indexOfSubList, min, max, nCopies, replaceAll, reverse, rotate, shuffle, sort, swap, …

```
public class Main() {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList(args);
    Collections.sort(lst);
    for (String s : lst) {
      System.out.println(s);
    }
  }
}
```

The java.util.Collections class

- Standard implementations of common algorithms
  - binarySearch, copy, fill, frequency, indexOfSubList, min, max, nCopies, replaceAll, reverse, rotate, shuffle, sort, swap, …

```
public class Main() {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList(args);
    int x = Collections.frequency(lst, "Charlie");
    System.out.println("There are " + x +
                       " students named Charlie");
  }
}
```

The java.util.Collections class

- Many uses of the Decorator pattern:

```
static List<T>  unmodifiableList(List<T>  lst);
static Set<T>   unmodifiableSet( Set<T>   set);
static Map<K,V> unmodifiableMap( Map<K,V> map);
static List<T>  synchronizedList(List<T>  lst);
static Set<T>   synchronizedSet( Set<T>   set);
static Map<K,V> synchronizedMap( Map<K,V> map);
                        .
                        .
                        .
```

The java.util.Collections class

- An actual method declaration

```
static int binarySearch(
        List<? extends Comparable<? super T>> list,
        T                                     key);
```

An object of some type T to search for

A List of objects of some type that has a compareTo method that can take an object of type T as an argument

Java Collections as a *framework*

- You can write specialty collections
  - Custom representations and algorithms
  - Custom behavioral guarantees
    - e.g., file-based storage
- JDK built-in algorithms would then be calling your collections code

The abstract java.util.AbstractList<T>

```
abstract T   get(int i);                  // Template Method.
abstract int size();                      // Template Method.
boolean      set(int i, E e);             // set add remove are
boolean      add(E e);                    // pseudo-abstract
boolean      remove(E e);                 // Template Methods.
boolean      addAll(Collection<E> c);
boolean      removeAll(Collection<E> c);
boolean      retainAll(Collection<E> c);
boolean      contains(E e);
boolean      containsAll(Collection<E> c);
void         clear();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
…
```

Next time…

- Midterm exam on Tuesday