Java Collections Framework
Overview

• The *Java Collections Framework (JCF)* is a group of interfaces and classes similar to the OSU CSE components
  – The similarities will become clearly evident from examples
  – See Java libraries package `java.util`

• There are some important differences, too, however, that deserve mention (at the end)
Overview of Interfaces

- Iterable
- Collection
  - Set
    - Sorted-Set
    - Navigable-Set
  - List
  - Queue
  - Deque
- Map
  - Sorted-Map
  - Navigable-Map
Overview of Interfaces

Iterable

Collection

Set

List

Queue

Map

Sorted-Map

Navigable-Map

Sorted-Set

Navigable-Set

Note: Map does not extend Collection; but it is a “collection”.
Overview of Interfaces

Iterables are in java.lang (because of its intimate connection to for-each loops), but Iterators are in java.util.
Overview of Interfaces

Subsequent slides discuss only certain interfaces.

```
Iterable

Collection

Set

List

Queue

Sorted-Set

Navigable-Set

Map

Sorted-Map

Navigable-Map
```
The `Collection<E>` Interface

- Essentially a **finite multiset of E**
- No direct/efficient way to ask how many “copies” of a given element there are
- Two interesting methods to create arrays of the elements
- Many methods (including `add`, `remove`, `clear`) are “optional”
The **Set\(<E>\)** Interface

- Essentially a **finite set of E**
- No `removeAny` or similar method, so you must use `iterator` to iterate over a `Set`
  - Recall (from `Iterator`): “The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress [except using `Iterator.remove`].”
- Many methods (including `add`, `remove`, `clear`) are “optional”
The List\langle E \rangle Interface

• Essentially a *string of E*
• Access by position (similar to Sequence from OSU CSE components)
• Many methods (including add, remove, clear) are “optional”
• Two interesting additional features:
  – Sublist “views” of a List
  – A special two-way ListIterator
The `List<E>` Interface

- Essentially a `string of E`
- Access by position (similar to `Sequence` from OSU CSE components)
- Many methods (including `add`, `remove`, `clear`) are “optional”
- Two interesting additional features:
  - Sublist “views” of a `List`
  - A special two-way `ListIterator`
The Queue\textless E\textgreater Interface

• Essentially a \textit{string of E}

• Access at ends (similar to Queue from OSU CSE components)

• Here, \texttt{add} and \texttt{remove} are \textit{not} “optional”
  − \texttt{add} is similar to \texttt{enqueue} for OSU CSE components’ Queue
  − \texttt{remove} is similar to \texttt{dequeue}

• Curious names for other methods, e.g., \texttt{offer, peek, poll}
The `Map<K,V>` Interface

- Essentially a **finite set of (K,V)** with the function property
- No `removeAny` or similar method, so you must use `iterator` (somewhat indirectly) to iterate over a `Map`
- Many methods (including `put`, `remove`, `clear`) are “optional”
- Like `List`, a `Map` supports “views” of its elements
Views in the JCF

• A **view** is a “subcollection” of a collection
  – Not a *copy* of some of the elements, but rather “a collection within a collection” that is manipulated “in place”

• Views for **Map**:
  – Keys: `Set<K> keySet()`
  – Values: `Collection<V> values()`
  – Pairs: `Set<Map.Entry<K,V>> entrySet()`
Views in the JCF

• A **view** is a “subcollection” of a collection
  – Not a copy of some of the elements, but rather “a collection within a collection” that is manipulated “in place”

• Views for **Map**:
  – **Keys**: `Set<K> keySet()`
  – **Values**: `Collection<V> values()`
  – **Pairs**: `Set<Map.Entry<K,V>> entrySet()`

`Map.Entry<K,V>` in the JCF is very similar to `Map.Pair<K,V>` in the OSU CSE components.
Example: `Map<String, Integer> m`

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>`m = {(&quot;PB&quot;, 99),</td>
<td></td>
</tr>
<tr>
<td>(&quot;BK&quot;, 42),</td>
<td></td>
</tr>
<tr>
<td>(&quot;SA&quot;, 42)}`</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Set&lt;String&gt; s = m.keySet();</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>`m = {(&quot;PB&quot;, 99),</td>
<td></td>
</tr>
<tr>
<td>(&quot;BK&quot;, 42),</td>
<td></td>
</tr>
<tr>
<td>(&quot;SA&quot;, 42)}`</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>`s = {&quot;SA&quot;, &quot;BK&quot;,</td>
<td></td>
</tr>
<tr>
<td>&quot;PB&quot;}`</td>
<td></td>
</tr>
</tbody>
</table>
Example: `Map<String, Integer> m`

Note all the aliases here! There is no problem in this case because `String` is immutable, but consider the potential problems if it were not.

```java
Set<String> s = m.keySet();
```

```
State
m = {("PB", 99),
     ("BK", 42),
     ("SA", 42)}

s = {"SA", "BK", "PB"}
```
**Example:** `Map<String, Integer> m`  

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| `m = {("PB", 99),
      ("BK", 42),
      ("SA", 42)}` | `m = {("PB", 99),
       ("BK", 42),
       ("SA", 42)}` |
| `Collection<Integer> c = m.values();` | `m = {("PB", 99),
       ("BK", 42),
       ("SA", 42)}` |
|                                      | `c = {42, 99, 42}` |
Example: `Map<String, Integer> m`

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = { (&quot;PB&quot;, 99), (&quot;BK&quot;, 42) }</code></td>
<td><code>m = { (&quot;PB&quot;, 99), (&quot;BK&quot;, 42) }</code></td>
</tr>
<tr>
<td><code>Set&lt;Map.Entry&lt;String, Integer&gt;&gt; s = m.entrySet();</code></td>
<td><code>s = { (&quot;BK&quot;, 42), (&quot;PB&quot;, 99) }</code></td>
</tr>
</tbody>
</table>
View “Backed By” Collection

• A view is **backed by** the underlying collection, which means that if the view is modified then the underlying (“backing”) collection is also modified, and vice versa
  – See Javadoc for supported modifications
  – Be especially careful when iterating over a view of a collection and trying to modify it
**Example:** \( \text{List}\langle\text{Integer}\rangle \ s \)

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{s} = \langle10, 7, 4, -2\rangle)</td>
<td>(s = \langle10, 7, 4, -2\rangle)</td>
</tr>
<tr>
<td>(\text{s.subList(1,3).clear();})</td>
<td>(s = \langle10, -2\rangle)</td>
</tr>
</tbody>
</table>
**Example:** Map<String, Integer> m

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = {(&quot;PB&quot;, 99),&lt;br&gt;(&quot;BK&quot;, 42),&lt;br&gt;(&quot;SA&quot;, 42)}</td>
<td>m = {(&quot;PB&quot;, 99),&lt;br&gt;(&quot;SA&quot;, 42)}</td>
</tr>
<tr>
<td>m.values().remove(42);</td>
<td></td>
</tr>
<tr>
<td>m = {(&quot;PB&quot;, 99),&lt;br&gt;(&quot;SA&quot;, 42)}</td>
<td></td>
</tr>
</tbody>
</table>
Example: Map<String, Integer> m

Because remove for Collection (assuming it is supported for m.values!) removes one copy, we do not know which pair remains in m.

```
m = {"PB", 99),
     ("BK", 42),
     ("SA", 42) }
```

```
m.values().remove(42),
```

```
m = {"PB", 99),
     ("SA", 42)
```
Could `remove` Cause Trouble?

- The *object (dynamic) type* of `m.values()` in the above code might be an implementation of `List` or of `Queue` – But not of `Set`; why not?
- Could the *optional* `remove` method not be supported by the object type of `m.values()`?
Could **remove** Cause Trouble?

Yes! But only when the backing Map does not support remove(). Otherwise, it is supported because the informal Javadoc for the `values` method says:

“The collection supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Collection.remove`, `removeAll`, `retainAll` and `clear` operations. It does not support the `add` or `addAll` operations.”

• Could the optional `remove` method not be supported by the object type of `m.values()`?
Iterating Over a Map

- Because Map does not extend Iterable, but Collection (hence Set) does extend Iterable, you can (only) iterate over a Map using one of its three views:
  - Keys: \( \text{Set}<K> \) keySet()
  - Values: \( \text{Collection}<V> \) values()
  - Pairs: \( \text{Set}<\text{Map.Entry}<K,V>> \) entrySet()
Overview of Collection Classes

There are no classes that directly and fully implement Collection.
AbstractCollection

• Has code for many methods (shared, and possibly overridden, by all later implementations of Collection):
  – add
  – remove
  – clear
  – ...

AbstractCollection

- Has code for many methods (shared, and possibly overridden, by all later implementations of Collection):
  - add
  - remove
  - clear
  - ...

This method’s implementation here, for example, “always throws an UnsupportedOperationException.”
Overview of *Set* Classes

```
Set
  ↓
Abstract-Collection
  ↓
AbstractSet
  ↓
HashSet  TreeSet
  ↑
Abstract-Collection
  ↑
Collection
  ↑
Iterable
  ↑
Object
```
AbstractSet

• Has code for these methods (shared, and possibly overridden, by all later implementations of Set):
  – equals
  – hashCode
  – removeAll
HashSet

- Uses **hashing** in the *Set* representation
- Has code for these methods (overriding those in *AbstractSet*):
  - add
  - remove
  - clear
  - clone
HashSet

- Uses **hashing** in the *Set* representation
- Has code for these methods (overriding those in *AbstractSet*):
  - `add`
  - `remove`
  - `clear`
  - `clone`

The first three methods, though “optional”, are implemented here and do what you should expect.
HashSet

• Uses **hashing** in the **Set** representation
• Has code for these methods (overriding those in **AbstractSet**):
  - add
  - remove
  - clear
  - clone

  The **clone** method “makes a shallow copy”, i.e., the elements are not “cloned”; which raises many questions. **Best practice**: do not use it!
TreeSet

• Uses a \textit{balanced binary search tree} as the \texttt{Set} representation

• Has code for several methods (overriding those in \texttt{AbstractSet})
Overview of List Classes

- *List*
  - *Iterable*
    - *Collection*
      - *Abstract-Collection*
        - *AbstractList*
          - *ArrayList*
          - *LinkedList*
AbstractList

- Has code for many methods (shared, and possibly overridden, by all later implementations of List)
- Similar to AbstractSet but with code for many more methods (because List has many more potentially layered methods than Set)
ArrayList

- Uses *arrays* in the *List* representation
- Has code for many methods (overriding those in *AbstractList*)
LinkedLIst

• Uses a *doubly-linked list* as the List representation
• Has code for many methods (overriding those in AbstractList)
• There is even more detail to the interfaces and abstract classes related to LinkedList, which you can look up if interested
Overview of Map Classes

Map

AbstractMap

HashMap

TreeMap

Object
AbstractMap

• Has code for many methods (shared, and possibly overridden, by all later implementations of Map)
• Similar to AbstractSet but with code for many more methods (because Map has many more potentially layered methods than Set)
HashMap

• Uses **hashing** in the **Map** representation
• Has code for many methods (overriding those in **AbstractMap**)
TreeMap

- Uses a *balanced binary search tree* as the `Map` representation
- Has code for several methods (overriding those in `AbstractMap`)

14 April 2021
JCF Algorithms: Collections

• A number of useful algorithms (and simple but convenient utilities) to process collections are *static methods* in the class *Collections*, e.g.:
  
  - sort
  - reverse
  - min, max
  - shuffle
  - frequency
JCF Algorithms: Collections

- A number of useful algorithms (and simple but convenient utilities) to process collections are static methods in the class Collections, e.g.:
  - sort
  - reverse
  - min, max
  - shuffle
  - frequency

Notice that the class Collections is different from the interface Collection, and in particular it does not implement that interface!
JCF Utilities: Arrays

- A number of useful algorithms (and simple but convenient utilities) to process built-in arrays are *static methods* in the class Arrays, e.g.:
  - sort
  - fill
  - deepEquals
  - deepHashCode
  - deepToString
OSU CSE vs. JCF Components

- The OSU CSE components are similar in design to the JCF interfaces and classes.
- Though some differences can be attributed to pedagogical concerns, there are other important technical differences, too!
Difference #1: Level of Formalism

- JCF interfaces include only informal Javadoc comments for contracts (rather than using explicit mathematical models and requires/ensures clauses)
  - JCF descriptions and contracts use similar terms, though; e.g., “collections” may:
    - be “ordered” or “unordered”
    - “have duplicates” or “not have duplicates”
Difference #1: Level of Formalism

JCF `java.util.Set<E>`:

```java
boolean add(E e)
```

Adds the specified element to this set if it is not already present (optional operation). More formally, adds the specified element `e` to this set if the set contains no element `e2` such that 

\[(e==null \ ? \ e2==null : e.equals(e2))\].

If this set already contains the element, the call leaves the set unchanged and returns `false`. In combination with the restriction on constructors, this ensures that sets never contain duplicate elements.

The stipulation above does not imply that sets must accept all elements; sets may refuse to add any particular element, including `null`, and throw an exception, as described in the specification for `Collection.add`. Individual set implementations should clearly document any restrictions on the elements that they may contain.

Throws:

- `UnsupportedOperationException` - if the add operation is not supported by this set
- `ClassCastException` - if the class of the specified element prevents it from being added to this set
- `NullPointerException` - if the specified element is null and this set does not permit null elements
- `IllegalArgumentException` - if some property of the specified element prevents it from being added to this set
Difference #1: Level of Formalism

OSU CSE components.set.Set<T>:

```java
void add(T x)
    Adds x to this.

Aliases:
    reference x

Updates:
    this

Requires:
    x is not in this

Ensures:
    this = #this union {x}
```
Difference #1: Level of Formalism

Hypothetical OSU CSE components.set.Set<T>:

```java
boolean add(T x)
```

Can you write a formal contract for the `add` method as it is designed in `java.util.Set`?
Difference #1: Level of Formalism

• JCF interfaces include only informal Javadoc comments for contracts (rather than using explicit mathematical models and requires/ensures clauses).

Warning about the JCF documentation:
The interface/class “summary” at the top of the Javadoc-generated page sometimes contains information that is missing from, or even apparently contradictory to, the method descriptions; e.g.:

• iterator for SortedSet
• a few methods for PriorityQueue

14 April 2021
OSU CSE
Difference #2: Parameter Modes

• JCF interfaces do not have any notion of parameter modes (rather than using them in contracts to help clarify and simplify behavioral descriptions)
  – If the JCF used parameter modes, though, the default mode also would be “restores”, as with the OSU CSE components
Difference #3: Aliasing

• JCF interfaces almost never explicitly mention aliasing (rather than advertising aliasing when it may arise)
  – JCF components also are not designed to try to avoid aliasing whenever possible, as the OSU CSE components are
Difference #4: Null

- JCF interfaces generally permit null references to be stored in collections (rather than having a blanket prohibition against null references)
  - JCF components do, however, sometimes include warnings against null references, which the OSU components always prohibit
Difference #5: Optional Methods

• JCF interfaces generally have “optional” methods (rather than requiring all methods to behave according to their specifications in all implementations)
  – JCF implementations of the same interface are therefore *not* plug-compatible: “optional” methods have bodies, but calling one might simply throw an exception:
    `UnsupportedOperationException`
Difference #6: Copy Constructors

• By convention, every class in the JCF has two “standard” constructors:
  – A no-argument constructor
  – A conversion constructor that “copies” references to the elements of its argument, which is another JCF collection
Difference #6: Copy Constructors

• By convention, every class in the JCF has two “standard” constructors:
  – A *no-argument constructor*
  – A *conversion constructor* that “copies” references to the elements of its argument, which is another JCF collection

This no-argument constructor creates an empty collection.
Difference #6: Copy Constructors

• By convention, every class in the JCF has two “standard” constructors:
  – A no-argument constructor
  – A conversion constructor that “copies” references to the elements of its argument, which is another JCF collection

Presumably, “copying” from a collection that may have duplicates, to one that may not, simply removes extra copies.
Difference #7: Exceptions

• Violation of what might have been considered a precondition leads to a specific exception being thrown (rather than simply a conceptual contract violation, which might or might not be checked using `assert`)

  – Example: an attempt to remove an element from an empty `Queue` is specified to result in a `NoSuchElementException`
Difference #8: Kernel Methods

• A single JCF interface usually contains all methods applicable to a type (rather than “kernel” methods being separated into a separate interface from all other methods)
  – JCF uses abstract classes, however, to provide default implementations of methods that presumably would be implemented in abstract classes in the OSU CSE components
  – Other JCF methods are like “kernel” methods
Resources

• *The Collections Framework (from Oracle)*

• *Effective Java, Third Edition*