Principles of Software Construction: Objects, Design and Concurrency

More design patterns and Java Collections

Christian Kästner                Charlie Garrod

© 2012-13  C Garrod, C Kästner, J Aldrich, and W Scherlis
Design patterns from last week

Abstract Factory
Composite
Strategy
Façade
Command
Factory Method
Template Method
Decorator
Adapter
Administrivia

• Midterm in class next Tuesday
  ▪ Review session Sunday 1 – 3 p.m., PH 100
  ▪ Sample midterm and solutions coming soon

• Homework 2 graded...

• Homework 3 grading expected by Saturday night

• Homework 4a due Thursday night
A note on design

• The previous exercise is backward
  • "Here's a design pattern. Now use it..."

• The real exercise: "How do I design this program to accomplish my goals?..."
  • "Aha! I've seen this problem before..."
One more design pattern…

• The scenario:
  • You need a single point of control for some data or operations
    • e.g.:
      • Encapsulate global data you don't want to pass around the application
      • Careful access of some resource

• A solution: Create a single instance of an object to contain the data and methods, but...
  • You must prevent more than one instance from being created
An example of the Singleton pattern

public class DbConnectionManager {
    private static DbConnectionManager manager;

    protected DbConnectionManager() {
        ...; // Construct as usual...
    }

    public static DbConnectionManager getManager() {
        if (manager == null) {
            manager = new DbConnectionManager();
        }
        return manager;
    }

    ...
}
Creational: Singleton

- **Applicability**
  - There must be exactly one instance of a class
  - When it must be accessible to clients from a well-known place
  - When the sole instance should be extensible by subclassing, with unmodified clients using the subclass

- **Consequences**
  - Controlled access to sole instance
  - Reduced name space (vs. global variables)
  - Can be refined in subclass or changed to allow multiple instances
  - More flexible than class operations
    - Can change later if you need to

- **Implementation**
  - Constructor is protected
  - Instance variable is private
  - Public operation returns singleton
    - May lazily create singleton

- **Subclassing**
  - Instance() method can look up subclass to create in environment
Today: The Java Collections Framework

• Interfaces (in `java.util`)
  - Collection
    - List
    - Set
    - Queue
    - SortedSet
  - Map
    - SortedMap

• Default Implementations
  - ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap, TreeMap, LinkedHashSet, LinkedHashMap, ...

• Algorithms
  - min, max, sort, reverse, binarySearch, shuffle, rotate, ...
Today's goals

• A tour of the Java Collections Framework
  ▪ Some of the features
  ▪ Some of the common usage patterns

• At least one new design pattern
  ▪ And see some design patterns in use

• See the big Collections picture
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

```java
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>`:

  ```java
  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 contract 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
    ```java
    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
    ```java
    String[] arr = {"foo", "bar", "baz", "qux");
    List<String> arguments = Arrays.asList(arr);
    ```
What do you want to do with your Collection today?
Traversing a Collection

- **Old-school Java for loop for ordered types**
  
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); ++i) {
      System.out.println(arguments.get(i));
  }
  ```

- **Modern standard Java for-each loop**
  
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
      System.out.println(s);
  }
  ```

- **Use an Iterator**
The *Iterator* pattern

```java
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>

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

• **Use the Collections.sort method:**
  
  ```java
  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:**
  
  ```java
  public static void main(String[] args) {
      SortedSet<String> set =
          new TreeSet<String>(Arrays.asList(args));
      for (String s : lst) {
          System.out.println(s);
      }
  }
  ```
public interface Comparable<T> {
    int compareTo(T o);
}

• General contracts:
  ▪ \(a \text{.compareTo}(b)\) 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 \(a \text{.compareTo}(b) < 0\) and \(b \text{.compareTo}(c) < 0\), then
      \(a \text{.compareTo}(c)\) should be \(< 0\)
    • If \(a \text{.compareTo}(b) < 0\), then \(b \text{.compareTo}(a)\) should
      be \(> 0\)
  ▪ Should usually be consistent with .equals
    • \(a \text{.compareTo}(b) == 0\) iff \(a \text{.equals}(b)\)
Comparable objects – an example

```java
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 \( \text{val} - \text{o.val} \)?
Comparable objects – another example

• Make Name comparable:

```java
public class Name {
    private String first;
    private String last;
    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }
    ...
}
```

• Hint: Strings implement Comparable<String>
Comparable objects – another example

• Make Name comparable:

```java
public class Name implements Comparable<Name> {
    private String first;
    private String last;
    public Name(String first, String last) {
        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

```java
public class Employee implements Comparable<Employee> {
    private Name name;
    private int salary;

    ... 
}
```

• What if we want to sort Employees by name, usually, but sometimes sort by salary?
Alternative comparisons

```java
public class Employee implements Comparable<Employee> {
    private Name name;
    private int salary;
    ...
}

• What if we want to sort Employees by name, usually, but sometimes sort by salary?
• Answer: There's an app\(^H^H^H\)interface for that
```

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

```java
public class Employee implements Comparable<Employee> {
    private Name name;
    private 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

```java
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`, ...

```java
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`, ...

```java
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

- Standard implementations of common algorithms
- An actual method declaration
  
  ```java
  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**
Next time…

- A brief introduction to GUIs
- I/O in Java