Java Iterators

Motivation

- We often want to access every item in a collection of items
  - We call this *traversing* or *iterating*
  - Example: array
    ```
    for (int i = 0; i < array.length; i++)
        /* do something with array[i] */
    ```
  - Easy because we know exactly how an array works!

Motivation

- What if we want to traverse an arbitrary collection of objects?
  - Its underlying implementation may not be known to us
- Java provides an interface for stepping through all elements in *any* collection, called an *iterator*

Iterating through an ArrayList

- Iterating through an ArrayList of Strings:
  ```
  ArrayList<String> list = ...
  for (int i = 0; i < list.size(); i++) {
      String s = list.get(i);
      //do something with s
  }
  ```
- Alternative:
  ```
  Iterator<String> itr = list.iterator();
  while (itr.hasNext()) {
      String s = itr.next();
  }
  ```
  This syntax of iteration is generic and applies to any Java class that implements the Iterator interface.

Iterating through an ArrayList

- Iterating through an ArrayList of Strings:

```
ArrayList<String> list = ...
for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    //do something with s
}
```

- Alternative:

```
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    String s = itr.next();
}
```

Advantage of the alternative: the code will work even if we decide to store the data in a different data structure (as long as it provides an iterator)

Iterating through an ArrayList

- Iterating through an ArrayList of Strings:

```
LinkedList<String> list = ...
for (int i = 0; i < list.size(); i++) {
    String s = list.get(i);
    //do something with s
}
```

- Alternative:

```
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    String s = itr.next();
}
```

This syntax of iteration is generic and applies to any Java class that implements the Iterator interface.

The Java **Iterator** Interface

- Iterator<T>: a generic interface with the following methods
  - **public boolean hasNext();**
    returns true if there are more elements to iterate over
  - **public T next();**
    returns the next element
  - **public void remove();**
    removes the last element returned by the iterator (*optional operation*)
- It is in the java.util package
- Which Java class that you know implement this interface?

The Java **Iterator** Interface

- **public boolean hasNext();**
  returns true if there are more elements to iterate over
- **public T next();**
  returns the next element

  throws a NoSuchElement exception if a next element does not exist
- **public void remove();**
  removes the last element returned by the iterator

  optional operation: if you choose not to implement it, the method needs to throw an UnsupportedOperationException

The Java **Iterator** Interface

```java
public interface Iterator<E> {
   // Returns the next element. Throws a NoSuchElementException
   // if there is no next element.
   public E next();

   // Returns true if there is a next element to return.
   public boolean hasNext();

   // Removes the last element that was returned by next.
   // Throws an UnsupportedOperationException if the remove
   // method is not supported by this Iterator. Throws an
   // IllegalStateException if the next method has not yet been
   // called or if the remove method has already been called
   // after the last call to the next method.
   public void remove();
}
```

Using an iterator

```java
Integer[] array = {1, 10, 100};  // not int[]
ArrayIterator<Integer> itr = new ArrayIterator<Integer>(array);
while (itr.hasNext()){
   Integer element = itr.next();
}
```

Example: an array iterator

```java
public class ArrayIterator<T> implements Iterator<T> {
   private int current;
   private T[] array;
   public ArrayIterator (T [] array){
      this.array = array;
      this.current = 0;
   }
   public boolean hasNext(){
      return (current < array.length);
   }
   public T next(){
      if (!hasNext())
         throw new NoSuchElementException();
      current++;
      return array[current - 1];
   }
}
```

Example: an array iterator

```java
public class ArrayIterator<T> implements Iterator<T> {
   private int current;
   private T[] array;
   public ArrayIterator (T [] array){
      this.array = array;
      this.current = 0;
   }
   public boolean hasNext(){
      return (current < array.length);
   }
   public T next(){
      if (!hasNext())
         throw new NoSuchElementException();
      current++;
      return array[current - 1];
   }
}
```

Forgetting
Something?

(go back
3 slides)

The Iterable interface

Given an ArrayList we can traverse it using an iterator:

```
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    String s = itr.next();
}
```

Or using the foreach form of the for loop:

```
for (String s : list) {
    //do something with s
}
```

The latter is possible because an ArrayList is Iterable.

The Iterable interface

- The Java API has a generic interface called `Iterable<T>` that allows an object to be the target of a "foreach" statement
  - `public Iterator<T> iterator();`
    returns an iterator
- Why do we need `Iterable`?
  - An Iterator can only be used once, Iterables can be the subject of "foreach" multiple times.

```
public class MyArrayList<T> implements Iterable<T> {
    private T [] array;
    // not shown:  constructors and add/remove etc.
    public Iterator iterator() {
        Iterator<T> itr = new ArrayIterator();
        return itr;
    }
    private class ArrayIterator implements Iterator<T> {
        int current;
        public ArrayIterator (){
            this.current = 0;
        }
        public boolean hasNext(){
            return (current < array.length);
        }
        public Object next() {
            if (!hasNext())
                throw new NoSuchElementException();
            current++;
            return array[current - 1];
        }
    }
}
```

Inner classes

- **Inner class**: defined inside another class
- If declared private it can't be used by other classes
- The methods of the inner and outer classes have access to each other's methods and instance variables, even if declared private.

Why use Iterators?

- Traversing through the elements of a collection is very common in programming, and iterators provide a *uniform* way of doing so.
- Advantages?
  - Using an iterator, we don't need to know how the data structure is implemented!
  - Can change the data structure declaration without changing any other code!