Principles of Software Construction: Objects, Design and Concurrency

Design Case Study: Collections

15-214

toad

Christian Kästner Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Learning Goals

- Understand the design aspects of collection libraries
- Understand the design patterns involved and their rationale
  - Marker Interface
  - Decorator
  - Factory Method
  - Iterator
  - Strategy
  - Template Method
  - Adapter
- Knowledge of useful collection functions, including helpers in Collections class

Design patterns we have seen so far:

Composite

Template Method

Strategy

Observer

Façade

Model-View-Controller

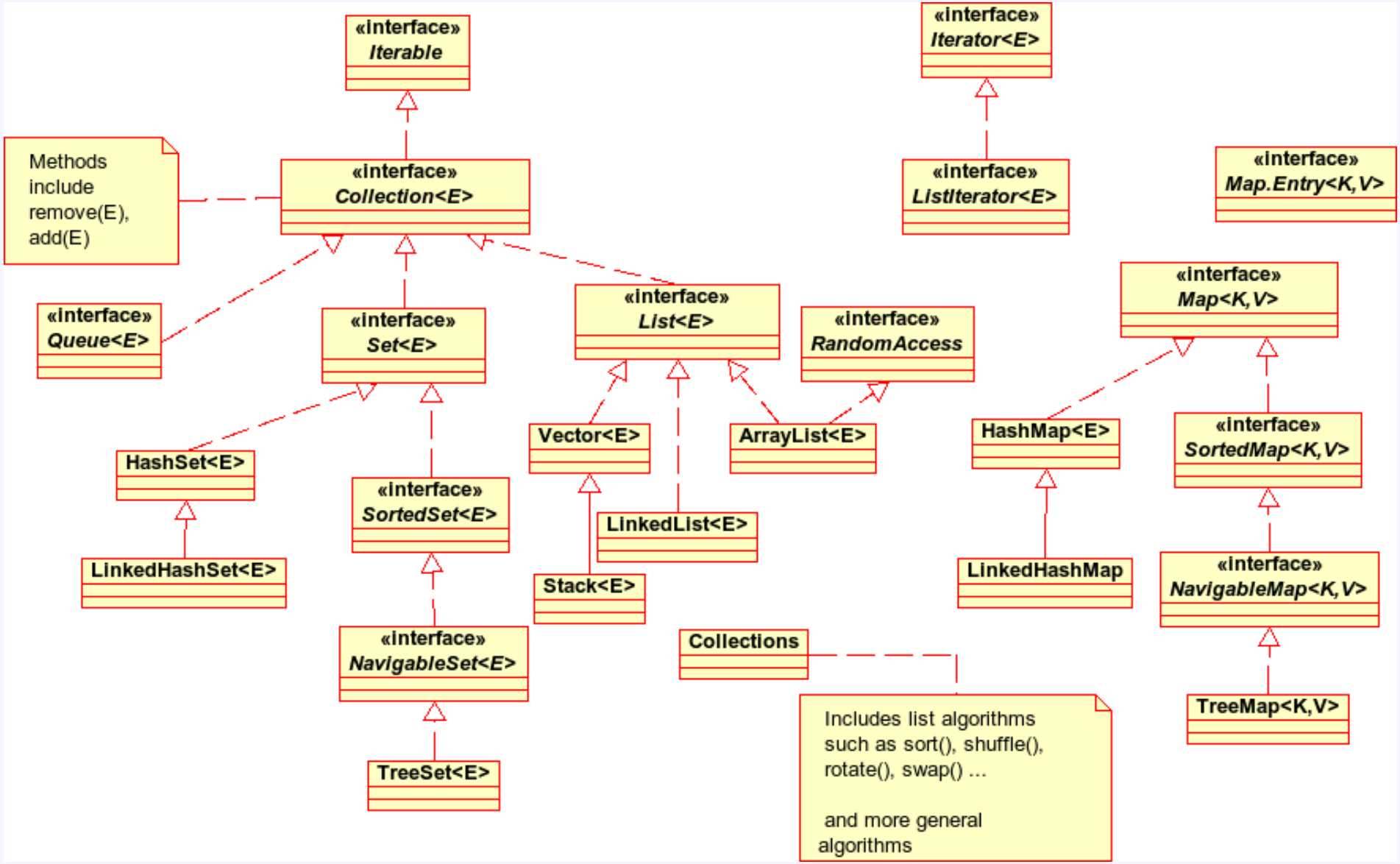
The philosophy of the Collections framework

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

Collection Designs

- Different data types: lists, sets, maps, stacks, queues, …
- Different implementations
  - ArrayList vs LinkedList
  - HashSet vs TreeSet
- Different concepts and designs
  - Mutable vs immutable
  - Sorted vs unsorted
  - Accepts null or not
  - Accepts duplicates or not
  - Concurrency/threadsafe or not
  - …

Collections Library Overview



Source: http://wiki3.cosc.canterbury.ac.nz/index.php/User:Jenny_Harlow/Design_study/Java_Collections_Framework

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);
```

Design Decisions

- Collection represent "group of elements"
  - for lists, queues, sets, stacks, …
- No concept of order or uniqueness
- Seems mutable (clear, add, …)
  - but: Mutation is optional, may throw UnsupportedOperationException
  - Documentation describes whether mutation is supported (invariant as textual specification instead of type checking)
  - Note: Scala separates mutable and immutable collection hierarchies with different interfaces
- Maps are not collections
- Helper functions (sort, search, copy) in separate Collections class

The java.util.List<E> interface

- Defines order of a collection
  - Uniqueness unspecified
- 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>:
  - //adds invariant (textual specification) only
- The *Marker Interface* design pattern
  - Problem: You want to define a behavioral constraint not enforced at compile time.
  - Solution: Define an interface with no methods, but with additional invariants as JavaDoc comment or JML specification

The java.util.Queue<E> interface

- Additional helper methods only
- Behavioral subtype?

- 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

- Map of keys to values; keys are unique
- Does not extend java.util.Collection<E> (Why?)

```
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);
```

- The *Adapter* design pattern
  - ArrayList adapter from array to List interface
  - asList and toArray create copies in other representation (adapter to other interface for read part, not for mutation though)

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 (e.g. all helper functions in Collections) 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);
…
```

Iterators

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);
}
```

- Works for every implementation of **Iterable**
  - Iterable is interface with single "Iterator<T> iterator()" method

The *Iterator* interface

```
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>: 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
}
```

Pair example

```
public class Pair<E> implements Iterable<E> {
        private final E first, second;
        public Pair(E f, E s) { first = f; second=s;}
        public Iterator<E> iterator() {
                return new PairIterator();
        }
        private class PairIterator implements Iterator<E> {
                private boolean seen1=false, seen2=false;
                public boolean hasNext() { return !seen2; }
                public E next() {
                        if (!seen1) { seen1=true; return first; }
                        if (!seen2) { seen2=true; return second; }
                        throw new NoSuchElementException();
                }
                public void remove() { throw new UnsupportedOperati

        }
}
```

```
for (Integer i: new Pair<Integer>(3,4)) println(i);
```

The Iterator design pattern

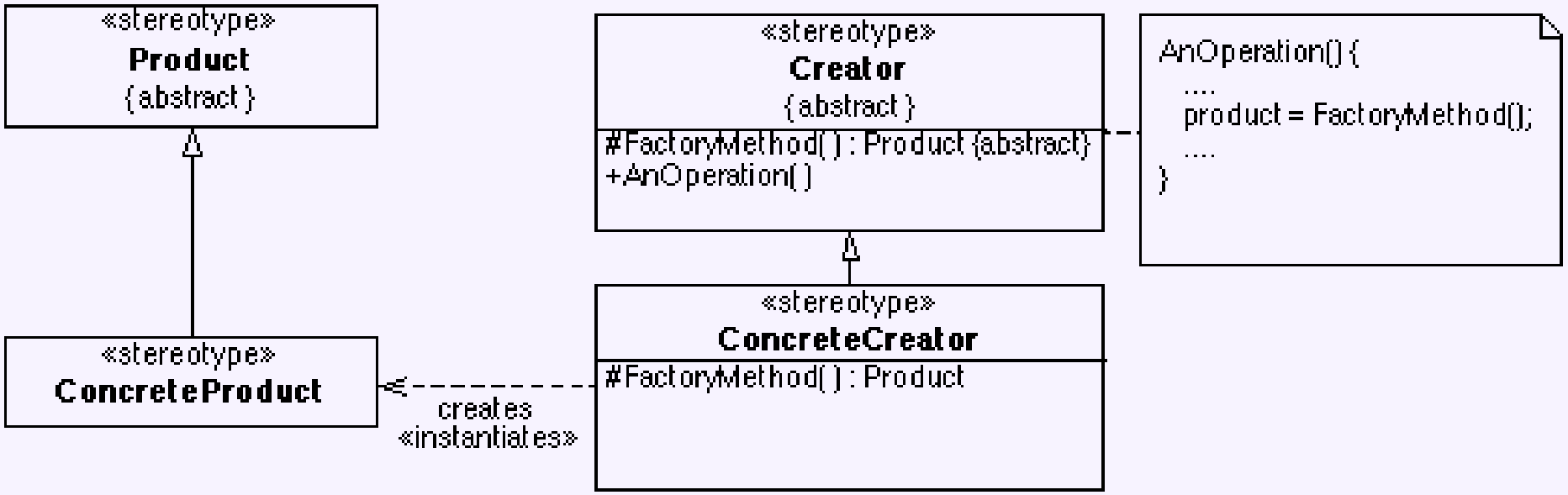
- Provide a strategy to uniformly access all elements of a container in a sequence
  - independent of how the container is implemented
  - independent of how children are stored in the container
  - ordering is unspecified, but every child is only visited once
- Design for change, information hiding
  - Hides internal implementation of container behind uniform explicit interface
- Design for reuse, division of labor
  - Hides complex data structure behind simple interface, simple interface to communicate between parts of the programs

Creating Iterators

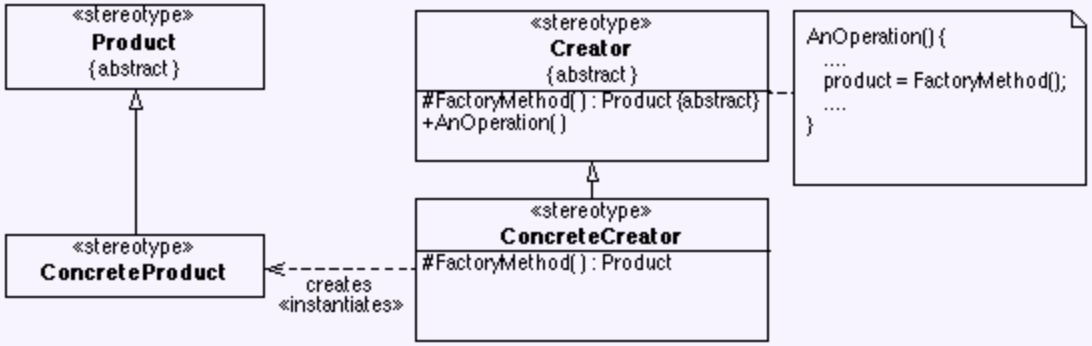
```
public interface Collection<E> {
  boolean      add(E e);
  boolean      addAll(Collection<E> c);
  boolean      remove(E e);
  boolean      removeAll(Collection<E> c);
  boolean      retainAll(Collection<E> c);
  boolean      contains(E e);
  boolean      containsAll(Collection<E> c);
  void         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.

The *Factory Method* design pattern



The *Factory Method* design pattern



- Applicability
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
- Consequences
  - Provides hooks for subclasses to customize creation behavior
  - Connects parallel class hierarchies

Sorting

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);
}
```

- Collections.sort(list) static helper function to sort list of comparable elements

Why not member of List (List.sort())?

- 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;
  }
}
```

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 design pattern** interface for that

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

static helper function Collections.sort(list, comparator)

Tradeoffs

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```

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;
  }
  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());
  }
}
```

Immutable

The java.util.Collections class

- Helper methods in java.util.Collections:

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

- Turn a mutable list into an immutable list
  - All mutation operations on resulting list throw UnsupportedOperationException

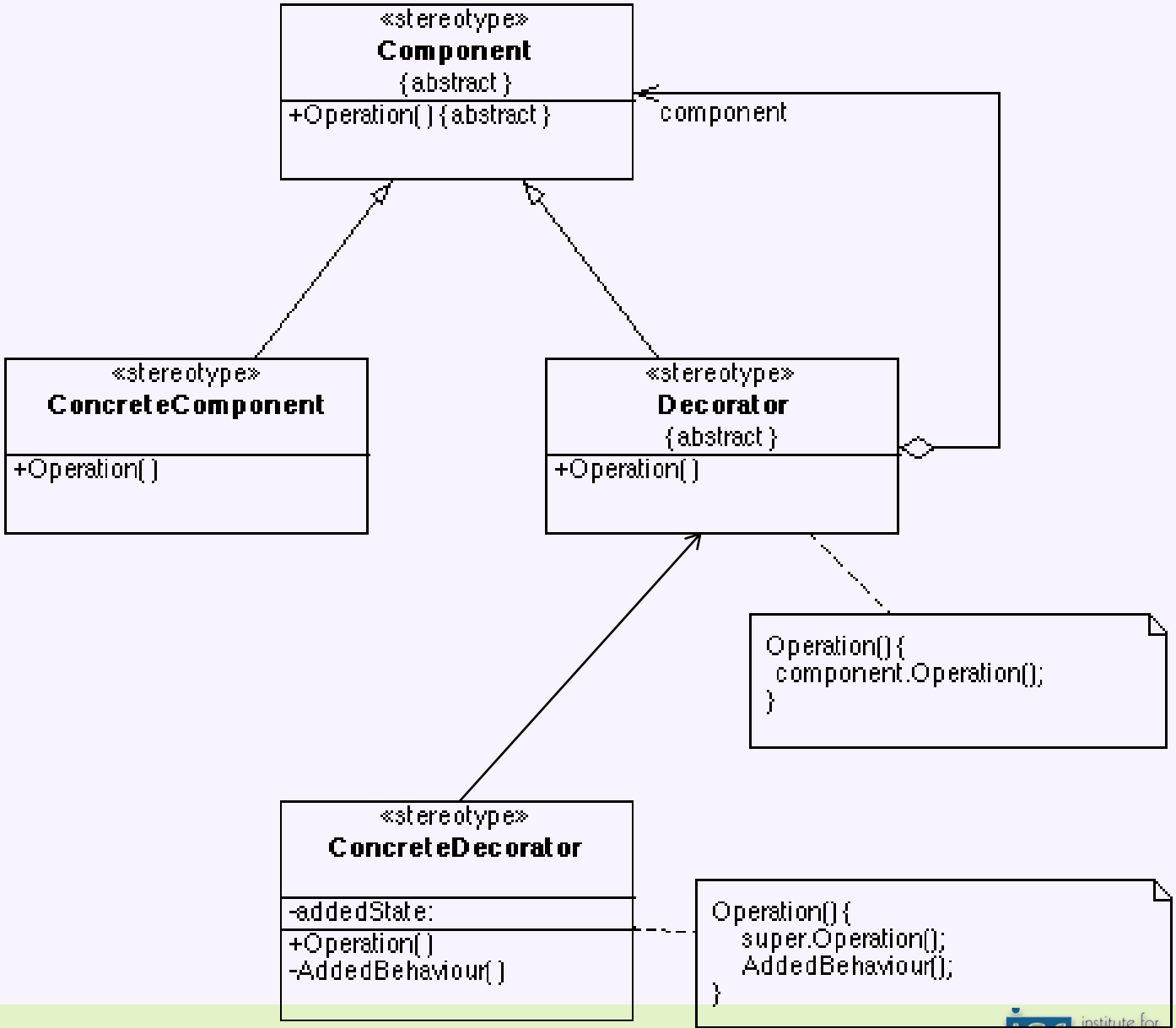
- Similar for synchronization:

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

java.util.Collections (simplified excerpt)

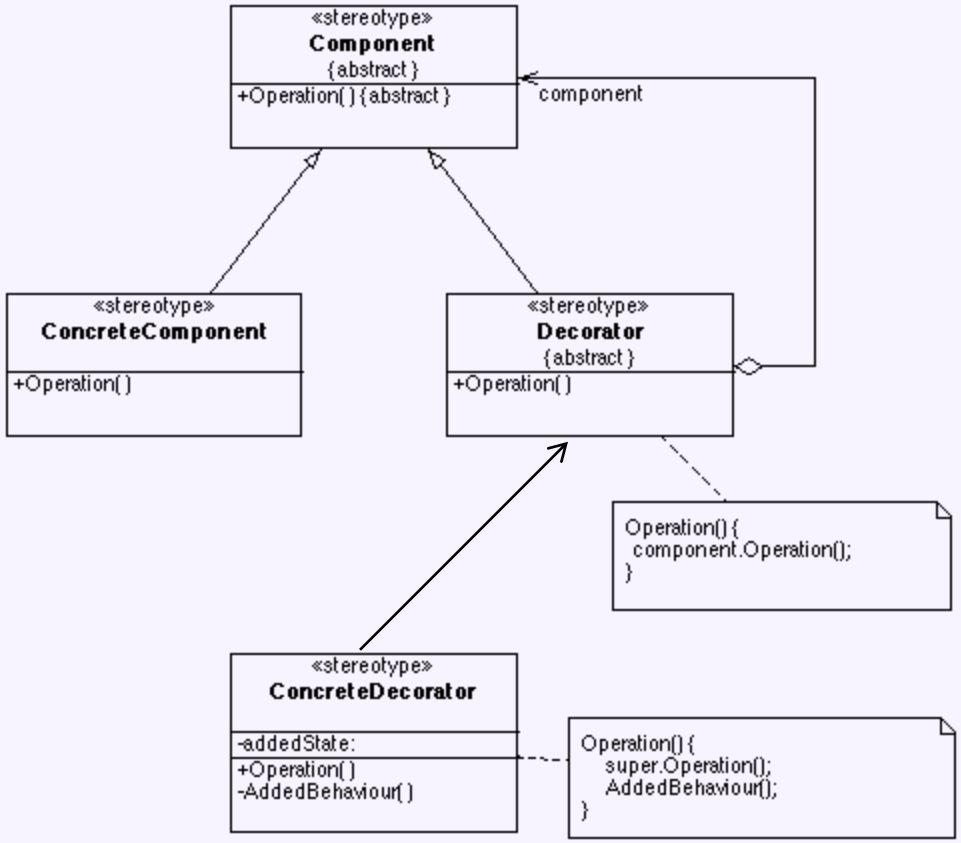
```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
        return new UnmodifiableCollection<>(c);
}
class UnmodifiableCollection<E> implements Collection<E>, Serializable {
        final Collection<E> c;
        UnmodifiableCollection(Collection<> c) {this.c = c; }
        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 UnsupportedOperationI
        public boolean containsAll(Collection<?> coll) {  return c.containsAll(
        public boolean addAll(Collection<? extends E> coll) { throw new Uns
        public boolean removeAll(Collection<?> coll) { throw new Unsupporte
        public boolean retainAll(Collection<?> coll) { throw new Unsupported
        public void clear() {  throw new UnsupportedOperationException();
    }
```

The *Decorator* design pattern



The *Decorator* design pattern

- Applicability
  - To add responsibilities to individual objects dynamically and transparently
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical
- Consequences
  - More flexible than static inheritance
  - Avoids monolithic classes
  - Breaks object identity
  - Lots of little objects



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 behavior changes in subclasses
- Adapters to bridge between implementations
- Strategy pattern for sorting