Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Design Case Studies

Design Case Study: Java Collections

Jonathan Aldrich **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

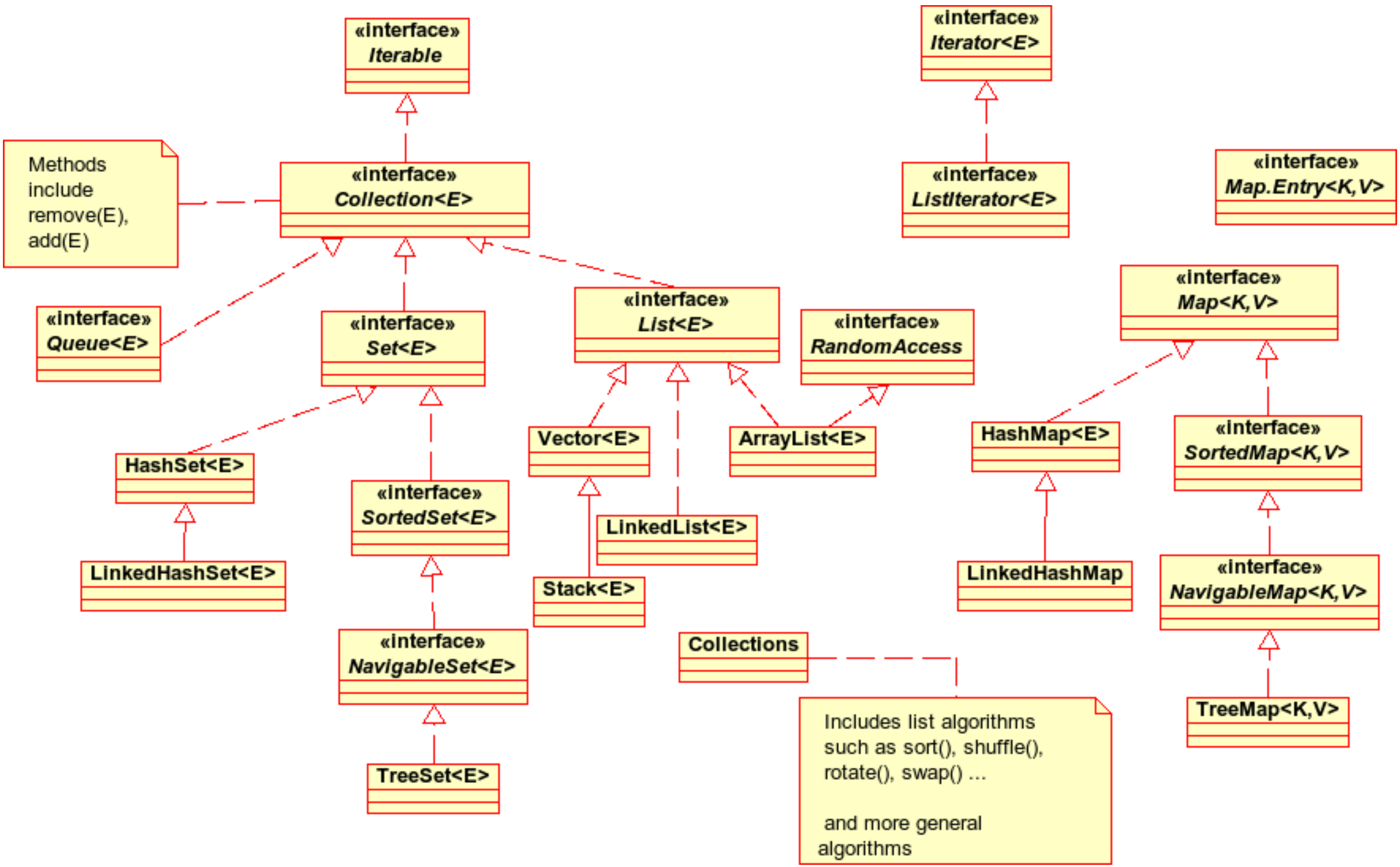
- Homework 4b due next Thursday
- Homework 4a feedback available at end of class
- Midterm exams still available…
- Can earn 75% of lost Homework 4a credit
  - Directly address TA comments when you turn in Homework 4c

Key concepts from Tuesday

Key concepts from Tuesday

- Separation of GUI from core implementation
  - Observer pattern
  - Model-View-Controller (MVC)
- Many design patterns…

Today: Java Collections



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

Learning goals for today

- Understand the design aspects 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
- Be able to use common collection classes, including helpers in the `Collections` class.

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 designs
  - 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
  - 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.Map<K,V>` interface

Map of keys to values; keys are unique:

```
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();
```

Java Collections design decisions

- `Collection` represents group of elements
  - e.g. lists, queues, sets, stacks, …
- No inherent concept of order or uniqueness
- Mutation is optional
  - May throw `UnsupportedOperationException`
  - Documentation describes whether mutation is supported
- Maps are not `Collections`
- Common functions (`sort`, `search`, `copy`, …) in a 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 a Javadoc comment or JML specification.

The `java.util.Queue<E>` interface

- Additional helper methods only
- 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
```

- Aside: Is Queue<E> a behavioral subtype?

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);
```

What design pattern is this?

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
  - `Arrays.asList()` returns an adapter from an array to the `List` interface

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 pattern
abstract int size();                       // Template Method pattern
boolean      set(int i, E e);              // set add remove are
boolean      add(E e);                     // pseudo-abstract,
boolean      remove(E e);                  // Template Methods pattern
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);
…
```

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`

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

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);
}
```

The Iterator design pattern

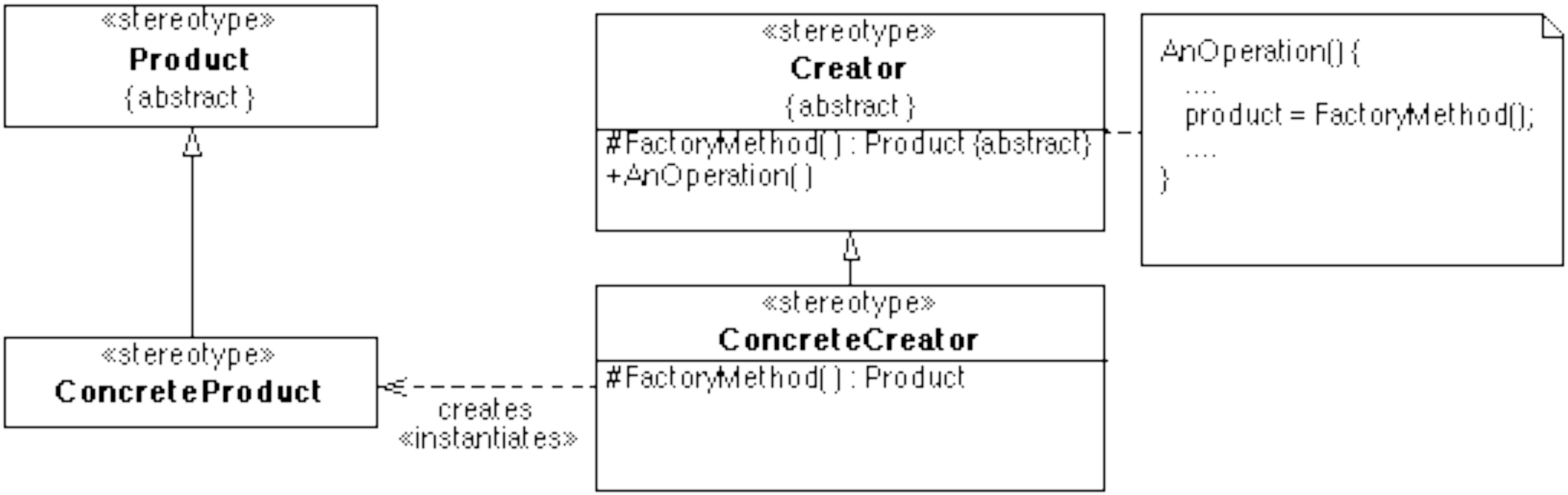
- Provide a strategy to uniformly access all elements of a container in a sequence
  - Independent of the container implementation
  - Ordering is unspecified, but every element visited once
- Design for change, information hiding
  - Hides internal implementation of container behind uniform explicit interface
- Design for reuse, division of labor
  - Hides complex data structure behind simple interface
  - Facilitates communication between parts of the program

Getting an Iterator

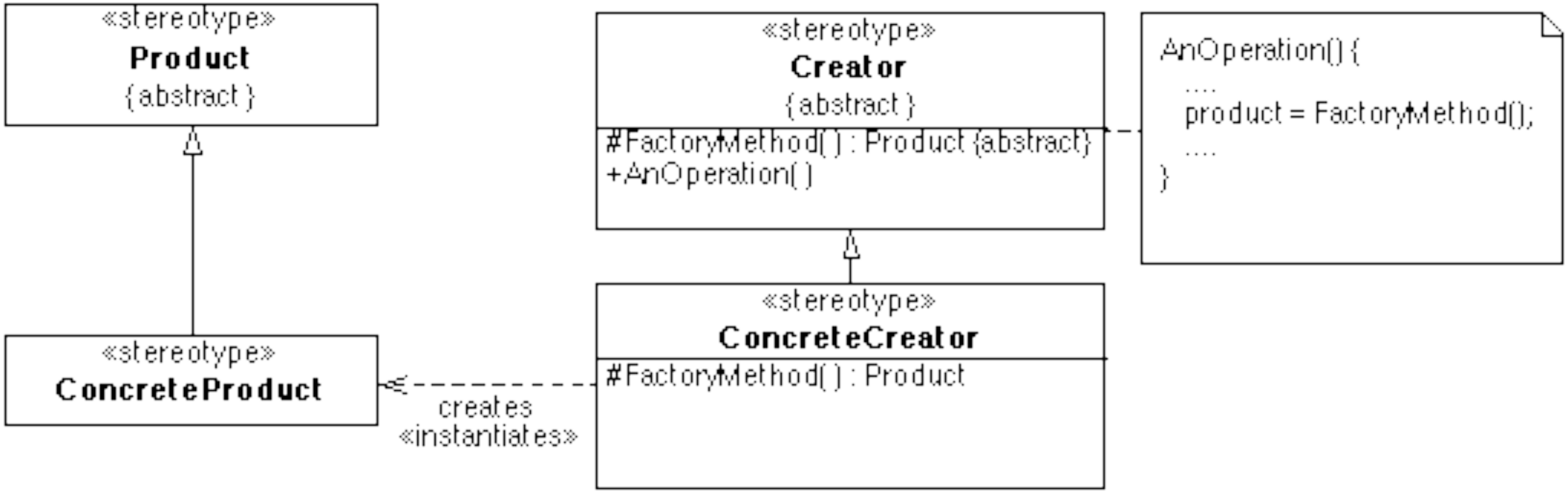
```
public interface Collection<E> {
  boolean      add(E e);
  boolean      addAll(Collection<E> c);
  boolean      remove(E e);
  boolean      removeAll(Collection<E> c);
  boolean      retainAll(Collection<E> c);
  boolean      contains(E e);
  boolean      containsAll(Collection<E> c);
  void   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

An Iterator implementation for Pairs

```
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 UnsupportedOperationExcept

  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

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

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - One way to fix:

```
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:

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

- A hacky aside: 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: Is this the Template Method pattern?

Comparable objects – another example

- Make `Name` comparable:

```
public class Name                                       {
  private final String first;  // not null
  private final String last;   // not null
  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 final String first;  // not null
  private final String last;   // not null
  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);
}
```

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());
  }
}
```

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

- 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);
```

  - e.g., Turn a mutable list into an immutable list
    - All mutable operations on resulting list throw an `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);
```

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 UnsupportedOperationEx
  public boolean   containsAll(Collection<?> coll) { return c.containsAll
  public boolean   addAll(Collection<? extends E> coll) { throw new Unsup
  public boolean   removeAll(Collection<?> coll) { throw new UnsupportedO
```

e.g., The UnmodifiableCollection class

```
public static <T> Collection<T> u
  return new UnmodifiableCollectio
}
```

What design pattern is this?

```
                                   llection<T> 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 UnsupportedOperationEx
  public boolean    containsAll(Collection<?> coll) { return c.containsAll
  public boolean    addAll(Collection<? extends E> coll) { throw new Unsup
  public boolean    removeAll(Collection<?> coll) { throw new UnsupportedO
```

e.g., The `UnmodifiableCollection` class

```
public static <T> Collection<T> u                                llection<T> c)
  return new UnmodifiableCollectio
}

class UnmodifiableCollection<E>
                implements Collectio

  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 UnsupportedOperationEx
  public boolean    containsAll(Collection<?> coll) { return c.containsAll
  public boolean    addAll(Collection<? extends E> coll) { throw new Unsup
  public boolean    removeAll(Collection<?> coll) { throw new UnsupportedO
  public boolean    retainAll(Collection<?> coll) { throw new UnsupportedO
```

What design pattern is this?

`UnmodifiableCollection` decorates `Collection` by removing functionality…

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