CSE 12
The Java Collections Framework

- The Java Collections Framework
- The Collection and List Interfaces
- Ambiguities in Interface Specification
- Methods Inherited from the Object Class
- Iterators and the Iterator Pattern
Data structures, collections, containers

- A data structure is... an object that contains data

- Another word for a data structure is a “collection” or “container”

- Different programming language libraries provide various data structures that can be used by application programmers

- In Java, this library is called the Java Collections Framework (JCF)

- The JCF is a collection of interfaces, abstract and concrete classes providing a standard set of collection (container) types, in the \texttt{java.util} package
Interfaces in JCF
List implementations in JCF
The Collection\(<E>\) interface

• A very important interface in the JCF is Collection\(<E>\)

• We will look at the Javadoc documentation of this interface

• Collection\(<E>\) specifies 15 instance methods which cover most general-purpose data structure operations

• So, a class declaring itself to implement Collection\(<E>\) must provide definitions for all the methods, or it will not compile

• However, in the documentation of Collection\(<E>\), some of these methods are listed as “optional operations”...

• Question: What do all the optional operations have in common?
The `Collection<E>` interface

- java.util
  Interface `Collection<E>`

- All Superinterfaces: `Iterable<E>`

- All Known Subinterfaces: `BeanContext`, `BeanContextServices`, `BlockingDeque<E>`, `BlockingQueue<E>`, `Deque<E>`, `List<E>`, `NavigableSet<E>`, `Queue<E>`, `Set<E>`, `SortedSet<E>`

- All Known Implementing Classes: `AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractList`, `ArrayBlockingQueue`, `ArrayDeque`, `ArrayList`, `AttributeList`, `BeanContextServicesSupport`, `BeanContextSupport`, `ConcurrentLinkedQueue`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `EnumSet`, `HashSet`, `JobStateReasons`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedHashSet`, `LinkedList`, `PriorityBlockingQueue`, `PriorityQueue`, `RoleList`, `RoleUnresolvedList`, `Stack`, `SynchronousQueue`, `TreeSet`, `Vector`
public interface Collection<E> extends Iterable<E>

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw UnsupportedOperation Exception if this collection does not support the operation.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically NullPointerException or ClassCastException.
### Collection\(<E>\) interface operations

- **boolean add** (\(E\) \(e\))
  
  Ensures that this collection contains the specified element (optional operation).

- **boolean addAll** (\(\text{Collection}\)\(<\text{? extends } E\)\(\)\(c\))
  
  Adds all of the elements in the specified collection to this collection (optional operation).

- **void clear** ()
  
  Removes all of the elements from this collection (optional operation).

- **boolean contains** (\(\text{Object}\) \(o\))
  
  Returns true if this collection contains the specified element.

- **boolean containsAll** (\(\text{Collection}\)\(<\text{?}> c\))
  
  Returns true if this collection contains all of the elements in the specified collection.
Collection\(<E>\) interface operations

- boolean **equals** (Object o)
  
  Compares the specified object with this collection for equality.

- int **hashCode** ()
  
  Returns the hash code value for this collection.

- boolean **isEmpty** ()
  
  Returns true if this collection contains no elements.

- Iterator\(<E>\)**iterator** ()
  
  Returns an iterator over the elements in this collection.

- boolean **remove** (Object o)
  
  Removes a single instance of the specified element from this collection, if it is present (optional operation).
Collection<E> interface operations

- boolean **removeAll** (Collection<? c)
  Removes all of this collection's elements that are also contained in the specified collection (optional operation).

- boolean **retainAll** (Collection<? c)
  Retains only the elements in this collection that are contained in the specified collection (optional operation).

- int **size** ()
  Returns the number of elements in this collection.

- **Object[] toArray** ()
  Returns an array containing all of the elements in this collection.

- **<T> T[] toArray** (T[] a)
  Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
Collection\<E\> add method

- Those descriptions of the Collection\<E\> operations are short, summary descriptions…
- Let’s look at the longer, more complete description of one of those operations:

```java
boolean add (E e)

Ensures that this collection contains the specified element (optional operation).
```
Collection\<E\> add method

- boolean \texttt{add(E e)}
- Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)
- Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.
- If a collection refuses to add a particular element for any reason other than that it already contains the element, it \textit{must} throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns.
Collection\(<E>\) add method

- **Parameters:** \(e\) - element whose presence in this collection is to be ensured
- **Returns:** true if this collection changed as a result of the call
- **Throws:**
  - `UnsupportedOperationException` - if the add operation is not supported by this collection
  - `ClassCastException` - if the class of the specified element prevents it from being added to this collection
  - `NullPointerException` - if the specified element is null and this collection does not permit null elements
  - `IllegalArgumentException` - if some property of the element prevents it from being added to this collection
  - `IllegalStateException` - if the element cannot be added at this time due to insertion restrictions
Collection<E> interface options

• Note that the longer description is more detailed, but it allows many different options

• Are null elements allowed? Are duplicate elements allowed? Etc.

• Any implementation of the Collection<E> interface must make choices about those options, and those choices should be documented for that implementation

• Let’s go on and look at the List<E> interface, a subinterface of Collection<E>…
The `List<E>` interface

- `java.util` Interface `List<E>`

- All Superinterfaces: `Collection<E>`, `Iterable<E>`

- All Known Implementing Classes: 
  `AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`
List\langle E \rangle \text{ interface description}

- public interface List\langle E \rangle extends Collection\langle E \rangle
- An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
- Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements $e_1$ and $e_2$ such that $e_1.equals(e_2)$, and they typically allow multiple null elements if they allow null elements at all.
- The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods.
- The List interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based.
List<E> interface operations

• The List<E> interface specifies 10 methods beyond the 15 inherited from its superinterface Collection<E>.
• And the List<E> interface specifies somewhat different behavior for some of the Collection<E> methods
• As one example of this, consider the add(E) method. Its short description in Collection<E> is:
  • boolean **add** (E e)
    Ensures that this collection contains the specified element (**optional operation**).
• The short description in List<E> is:
  • boolean **add** (E e)
    Appends the specified element to the end of this list (**optional operation**).
• Let’s look at the long description…
List<E> add method

• boolean add(E e)

• Appends the specified element to the end of this list (optional operation).

• Lists that support this operation may place limitations on what elements may be added to this list. In particular, some lists will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. List classes should clearly specify in their documentation any restrictions on what elements may be added.

• Specified by: add in interface Collection<E>

• Parameters: e - element to be appended to this list

• Returns: true (as specified by Collection.add(E))
List<E> add method, continued

- **Throws:**
  - `UnsupportedOperationException` - if the add operation is not supported by this list
  - `ClassCastException` - if the class of the specified element prevents it from being added to this list
  - `NullPointerException` - if the specified element is null and this list does not permit null elements
  - `IllegalArgumentException` - if some property of this element prevents it from being added to this list
List\(<E>\) interface options

- The difference in the specification of the add\((E)\) operation for List\(<E>\) is due to the fact that a List is a linear positional data structure: there is a first element, second, …, last element. Collections need not be positional.
- However, the add\((E)\) operation of List\(<E>\) still allows many choices in implementation: Are null elements allowed? Etc.
- Let’s look at one implementation of List\(<E>\) in the JCF to see how it makes these implementation choices: the LinkedList\(<E>\) class…
The LinkedList\<E\> class

- **java.util**
  Class LinkedList\<E\>

- **Type Parameters**: \(E\) - the type of elements held in this collection

- **All Implemented Interfaces**:
  - `Serializable`, `Cloneable`, `Iterable\<E\>`, `Collection\<E\>`, `Deque\<E\>`, `List\<E\>`, `Queue\<E\>`
**LinkedList<E> class description**

- public class `LinkedList<E>` extends `AbstractSequentialList<E>` implements `List<E>`, `Deque<E>`, `Cloneable`, `Serializable`

- Linked list implementation of the List interface. Implements all optional list operations, and permits all elements (including null).

- All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.
LinkedList<E> add method

- public boolean add(E e)
- Appends the specified element to the end of this list.
- Specified by: add in interface Collection<E>
- Specified by: add in interface List<E>
- Parameters: e - element to be appended to this list
- Returns: true (as specified by Collection.add(E))

- So, the LinkedList<E> class makes these implementation choices:
  - allows duplicate elements,
  - allows null elements,
  - and its add(E) method throws no exceptions
Methods Required by an Interface

• A Java interface specifies instance methods that any class implementing the interface must provide.
• The Java code in an interface only shows method headers; together with comments and other documentation this specifies what the methods should do.
• It is then up to the implementing class to provide implementations for those methods.
• The implementing class can inherit method implementations from superclasses; the method definitions do not need to be written inside the implementing class itself.
• Because Object is a superclass of every other class, every class has all those methods.
• However, they may need to be overridden…
In Java, the Object class is an ancestor of every other class. So, in Java, every object is a Object. This means that every public method of the Object class is inherited by every Java class. Some of these methods of the Object class should be overridden in subclasses and customized as appropriate for that subclass. The Object class has 9 public instance methods… Three of these should be overridden in every subclass: equals(Object), hashCode(), toString()
Method inheritance: equals(Object)

The equals(Object) method as defined in Object:
public boolean equals(Object o) {
    return this == o;
}

/**
 * Show the use of the equals(Object) method inherited from Object.
 */
public class ShapeEx3 {
    public static void main ( String [] args ) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();
        Object r3 = new Rectangle();
        Rectangle r4 = new Rectangle();
        r4.setLength( 4 ); r4.setHeight( 25 );
        System.out.println(r1.equals(r1));
        System.out.println(r1.equals(r2));
        System.out.println(r1.equals(r3));
        System.out.println(r1.equals(r4));
    }
}
/**
 * Indicates whether this object is "equal to" the argument object.
 * Overridden from <tt>Object</tt>.
 * @return <tt>true</tt> if equal, <tt>false</tt> otherwise
 */
public boolean equals(Object o) {
    // check that argument is-a Rectangle
    if (!(o instanceof Rectangle) ) return false;
    // downcast to Rectangle
    Rectangle other = (Rectangle) o;
    // access instance variables to test equality
    return this.length == other.length &&
           this.width == other.width;
}
Method inheritance: hashCode()

- For certain data structures to work correctly, the following must be true:
  - For any data objects \( a, b \) in the structure:
    \[
    \text{if } a\text{.equals}(b)\text{, then } a\text{.hashCode}() == b\text{.hashCode}()
    \]

- Therefore, if you override `equals()` in a class you are defining, then you should also override `hashCode()` to make it consistent with `equals()` in this way

- One way to do that for the Rectangle class is shown next…
Rectangle: Overriding hashCode()
Iterators and the JCF

• Before implementing an interface, or writing tests for a class that implements an interface, it is important to understand what the operations defined in the interface are supposed to do.

• One method required by the `Collection<E>` interface has this signature:

```
Iterator<E> iterator()
```

• To understand this operation, you have to understand what an `Iterator` is, and the `Iterator Software Design Pattern`
The Iterator Software Design Pattern

• A common situation: A client needs to inspect the data elements in a collection, without wanting to know details of how the collection structures its data internally

• Solution:
  – Define an interface that specifies how an iterator will behave
  – Design the collection to be able to supply an object that implements that iterator interface
  – A client then can ask the collection for an iterator object, and use that iterator to inspect the collection’s elements, without having to know how the collection is implemented
Iterator Behavior

1. client asks the collection for an iterator object

2. client uses the iterator object to iterate through the elements in the collection

There can be multiple iterators simultaneously active on a collection, and this raises some potential problems…
Iterable<E> Interface

- The Collection<E> interface extends the Iterable<E> interface, which is defined as follows:

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

- So any class that implements Collection<E> must define an instance method `iterator()` that returns an Iterator<E> object for that instance

- And Iterator<E> is also an interface in the JCF…
Iterator</E> Interface

- The Iterator</E> interface is defined as follows:

```java
public interface Iterator<E> {
    public E next();
    public boolean hasNext();
    public void remove();
}
```

- So, any object that is-a Iterator</E> will have those operations as part of its API.

- But what are these methods supposed to do? Let’s look at the documentation, and some examples
Iterator<E> method descriptions

- boolean hasNext()
  
  **Returns:** true if the iteration has more elements. (In other words, returns true if next would return an element rather than throwing an exception.)

- E next()
  
  **Returns:** the next element in the iteration.

  **Throws:**
  
  **NoSuchElementException** - if iteration has no more elements.
Iterator<E> method descriptions, cont’d

• void **remove**()
  Removes from the underlying collection the last element returned by the iterator (optional operation).
  This method can be called only once per call to next().
  The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

**Throws:**

  **UnsupportedOperationException** - if the remove operation is not supported by this Iterator.

  **IllegalStateException** - if the next method has not yet been called, or the remove method has already been called after the last call to the next method.
Using Iterators in Java

• A typical use of an `Iterator` for a `Collection<MyObjs> c` would be in a context like:

```java
Iterator<MyObjs> i = c.iterator();

while ( i.hasNext() ) {
    MyObjs x = i.next(); // get the next data item in c
    if ( passesTest(x) ) process(x); // and process it
}

• (Here, `c` is referred to as the “backing collection” or “backing structure” of the Iterator `i`)
```
Using Iterators in Java 5

- Since Java 5, a new ‘foreach’ style syntax is available to even more conveniently write an iteration over the elements of any object (such as any Collection) that implements the Iterable interface.

- Again assuming that $c$ is a `Collection<MyObjs>`:

```java
for ( MyObjs x: c ) {
    if ( passesTest(x) ) {
        process(x);
    }
}
```
Iterating over a LinkedList

• As an example of some Iterator behavior, let’s create a LinkedList object, add some elements to it, obtain an Iterator that has the LinkedList as its backing collection, and then use the Iterator

• Recall: LinkedList is a class in the JCF; it implements the List interface, which extends the Collection interface… so every LinkedList is-a Collection

• Recall: the one-argument add() method of LinkedList adds the data element to the end of the list

• Note: an Iterator for a LinkedList will iterate over the elements of the LinkedList, starting at the beginning and going to the end
An Iterator in action

Collection<Integer> x = new LinkedList<Integer>();
x.add(15); x.add(10); x.add(40); x.add(77);

Iterator<Integer> it = x.iterator();

System.out.print(it.next());
System.out.print(it.next());
it.remove();
System.out.print(it.next());
System.out.print(it.hasNext());
Next time

- Algorithm costs: time and space
- Best case, worst case, average case analysis
- Counting instructions and asymptotic analysis
- Big-O, big-Omega, big-Theta notation
- Introduction to algorithm measurement

Reading: Gray, Ch 2 and Ch 5