

Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Designing (sub-) systems

Java Collections design case study

Michael Hilton

Bogdan Vasilescu

Administrivia

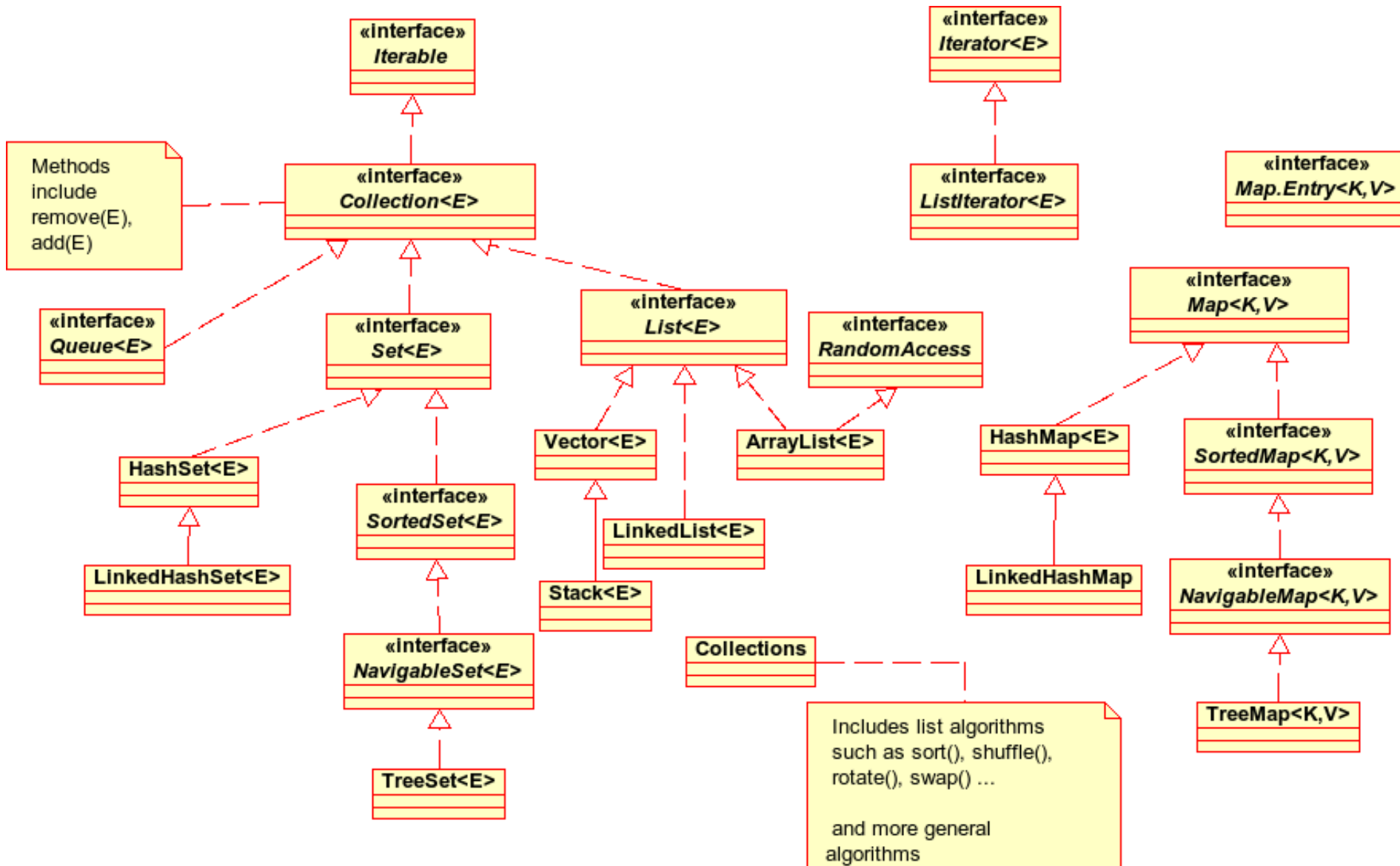
- Homework 4b due next Thursday, March 7th
 - Up to 3 late days are acceptable from your available budget, or 10% per day
- Remaining Homework 4a feedback available today at end of class
 - Can regain 75% of lost Homework 4a credit
 - Directly address TA comments when you turn in Homework 4c
 - Turn in revised design documents + scans of our feedback

Key concepts from Tuesday

Key concepts from Tuesday

- GUIs are filled with design patterns
 - Strategy
 - Template method
 - Observer
 - Composite
 - Decorator
 - Adapter
 - Façade
 - Command
 - Chain of responsibility

Today: Java Collections



Learning goals for today

- Understand the design challenges of collection libraries.
- Recognize the design patterns used and how those design patterns achieve design goals.
 - Marker interface
 - Decorator
 - Factory method
 - Iterator
 - Strategy
 - Template method
 - Adapter

Designing a data structure library

- Different data types: lists, sets, maps, stacks, queues, ...
- Different representations
 - Array-based lists vs. linked lists
 - Hash-based sets vs. tree-based sets
 - ...
- Many alternative design decisions
 - Mutable vs. immutable
 - Sorted vs. unsorted
 - Accepts null or not
 - Accepts duplicates or not
 - Concurrency/thread-safe or not
 - ...

The philosophy of the Collections framework

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

Overview of the Collections framework

- Core interfaces
- General-purpose implementations
- Wrapper and special-purpose implementations
- Abstract implementations for easy reuse
- Separate algorithms library

Example: How to find anagrams

- Alphabetize the characters in each word
 - cat → act, dog → dgo, mouse → emosu
 - Resulting string is called *alphagram*
- Anagrams share the same alphagram!
 - stop → **opst**, post → **opst**, tops → **opst**, opts → **opst**
- So go through word list making “multimap” from alphagram to word!

How to find anagrams in Java (1)

```
public static void main(String[] args) throws IOException {  
    // Read words from file and put into a simulated multimap  
    Map<String, List<String>> groups = new HashMap<>();  
    try (Scanner s = new Scanner(new File(args[0]))) {  
        while (s.hasNext()) {  
            String word = s.next();  
            String alpha = alphabetize(word);  
            List<String> group = groups.get(alpha);  
            if (group == null)  
                groups.put(alpha, group = new ArrayList<>());  
            group.add(word);  
        }  
    }  
}
```

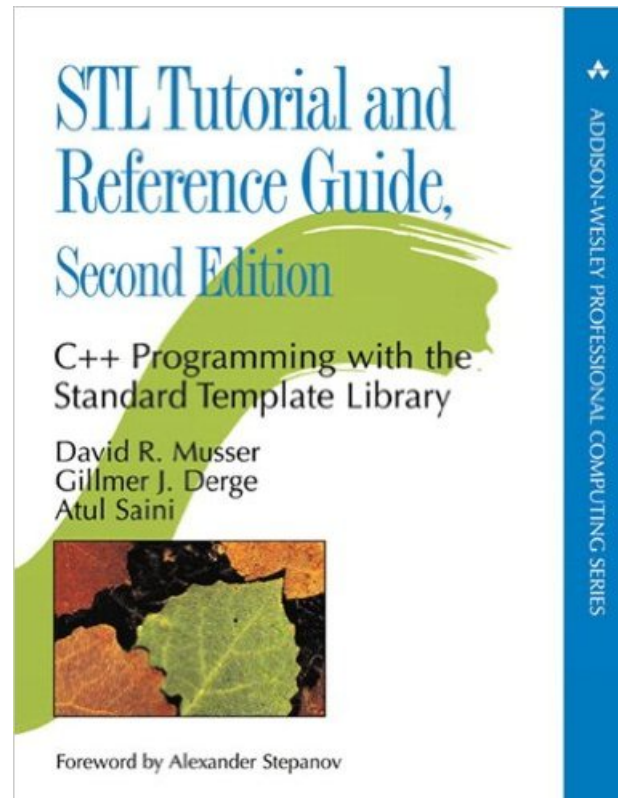
How to find anagrams in Java (2)

```
// Print all anagram groups above size threshold
int minGroupSize = Integer.parseInt(args[1]);
for (List<String> group : groups.values())
    if (group.size() >= minGroupSize)
        System.out.println(group.size() + ": " + group);
}

// Returns the alphagram for a string
private static String alphabetize(String s) {
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
```

Two slides in Java vs. **a chapter** in STL

Java's verbosity is somewhat exaggerated



The Collection interface

```
public interface Collection<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);           // Optional  
    boolean remove(Object element);  // Optional  
    Iterator<E> iterator();  
  
    Object[] toArray();  
    T[] toArray(T a[]);  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); // Optional  
    boolean removeAll(Collection<?> c);      // Optional  
    boolean retainAll(Collection<?> c);      // Optional  
    void clear();                             // Optional  
    ...  
}
```

14

The List interface: an ordered collection

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);        // Optional
    void add(int index, E element);     // Optional
    Object remove(int index);           // Optional
    boolean addAll(int index, Collection<? extends E> c);
                                        // Optional

    int indexOf(Object o);
    int lastIndexOf(Object o);

    List<E> subList(int from, int to);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
}
```

The Set interface: collection of distinct items

```
public interface Set<E> extends Collection<E> {  
}
```

- Adds no methods!
- Specification requires no duplicate items in collection
- The *marker interface* design pattern
 - Marker interfaces add invariants, no code

The Map interface: key-value mapping

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Object get(Object key);
    Object put(K key, V value);    // Optional
    Object remove(Object key);    // Optional
    void putAll(Map<? extends K, ? extends V> t); // Opt.
    void clear();                // Optional

    // Collection views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
}
```

Recall the Iterator interface

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    // Optional  
}
```

Aside: The *factory method* pattern

```
public interface Collection<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);           // Optional  
    boolean remove(Object element);  // Optional  
    Iterator<E> iterator();  
  
    Object[] toArray();  
    T[] toArray(T a[]);  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); // Optional  
    boolean removeAll(Collection<?> c);      // Optional  
    boolean retainAll(Collection<?> c);      // Optional  
    void clear();                             // Optional  
    ...  
}
```

 Defines an interface for creating an iterator, but allows collection implementation to decide which iterator to create.

19

The factory method design pattern

- Problem: Subclasses need to control the type of object created
- Solution: Define an method that constructs the object; subclasses can override the method.
- Consequences:
 - Names can be meaningful, self-documenting
 - Can avoid constructing a new object
 - Might be hard to distinguish factory method from other methods

Other factory method examples

- From `java.util.Collections`:

```
List<T>    emptyList();
```

```
Set<T>     emptySet();
```

```
Map<K,V>   emptyMap();
```

```
Set<T>     singleton(T item);
```

```
List<T>    singletonList(T item);
```

```
List<T>    nCopies(T item);
```

General-purpose implementations

Interface	Implementation
Set	HashSet
List	ArrayList
Queue	ArrayDeque
Deque	ArrayDeque
[stack]	ArrayDeque
Map	HashMap

General-purpose implementations (continued)

Interface	Implementation(s)
List	LinkedList
Set	LinkedHashSet TreeSet EnumSet
Queue	PriorityQueue
Map	LinkedHashMap TreeMap EnumMap

Wrapper and special-purpose implementations

- Unmodifiable collections (from `java.util.Collections`):
`Collection<T> unmodifiableCollection(Collection<? extends T> c);`
`List<T> unmodifiableList(List<? extends T> list);`
`Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m);`
- Synchronized collections (from `java.util.Collections`):
`Collection<T> synchronizedCollection(Collection<? extends T> c);`
`List<T> synchronizedList(List<? extends T> list);`
`Map<K,V> synchronizedMap(Map<? extends K, ? extends V> m);`
- A List backed from an array (from `java.util.Arrays`):
`List<T> asList(T... a);`



The adapter pattern: Returns a specialized list implementation that adapts the array API to the `java.util.List` API

e.g., The UnmodifiableCollection class

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {  
    return new UnmodifiableCollection<>(c);  
}
```

```
class UnmodifiableCollection<E>  
    implements Collection<E>, Serializable {
```

```
    final Collection<E> c;
```

```
    UnmodifiableCollection(Collection<> c) {this.c = c; }
```

```
    public int size() {return c.size();}
```

```
    public boolean isEmpty() {return c.isEmpty();}
```

```
    public boolean contains(Object o) {return c.contains(o);}
```

```
    public Object[] toArray() {return c.toArray();}
```

```
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
```

```
    public String toString() {return c.toString();}
```

```
    public boolean add(E e) {throw new UnsupportedOperationException(); }
```

```
    public boolean remove(Object o) { throw new UnsupportedOperationException(); }
```

```
    public boolean containsAll(Collection<?> coll) { return c.containsAll(coll);}
```

```
    public boolean addAll(Collection<? extends E> coll) { throw new UnsupportedOperationException(); }
```

```
    public boolean removeAll(Collection<?> coll) { throw new UnsupportedOperationException(); }
```

```
    public boolean retainAll(Collection<?> coll) { throw new UnsupportedOperationException(); }
```

```
    public void clear() { throw new UnsupportedOperationException(); }
```

e.g., The UnmodifiableCollection class

What design pattern is this?

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c)
    return new UnmodifiableCollection(c);
}
```

```
class UnmodifiableCollection<E>
    implements Collection<E>, Serializable {
```

```
    final Collection<E> c;
```

```
    UnmodifiableCollection(Collection<E> c) {this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) {throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) {return c.containsAll(coll);}
    public boolean addAll(Collection<? extends E> coll) {throw new UnsupportedOperationException();}
    public boolean removeAll(Collection<?> coll) {throw new UnsupportedOperationException();}
    public boolean retainAll(Collection<?> coll) {throw new UnsupportedOperationException();}
```

e.g., The UnmodifiableCollection class

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c)
    return new UnmodifiableCollection(c);
}
```

**What design
pattern is this?**

```
class UnmodifiableCollection<E>
    implements Collection<E>

    final Collection<E> c;
```

**UnmodifiableCollection
decorates Collection by
removing functionality...**

```
UnmodifiableCollection(Collection<E> c) {this.c = c; }
public int size() {return c.size();}
public boolean isEmpty() {return c.isEmpty();}
public boolean contains(Object o) {return c.contains(o);}
public Object[] toArray() {return c.toArray();}
public <T> T[] toArray(T[] a) {return c.toArray(a);}
public String toString() {return c.toString();}
public boolean add(E e) {throw new UnsupportedOperationException();}
public boolean remove(Object o) {throw new UnsupportedOperationException();}
public boolean containsAll(Collection<?> coll) {return c.containsAll(coll);}
public boolean addAll(Collection<? extends E> coll) {throw new UnsupportedOperationException();}
public boolean removeAll(Collection<?> coll) {throw new UnsupportedOperationException();}
public boolean retainAll(Collection<?> coll) {throw new UnsupportedOperationException();}
```

Abstract implementations for easy reuse

```
public abstract class AbstractList<E> extends AbstractCollection<E>
    implements List<E> {
    abstract public int size();
    abstract public E get(int index);

    public boolean isEmpty() { return size() == 0; }
    public boolean contains(Object element) { ... }
    public boolean add(int index, E element) { throw new UnsupportedOperationException(); }
    public boolean remove(int index) { throw new UnsupportedOperationException(); }
    ...
}
```

Abstract implementations for easy reuse

```
public abstract AbstractList<E> extends AbstractCollection<E>
                                   implements List<E> {
    abstract public int size();
    abstract public E get(int index);

    public boolean isEmpty() { return size() == 0; }
    public boolean contains(Object element) { ... }
    public boolean add(int index, E element) { throw new UnsupportedOperationException; }
    public boolean remove(int index) { throw new UnsupportedOperationException; }
    ...
}
```

Abstract implementations for easy reuse

```
public abstract ArrayList<E> extends AbstractCollection<E>
                                   implements List<E> {
    abstract public int size();
    abstract public E get(int index);

    public boolean isEmpty() { return size() == 0; }
    public boolean contains(Object element) { ... }
    public boolean add(int index, E element) { throw new UnsupportedOperationException; }
    public boolean remove(int index) { throw new UnsupportedOperationException; }
    ...
}
```

**What design
pattern is this?**

Abstract implementations for easy reuse

```
public abstract class AbstractList<E> extends AbstractCollection<E>
    implements List<E> {
    abstract public int size();
    abstract public E get(int index);

    public boolean isEmpty() { return size() == 0; }
    public boolean contains(Object element) { ... }
    public boolean add(int index, E element) { throw new UnsupportedOperationException(); }
    public boolean remove(int index) { throw new UnsupportedOperationException(); }
    ...
}
```

What design pattern is this?

The template method design pattern: size and get are variant, other methods are invariant.

Reusable algorithms in java.util.Collections

```
static <T extends Comparable<? super T>> void sort(List<T> list);  
static int binarySearch(List list, Object key);  
static <T extends Comparable<? super T>> T min(Collection<T> coll);  
static <T extends Comparable<? super T>> T max(Collection<T> coll);  
static <E> void fill(List<E> list, E e);  
static <E> void copy(List<E> dest, List<? Extends E> src);  
static void reverse(List<?> list);  
static void shuffle(List<?> list);
```

e.g. sorting a Collection

- Using `Collections.sort`:

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

Comparable objects – another example

- Make Name comparable:

```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first;  this.last = last;
    }
    ...
}
```

- Hint: Strings implement Comparable<String>

Comparable objects – another example

- Make Name comparable:

```
public class Name implements Comparable<Name> {  
    private final String first, last;  
    public Name(String first, String last) {  
        if (first == null || last == null)  
            throw new NullPointerException();  
        this.first = first; this.last = last;  
    }  
    public int compareTo(Name o) {  
        int lastComparison = last.compareTo(o.last);  
        if (lastComparison != 0) return lastComparison;  
        return first.compareTo(o.first);  
    }  
}
```

Hint: Strings implement Comparable<String>

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);  
}
```

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 SalaryComparator implements Comparator<Employee> {  
    public int compare (Employee o1, Employee o2) {  
        return o1.salary - o2.salary;  
    }  
}
```

Summary

- Collections as reusable and extensible data structures
 - design for reuse
 - design for change
- Iterators to abstract over internal structure
- Decorator to attach behavior at runtime
- Template methods and factory methods to support customization in subclasses
- Adapters to convert between implementations
- Strategy pattern for sorting