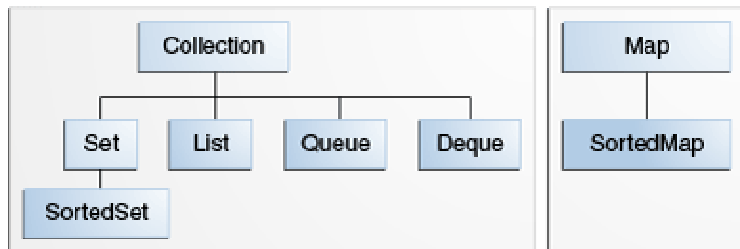
Introduction to Object-Oriented Programming

Iterators

Christopher Simpkins
chris.simpkins@gatech.edu

The Collections Framework



- A *collection* is an object that represents a group of objects.
- The collections framework allows different kinds of collections to be dealt with in an implementation-independent manner.

Collection Framework Components

The Java collections framework consists of:

- Collection interfaces representing different types of collections (sets, lists, etc)
- General purpose implementations (like `ArrayList` or `HashSet`)
- Absract implementations to support custom implementations
- Algorithms defined in static utility methods that operate on collections (like `Collections.sort(List<T> list)`)
- **Infrastructure interfaces that support collections (like `Iterator`)**

Today we'll learn a few basic concepts, then tour the collections library.

The `Collection` Interface

`Collection` is the root interface of the collections framework, declaring basic operations such as:

- `add(E e)` to add elements to the collection
- `contains(Object key)` to determine whether the collection contains `key`
- `isEmpty()` to test the collection for emptiness
- `iterator()` **to get an interator over the elements of the collection**
- `remove(Object o)` to remove a single instance of `o` from the collection, if present
- `size()` to find out the number of elements in the collection

None of the collection implementations in the Java library implement `Collection` directly. Instead they implement `List` or `Set`.

Iterators

Iterators are objects that provide access to the elements in a collection. In Java iterators are represented by the `Iterator` interface, which contains three methods:

- `hasNext()` returns true if the iteration has more elements.
- `next()` returns the next element in the iteration.
- `remove()` removes from the underlying collection the last element returned by the iterator (optional operation).

The most basic and common use of an iterator is to traverse a collection (visit all the elements in a collection):

```
ArrayList tasks = new ArrayList();
// ...
Iterator tasksIter = tasks.iterator();
while (tasksIter.hasNext()) {
    Object task = tasksIter.next();
    System.out.println(task);
}
```

See ArrayListBasics.java for examples.

The Iterable Interface

An instance of a class that implements the `Iterable` interface can be the target of a for-each loop. The `Iterable` interface has one abstract method, `iterator`:

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

Let's see how we can implement an iterator for DynamicArray.java

DynamicArray

DynamicArray.java is like an `ArrayList`

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        ???
    }
    private Object[] elements;
    private int lastIndex;

    public DynamicArray() { this(10); }
    public DynamicArray(int capacity) { ... }
    public Iterator<E> iterator() {return new DynamicArrayIterator();}
    public void add(E item) { ... }
    public E get(int index) { ... }
    public void set(int index, E item) { ... }
    public int size() { ... }
    public E remove(int index) { ... }
}
```

Assuming the methods above are defined, how do we write `DynamicArrayIterator`?

DynamicArrayIterator

The key component of an iterator is a *cursor*: a pointer to the next element in the collection.

- Since `DynamicArray` uses an array as its backing data store, the cursor is simply an index into this array
- The first element to be accessed is at index 0

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;

        public boolean hasNext() {
            return cursor <= lastIndex;
        }
        public E next() {
            if (!hasNext()) { throw new NoSuchElementException(); }
            E answer = get(cursor++);
            return answer;
        }
        public void remove() {
            DynamicArray.this.remove(cursor - 1);
        }
    }
```

DynamicArrayIterator's `next` Method

An `Iterator`'s `next` method

- returns the element the cursor currently points to, and
- moves the cursor to the next element in the collection

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;

        public boolean hasNext() { ... }
        public E next() {
            if (!hasNext()) { throw new NoSuchElementException(); }
            E answer = get(cursor++);
            return answer;
        }
        public void remove() { ... }
    }
    private Object[] elements;
    private int lastIndex;
```

DynamicArrayIterator's `hasNext` Method

An `Iterator`'s `hasNext` method

- is used by clients of the `Iterator` to determine whether unvisited elements of the collection remain
- for `DynamicArray` we simply test whether the cursor is still a valid array index

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;

        public boolean hasNext() {
            return cursor <= lastIndex;
        }
        public E next() {
            if (!hasNext()) { throw new NoSuchElementException(); }
            E answer = get(cursor++);
            return answer;
        }
        public void remove() { ... }
    }
    private Object[] elements;
```

DynamicArrayIterator's `remove` Method

- removes the last element returned by the iterator
- the only safe way to modify a collection being iterated over

We simply use the `DynamicArray`'s '`remove` method

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;
        public boolean hasNext() { return cursor <= lastIndex; }
        public E next() {
            if (!hasNext()) { throw new NoSuchElementException(); }
            E answer = get(cursor++);
            return answer;
        }
        public void remove() {
            DynamicArray.this.remove(cursor - 1);
        }
    }
```

Notice the syntax for distinguishing between the enclosing class's `remove` method and the inner class's `remove` method.

What if we called the inner class's `remove` method recursively?

The Iterable Interface and the For-Each Loop

An instance of a class that implements `Iterable` can be the target of a for-each loop.

```
DynamicArray<String> da = new DynamicArray<>(2);
da.add("Stan");
da.add("Kenny");
da.add("Cartman");
System.out.println("da contents:");
for (String e: da) {
    System.out.println(e);
}
```

See DynamicArray.java for implementation details.