Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Design case studies

Collections

Charlie Garrod **Michael Hilton**

School of Computer Science
institute for SOFTWARE RESEARCH

Administriva

- Homework 4b due Thursday, October 19th
  - Don't wait to start!!
- Optional reading due today: Effective Java, Item 1

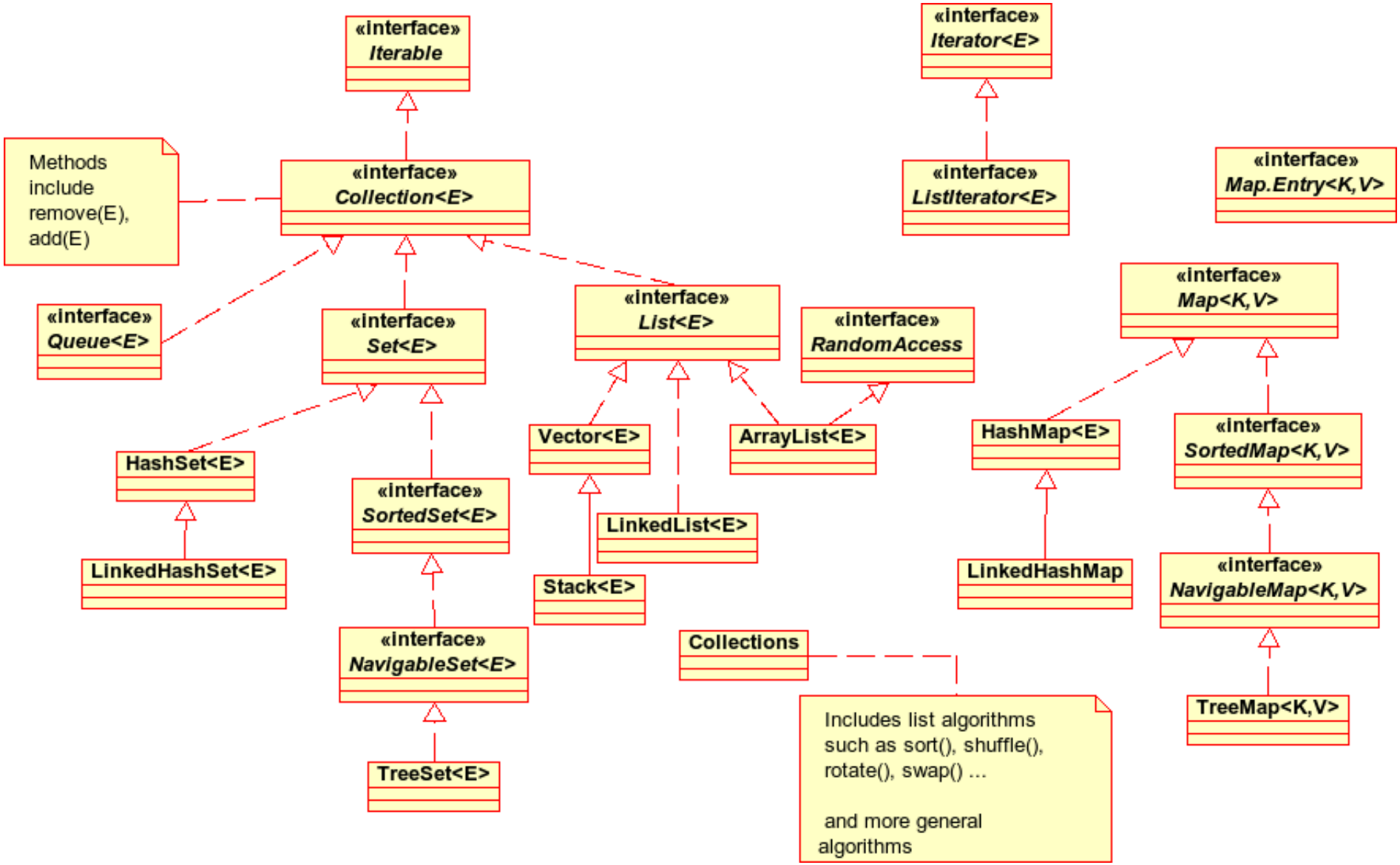
Early Informal Feedback

- Internal course feedback
  - Provide valuable feedback!
- 5 minutes

Review Key Concepts from Tuesday

- Design case study: GUI potpourri
  - Strategy
  - Template method
  - Observer
  - Composite
  - Decorator
  - Adapter
  - Façade
  - Command
  - Chain of responsibility

Today: Java Collections



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
  - …

We take you back now to the late '90s

- It was a simpler time
  - Java had only `Vector`, `Hashtable` & `Enumeration`
  - But it needed more; platform was growing!
- The barbarians were pounding the gates
  - JGL was a transliteration of STL to Java
  - It had 130 (!) classes and interfaces
  - The JGL designers wanted badly to put it in the JDK

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"

Java HashSet vs STL hash_set

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Adds the specified element to this set if it is not already present.
void	**clear()** Removes all of the elements from this set.
Object	**clone()** Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	**contains(Object o)** Returns true if this set contains the specified element.
boolean	**isEmpty()** Returns true if this set contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this set.
boolean	**remove(Object o)** Removes the specified element from this set if it is present.
int	**size()** Returns the number of elements in this set (its cardinality).

begin	Returns an iterator that addresses the first element in the hash_set.
hash_set::cbegin	Returns a const iterator addressing the first element in the hash_set.
hash_set::cend	Returns a const iterator that addresses the location succeeding the last element in a hash_set.
clear	Erases all the elements of a hash_set.
count	Returns the number of elements in a hash_set whose key matches a parameter-specified key.
hash_set::crbegin	Returns a const iterator addressing the first element in a reversed hash_set.
hash_set::crend	Returns a const iterator that addresses the location succeeding the last element in a reversed hash_set.
hash_set::emplace	Inserts an element constructed in place into a hash_set.
hash_set::emplace_hint	Inserts an element constructed in place into a hash_set, with a placement hint.
empty	Tests if a hash_set is empty.
end	Returns an iterator that addresses the location succeeding the last element in a hash_set.
equal_range	Returns a pair of iterators respectively to the first element in a hash_set with a key that is greater than a specified key and to the first element in the hash_set with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a hash_set from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.
get_allocator	Returns a copy of the allocator object used to construct the hash_set.
insert	Inserts an element or a range of elements into a hash_set.
key_comp	Retrieves a copy of the comparison object used to order keys in a hash_set.
lower_bound	Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the hash_set.
rbegin	Returns an iterator addressing the first element in a reversed hash_set.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.
size	Returns the number of elements in the hash_set.
swap	Exchanges the elements of two hash_sets.
upper_bound	Returns an iterator to the first element in a hash_set that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a hash_set.

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
```

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

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

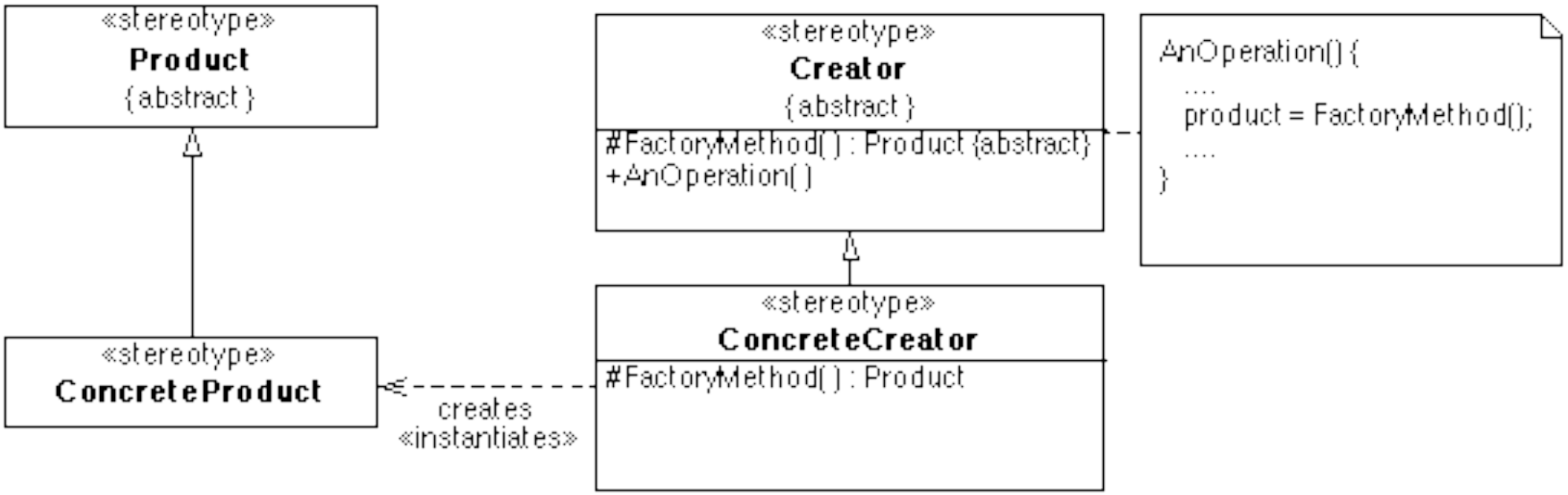
- Provide a strategy to uniformly access all elements of a container in a sequence
  - Independent of the container implementation
  - Ordering is unspecified, but every element visited once
- Design for change, information hiding
  - Hides internal implementation of container behind uniform explicit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

Getting an Iterator

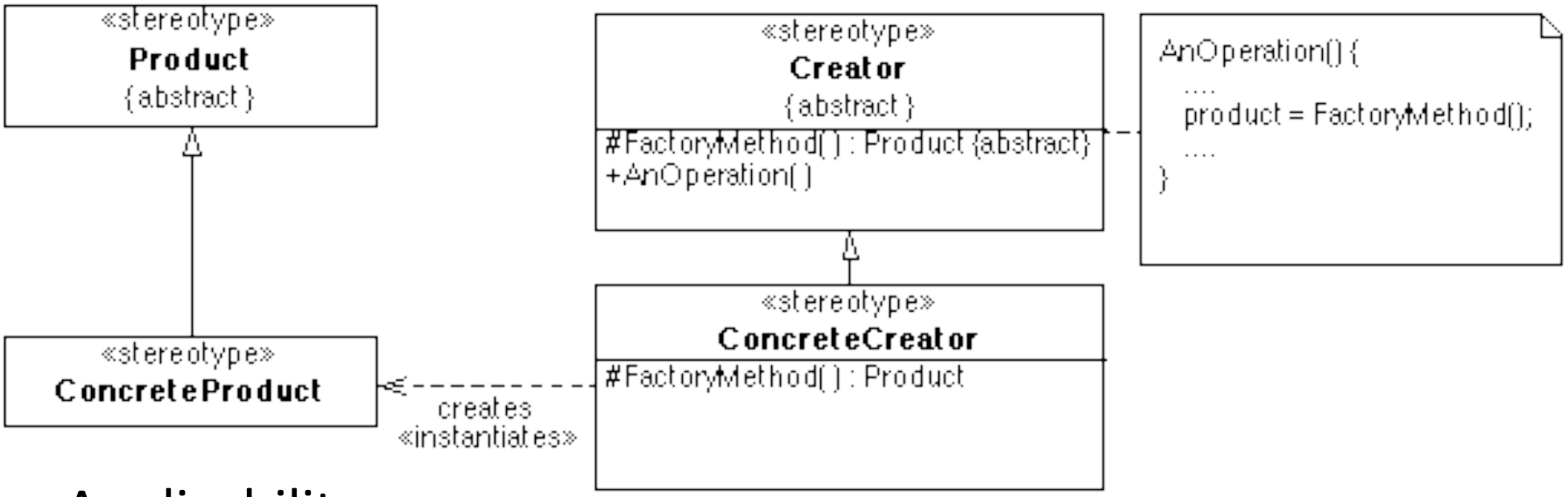
```
public interface Collection<E> {
  boolean      add(E e);
  boolean      addAll(Collection<E> c);
  boolean      remove(E e);
  boolean      removeAll(Collection<E> c);
  boolean      retainAll(Collection<E> c);
  boolean      contains(E e);
  boolean      containsAll(Collection<E> c);
  void   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

Factory Method design pattern

- Advantages:
  - Have names
  - Not required to create a new object
  - Return an object of any subtype
  - Can reduce verbosity
- Disadvantages
  - Classes without public or private constructors cannot be subclassed
  - Not readily distinguishable from other static methods

Factory Method Design Pattern Usage

```
Collections:
  List<T>      emptyList();
  Map<K,V>      emptyMap();
  Set<T>       emptySet();

public List<String> getCountries {
    if (/*some condition*/) {
      …
      return listCountries;
    } else {
      return Collections.emptyList();
    }
  }
```

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 UnsupportedOperationException();
       }
  }
}
```

```
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(”Michael"))
    arguments.remove(”Michael"); // 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) {
    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;  // not null
  private final String last;   // not null
  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);
  }
}
```

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, "Hilton");
    System.out.println("There are " + x +
                       " students named Hilton");
  }
}
```

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 UnsupportedOperationExcepti
   public boolean     containsAll(Collection<?> coll) { return c.containsAll(coll);
   public boolean     addAll(Collection<? extends E> coll) { throw new UnsupportedOp
   public boolean     removeAll(Collection<?> coll) { throw new UnsupportedOperatior
   public boolean     retainAll(Collection<?> coll) { throw new UnsupportedOperatior
   public void clear() {  throw new UnsupportedOperationException();   }
```

e.g., The UnmodifiableCollection class

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

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 UnsupportedOperationExc
  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?

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 UnsupportedOperationExc
  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…

DISCUSSION: DESIGNING AN API (JOSH BLOCH)

Joshua Bloch

From Wikipedia, the free encyclopedia

Joshua J. Bloch (born August 28, 1961) is a software engineer and a technology author, formerly employed at Sun Microsystems and Google. He led the design and implementation of numerous Java platform features, including the Java Collections Framework, the `java.math` package, and the `assert` mechanism.[1] He is the author of the programming guide *Effective Java* (2001), which won the 2001 Jolt Award,[2] and is a co-author of two other Java books, *Java Puzzlers* (2005) and *Java Concurrency In Practice* (2006).

Bloch holds a B.S. in computer science from Columbia University and a Ph.D. in computer science from Carnegie Mellon University.[1] His 1990 thesis was titled *A Practical Approach to Replication of Abstract Data Objects*[3] and was nominated for the ACM Distinguished Doctoral Dissertation Award.[4]

Bloch has worked as a Senior Systems Designer at Transarc, and later as a Distinguished Engineer at Sun Microsystems. In June 2004, he left Sun and became Chief Java Architect at Google.[5] On August 3, 2012, Bloch announced that he would be leaving Google.[6]

In December 2004, *Java Developer's Journal* included Bloch in its list of the "Top 40 Software People in the World".[7]

Joshua J. Bloch

Bloch in 2008

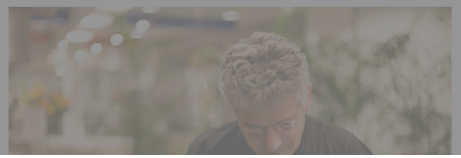
Born	August 28, 1961 (age 56) Southampton, New York
Alma mater	Columbia University, Carnegie Mellon University
Occupation	Professor of the Practice at Carnegie Mellon University
Spouse(s)	Cynthia Bloch
Children	Matthew Bloch Timothy Bloch

Bloch has proposed the extension of the Java programming language with two features: Concise Instance Creation Expressions (CICE) (coproposed with Bob Lee and Doug Lea) and Automatic Resource Management (ARM) blocks. The combination of CICE and ARM formed one of the three early proposals for adding support for closures to Java.[8] ARM blocks were added to the language in JDK7.[9]

Bloch is currently a faculty member of the Institute for Software Research at Carnegie Mellon University, where he holds the title "Professor of the Practice". In addition to his research, Bloch teaches coursework in Software Engineering, course 15-214.

Joshua Bloch

From Wikipedia, the free encyclopedia



Joshua J. Bloch (born August 28, 1961) is a software engineer and a technology author, formerly employed at Sun Microsystems and Google. He led the design and implementation of numerous Java platform features, including the Java Collections Framework, the `java.math` package, and the `assert` mechanism.[1] He is the author of the programming guide *Effective Java* (2001), which won the 2001 Jolt Award,[2] and is a co-author of two other Java books, *Java Puzzlers* (2005) and *Java Concurrency In Practice* (2006).

In December 2004, *Java Developer's Journal* included Bloch in its list of the "Top 40 Software People in the World".[7]

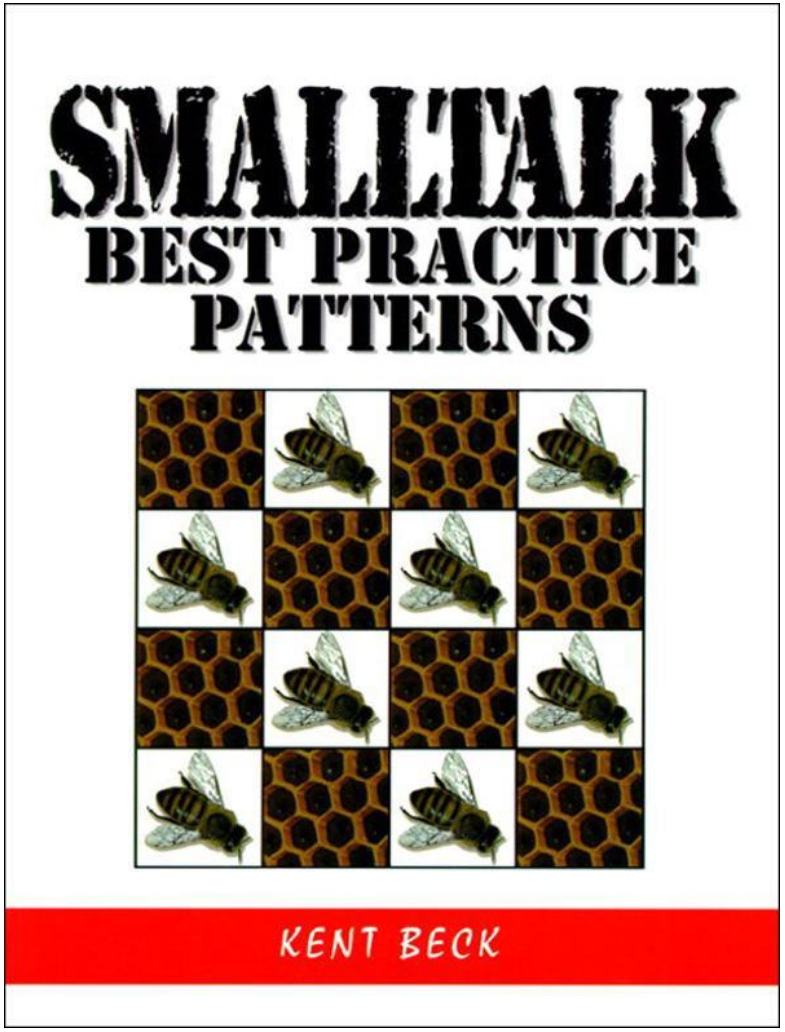
Children Matthew Bloch Timothy Bloch

Bloch has proposed the extension of the Java programming language with two features: Concise Instance Creation Expressions (CICE) (coproposed with Bob Lee and Doug Lea) and Automatic Resource Management (ARM) blocks. The combination of CICE and ARM formed one of the three early proposals for adding support for closures to Java.[8] ARM blocks were added to the language in JDK7.[9]

Bloch is currently a faculty member of the Institute for Software Research at Carnegie Mellon University, where he holds the title "Professor of the Practice". In addition to his research, Bloch teaches coursework in Software Engineering, course 15-214.

A wonderful source of use cases

"Good artists copy, great artists steal." – Pablo Picasso



You must maintain an *issues list*

- Centralizes all open and closed design issues
- List pros and cons for each possible decision
- Essential for efficient progress
- Forms the basis of a design rationale

The first draft of API was not so nice

- Map was called `Table`
- No `HashMap`, only `Hashtable`
- No algorithms (`Collections`, `Arrays`)
- Contained some unbelievable garbage

Automatic alias detection
A horrible idea that died on the vine

```
/**
 * This interface must be implemented by Collections and Tables that are
 * <i>views</i> on some backing collection.  (It is necessary to
 * implement this interface only if the backing collection is not
 * <i>encapsulated</i> by this Collection or Table; that is, if the
 * backing collection might conceivably be be accessed in some way other
 * than through this Collection or Table.)  This allows users
 * to detect potential <i>aliasing</i> between collections.
 * <p>
 * If a user attempts to modify one collection
 * object while iterating over another, and they are in fact views on
 * the same backing object, the iteration may behave erratically.
 * However, these problems can be prevented by recognizing the
 * situation, and "defensively copying" the Collection over which
 * iteration is to take place, prior to the iteration.
 */

public interface Alias {
    /**
     * Returns the identityHashCode of the object "ultimately backing" this
     * collection, or zero if the backing object is undefined or unknown.
     * The purpose of this method is to allow the programmer to determine
     * when the possiblity of <i>aliasing</i> exists between two collections
     * (in other words, modifying one collection could affect the other).
       This
     * is critical if the programmer wants to iterate over one collection and
     * modify another; if the two collections are aliases, the effects of
     * the iteration are undefined, and it could loop forever.  To avoid
     * this behavior, the careful programmer must "defensively copy" the
     * collection prior to iterating over it whenver the possibility of
     * aliasing exists.
     * <p>
     * If this collection is a view on an Object that does not impelement
     * Alias, this method must return the IdentityHashCode of the backing
     * Object.  For example, a List backed by a user-provided array would
     * return the IdentityHashCode of the array.
     * If this collection is a <i>view</i> on another Object that implements
     * Alias, this method must return the backingObjectId of the backing
     * Object.  (To avoid the cost of recursive calls to this method, the
     * backingObjectId may be cached at creation time).
     * <p>
     * For all collections backed by a particular "external data source" (a
     * SQL database, for example), this method must return the same value.
     * The IdentityHashCode of a "proxy" Object created just for this
     * purpose will do nicely, as will a pseudo-random integer permanently
     * associated with the external data source.
     * <p>
     * For any collection backed by multiple Objects (a "concatenation
     * view" of two Lists, for instance), this method must return zero.
     * Similarly, for any <i>view</i> collection for which it cannot be
     * determined what Object backs the collection, this method must return
     * zero.  It is always safe for a collection to return zero as its
     * backingObjectId, but doing so when it is not necessary will lead to
     * inefficiency.
     *  <p>
     * The possibility of aliasing between two collections exists iff
     * any of the following conditions are true:<ol>
     *             <li>The two collections are the same Object.
     *             <li>Either collection implements Alias and has a
     *                 backingObjectId that is the identityHashCode of
     *                 the other collection.
     *             <li>Either collection implements Alias and has a
     *                 backingObjectId of zero.
     *             <li>Both collections implement Alias and they have equal
     *                 backingObjectId's.</ol>
     *
     * @see java.lang.System#identityHashCode
     * @since JDK1.2
     */
    int backingObjectId();
}
```

I received a *lot* of feedback

- Initially from a small circle of colleagues
  - Some *very* good advice
  - Some not so good
- Then from the public at large: beta releases
  - Hundreds of messages
  - Many API flaws were fixed in this stage
  - I put up with a lot of flaming

Review from a ***very*** senior engineer

```
API                vote    notes
=================================================================
Array              yes       But remove binarySearch* and toList
BasicCollection    no        I don't expect lots of collection classes
BasicList          no        see List below
Collection         yes       But cut toArray
Comparator         no
DoublyLinkedList   no        (without generics this isn't worth it)
HashSet            no
LinkedList         no        (without generics this isn't worth it)
List               no        I'd like to say yes, but it's just way
                             bigger than I was expecting
RemovalEnumeration no
Table              yes       BUT IT NEEDS A DIFFERENT NAME
TreeSet            no

I'm generally not keen on the toArray methods because they add complexity

Simiarly, I don't think that the table Entry subclass or the various
views mechanisms carry their weight.
```

III. Evolution of Java collections

Release, Year	Changes
JDK 1.0, 1996	Java Released: `Vector`, `Hashtable`, `Enumeration`
JDK 1.1, 1996	(No API changes)
J2SE 1.2, 1998	Collections framework added
J2SE 1.3, 2000	(No API changes)
J2SE 1.4, 2002	`LinkedHash{Map,Set}`, `IdentityHashSet`, 6 new algorithms
J2SE 5.0, 2004	Generics, for-each, enums: generified everything, `Iterable` `Queue`, `Enum{Set,Map}`, concurrent collections
Java 6, 2006	`Deque`, `Navigable{Set,Map}`, `newSetFromMap`, `asLifoQueue`
Java 7, 2011	No API changes. Improved sorts & defensive hashing
Java 8, 2014	Lambdas (+ streams and internal iterators)

Some things I wish I'd done differently

- Algorithms should return collection, not void or boolean
  - Turns ugly multiliners into nice one-liners

```
private static String alphabetize(String s) {
    return new String(Arrays.sort(s.toCharArray()));
}
```

- Collection should have `get()`, `remove()`
  - Queue and Deque eventually did this
- `Sorted{Set,Map}` should have proper navigation
  - `Navigable{Set,Map}` are warts

Conclusion – Josh Bloch

- **It takes a lot of work to make something that appears obvious**
  - Coherent, unified vision
  - Willingness to listen to others
  - Flexibility to accept change
  - Tenacity to resist change
  - Good documentation!
- **It's worth the effort!**
  - A solid foundation can last two+ decades