

## 6.170 Hall of Shame

Returning collections of nodes or edges from BipartiteGraph. Which is better?

```
Object[] blackNodes();  
List blackNodes();  
Set blackNodes();  
Iterator blackNodes();
```

## 6.170 Hall of

Returning collections of BipartiteGraph. Which

```
Object[] blackNodes();  
List blackNodes();  
Set blackNodes();  
Iterator blackNodes();
```

Object[] is bad because it's  $O(n)$  in space as well as time.

Set is better than List because it expresses important properties of the node collection (nodes are unique, and order doesn't matter).

Iterator is the weakest specification, so it is best for the graph implementer because it gives the most flexibility. But Set/List offer more power to the client: not just the ability to iterate, but also to get size.

## Case Study: The Java Collections Framework

### OUTLINE

- Reviewing/Exploring Java Collections Framework
- Evaluate design decisions
- What's new in Java 1.5

## Case Study: The Java Collections Framework

### OUTLINE

- **Reviewing/Exploring Java Collections Framework**
- Evaluate design decisions
- What's new in Java 1.5

## The Java Collections Framework Review

What is the Java Collections Framework?

**Interfaces:** abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation.

**Implementations:** concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.

**Algorithms:** methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

## Collections Framework Subtleties: Exhibit A

What does the following code print out?

```
List l = new ArrayList();  
l.add("Miller");  
l.add("Fitzpatrick");  
Iterator i = l.iterator();  
l.add("Noto");  
System.out.println(i.next());  
System.out.println(i.next());  
System.out.println(i.next());
```

## Collections Framework Subtleties: Exhibit A

Exception in thread "main" java.util.ConcurrentModificationException  
at java.util.AbstractList\$Itr.checkForComodification(AbstractList.java:444)  
at java.util.AbstractList\$Itr.next(AbstractList.java:417)

```
List l = new ArrayList();  
l.add("Miller");  
l.add("Fitzpatrick");  
Iterator i = l.iterator();  
l.add("Noto");  
System.out.println(i.next());  
System.out.println(i.next());  
System.out.println(i.next());
```

## Collections Framework Subtleties: Exhibit A

From Iterator API...

The **behavior of an iterator is unspecified** if the underlying collection is modified while the iteration is in progress in any ...

From ArrayList API...

The iterators returned by this class's `iterator` and `listIterator` methods are **fail-fast**: if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. Note that the fail-fast behavior of an iterator **cannot be guaranteed**... Fail-fast iterators throw `ConcurrentModificationException` on a **best-effort basis**... *the fail-fast behavior of iterators should be used only to detect bugs* (emphasis in original).

## Collections Framework Subtleties: Exhibit B

What does the following code print out?

```
List list = new ArrayList();  
list.add("Miller");  
list.add("Fitzpatrick");  
list.add("TBD");  
List view = list.subList(2, list.size());  
view.set(0, "Noto");  
System.out.println(list.get(0));  
System.out.println(list.get(1));  
System.out.println(list.get(2));
```

## Collections Framework Subtleties: Exhibit B

What does the following code print out?

```
List list = new ArrayList();  
list.add("Miller");  
list.add("Fitzpatrick");  
list.add("TBD");  
List view = list.subList(2, list.size());  
view.set(0, "Noto");  
System.out.println(list.get(0));  
System.out.println(list.get(1));  
System.out.println(list.get(2));
```

Miller  
Fitzpatrick  
Noto

## Collections Framework Subtleties: Exhibit B

```
public List subList(int fromIndex, toIndex);
```

...

All methods first check to see if the actual `modCount` of the backing list is equal to its expected value, and throw a `ConcurrentModificationException` if it is not.

## Collections Framework Subtleties: Exhibit C

From Set API...

```
public boolean add(Object o)
```

Adds the specified element to this set if it is not already present (**optional operation**)... Individual set implementations should clearly document any restrictions on the the elements that they may contain.

**throws:**

`UnsupportedOperationException` - if the `add` method is not supported by this set.

## Collections Framework Subtleties: Exhibit C

From HashMap API...

### public Set **keySet()**

Returns a set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from this map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. **It does not support the add or addAll operations.**

## Back to the Hall of Shame

Reconsider the following spec (and simple implementation):

```
public Set blackNodes() {
    Set blackNodes =
        new HashSet(blackNodesMap.keySet());
    return blackNodes;
}
```

## Back to the Hall of Shame

A new implementation...

```
public Set blackNodes() {
    //Set blackNodes =
    // new HashSet(blackNodesMap.keySet());
    //return blackNodes;
    return blackNodesMap.keySet();
}
```

## Back to the Hall of Shame

A new implementation...

```
public Set blackNodes() {
    //Set blackNodes =
    // new HashSet(blackNodesMap.keySet());
    //return blackNodes;
    return blackNodesMap.keySet();
}
```

**REPEXPOSURE!**



## Back to the Hall of Shame

One last try...

```
public Set blackNodes() {
    Set s = blackNodesMap.keySet();
    return Collections.unmodifiableSet(s);
}
```

## Recap of Java Collection Framework Subtleties

- `ConcurrentModificationException`
- `UnsupportedOperationException`
- “views” (`subList`, `keySet`)

## Case Study: The Java Collections Framework

### OUTLINE

- Reviewing/Exploring Java collections
- **Evaluate design decisions**
  - Type hierarchy
  - Interface design
  - Strategy design pattern
  - Polymorphism
- What's new in Java 1.5

## Design Decisions: Type hierarchy

- Framework uses interfaces for specification-based decoupling
- Skeletal implementation classes factor out shared code among related implementations.

## Design Decisions: Interface design

Each collections interface specifies “optional methods” that the implementer is not required to implement (if not implemented, the implementer can merely throw an `UnsupportedOperationException` runtime exception). Why was this done?

## Design decisions: Strategy pattern

Java Collections Framework: [sort/binarySearch/min/max](#)

The **Strategy** design pattern is an algorithm that is decoupled from the code that uses it.

### NOTES:

- `sort/binarySearch` require either that the object implements `Comparable` or that the client provides a `Comparator`.
- `comparison` suffers from the same problems as `equality` when it comes to using mutable objects (e.g. `Date`).

## Design decisions: Polymorphism

All collections take elements of type `Object`. This means that they allow you to make different kinds of containers: lists of `Integers`, lists of `URLs`, etc., with only minimal “casting boilerplate”.

```
List bookmarks = new LinkedList();
URL u = ...;
bookmarks.add(u);
..
URL x = (URL) bookmarks.get(0);
```

## Case Study: The Java Collections Framework

- Reviewing/Exploring Java collections
- Evaluate design decisions
- **What's new in Java 1.5**

## Java 1.5: from the “guru” himself (Bloch)

*“The new language features all have one thing in common: they take some common idiom and provide linguistic support for it. In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler.”*

## Java 1.5: “generics” and “enhanced for loop”

generics: Java support for “parametric polymorphism”. The collections will be retrofitted to support this feature.

enhanced ‘for’ loop: a step towards cleaning-up standard iteration over elements in a collection.

let’s see how they will look...

## Java 1.5: “generics”

### OLD WAY:

```
//requires: c != null
//          c must contain only TimerTask objects
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask)i.next();
        tt.cancel();
    }
}
```

### NEW WAY:

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i=c.iterator(); i.hasNext(); ) {
        TimerTask tt = i.next();
        tt.cancel();
    }
}
```

## Java 1.5: “enhanced for loop”

### OLD WAY:

```
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

### NEW WAY:

```
void cancelAll(Collection c) {
    for (Object o : c)
        ((TimerTask)o).cancel();
}
```

## “generics” + “enhanced for loop” = easy-to-read goodness!

### OLD WAY:

```
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

### NEW WAY:

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask task : c)
        task.cancel();
}
```

## Java 1.5: further reading

### Interview with Bloch:

[http://java.sun.com/features/2003/05/bloch\\_qa.html](http://java.sun.com/features/2003/05/bloch_qa.html)

### Java One conference slides:

<http://servlet.java.sun.com/javaone/sf2003/conf/sessions/display-3072.en.jsp>

**Reminder:** Java 1.5 is still in Beta and can’t be used for 6.170!!!