Recall: Linear Search

- First version:
  - Input was int[], used “==” to compare elements

- More generic version:
  - Input was Comparable[], used compareTo()

- Is there a still more generic version that is independent of the data structure?
  - For example, works even with Comparable[][]

Iterator Interface

- java.util.Iterator

- Linear search can be written once and for all using iterator interface
  - Any data structure that wants to support linear search must implement Iterator
  - We look at three ways to implement Iterator
    - Using a separate class
    - Using an inner class
    - Using an anonymous inner class

Linear Search

```java
boolean linearSearch(Comparable[] a, Object v) {
    for (int i = 0; i < a.length; i++) {
        if (a[i].compareTo(v) == 0) return true;
    }
    return false;
}
```

- Relies on data being stored in a 1D array
  - Will not work if data is stored in another data structure such as a 2D array, list, stack, queue, ...
  - All linear search really needs is:
    - Are there more elements to look at?
    - If so, get me the next element

Generic Linear Search

- Data is contained in some object
- Object has an adapter that permits data to be enumerated in some order
- Adapter has two buttons
  - boolean hasNext(): are there more elements?
  - Object next(): if so, give me a new element that has not been enumerated so far
**Iterator Interface**

- Predefined in Java
  - `java.util.Iterator`
  - `java.util.Iterable`
- Linear search can be written using Iterator interface
- Any class that wishes to allow linear searching can do so by implementing Iterable (i.e., by providing an Iterator)

```java
interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Optional operation
}

interface Iterable {
    Iterator iterator();
}
```

**Enumeration Interface**

- You may see some code that uses the Enumeration interface instead of the Iterator interface
  - Enumeration was part of the earliest versions of Java
  - Similar functionality to Iterator (no `remove` method)
  - Iterator is preferred

```java
interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

**Generic Linear Search**

**Array version**

```java
boolean linearSearch(Object[] a, Object v) {
    for (int i = 0; i < a.length; i++) {
        if (a[i].equals(v)) return true;
    }
    return false;
}
```

**Iterator version**

```java
boolean linearSearch(Iterator a, Object v) {
    while (a.hasNext()) {
        if (a.next().equals(v)) return true;
    }
    return false;
}
```

**How Do We Create an Iterator?**

- Iterator is a Java interface, so we must create a class that implements `Iterator`
- To create an Iterator for class X, we can
  - Use a separate class
  - Use an inner class within X
  - Use an anonymous inner class within X

**An Array Iterator (Version 1)**

```java
class ArrayIterator implements Iterator {
    private Object[] data;
    private int index = 0; // Index of next element

    public ArrayIterator(Object[] a) {
        data = a;
    }

    public boolean hasNext() {
        return (index < data.length);
    }

    public Object next() {
        return data[index++];
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

**Using the ArrayIterator**

```java
String[] a = {"Hello", "world"};
Iterator iter = new ArrayIterator(a);
while (iter.hasNext()) {
    System.out.println(iter.next());
}
int iter = new ArrayIterator(a);
if (linearSearch(iter, "world") {
    System.out.println("found!");
}
```
Iterators Features

- Can create as many iterators as needed
  - Multiple iterators over the same data set are fine (as long as the data set isn’t changed during iteration)

- Works for most data structures
  - Example: 2D arrays
    - Can keep two cursors, one for row, one for column
    - Standard orders of enumeration
      - Row-major
      - Column-major

Sharks and Remoras

- Iterator implementation is like a remora
- Data class is like a shark

A single shark must allow many remoras to hook to it

Client Code

```java
class Array2DIterator implements Iterator {
  private Object[][] data;
  private int rowIndex = 0, colIndex = 0;
  public Array2DIterator(Object[][] a) { data = a; }
  public boolean hasNext() {
    while (rowIndex < data.length
      && colIndex >= data[rowIndex].length) {
      rowIndex++; colIndex = 0; // if end of row
    }
    return (rowIndex < data.length
      && colIndex < data[rowIndex].length);
  }
  public Object next() {
    if (hasNext()) return data[rowIndex][colIndex++];
    else throw new NoSuchElementException();
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

```java
class Shark implements Iterable {
  public Object[] data;
  public Shark(Object[] a) { data = a; }
  public Iterator iterator() {return new Remora(this);}
}
```

```java
class Remora implements Iterator {
  private int index = 0;
  private Shark shark;
  public Remora(Shark s) { shark = s; }
  public boolean hasNext() { return (index < shark.data.length); }
  public Object next() { return shark.data[index++]; }
  public void remove() { throw new UnsupportedOperationException(); }
}
```

```
String[] a = {"Hello", "world"};
Shark s = new Shark(a); // object containing data
boolean b = linearSearch(s.iterator(), "Hello");
boolean c = linearSearch(s.iterator(), "world");
boolean d = linearSearch(s.iterator(), "Bye");
```

Critique: Iterator as Separate Class

- Good
  - Shark class focuses on data
  - Remora class focuses on iteration

- Bad
  - Remora code relies on being able to access Shark variables such as data array
    - What if data were declared private?
  - Remora is specialized to Shark, but code appears outside Shark class
    - We may change Shark class and forget to update Remora
  - Clients can create Remoras without invoking iterator() method of Shark
    - Better to have language construct to enforce convention
Better: Iterator as an Inner Class

• Inner class: Java allows you declare a class within another class
• Inner classes can occur at many levels within another class
  • Member level
    • Inner class defined as if it were another field or method
  • Statement level
    • Inner class defined as if it were a statement in a method
  • Expression level
    • Inner class defined as it were part of an expression
    • Such expression-level classes are called anonymous classes

• Initially, we focus on member-level inner classes

Example of an Inner Class

```java
class Shark implements Iterable {
    public Object[] data;
    public Shark(Object[] a) { data = a; }
    public Iterator iterator() {
        return new Remora();
    }

    class Remora implements Iterator {
        private int index = 0;
        public boolean hasNext() { return (index < data.length); }
        public Object next() {
            return data[index++];
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }
}

String[] a = {"Hello", "world"};
Shark s = new Shark(a);
boolean b = linearSearch(s.iterator(), "Hello");
```

Client Code

```java
String[] a = {"Hello", "world"};
Shark s = new Shark(a);
boolean b = linearSearch(s.iterator(), "Hello");
```

Observations

• Inner class can be declared `public`, `private`, "package", or `protected`
  • Inner class name is visible accordingly
• Instances of an inner class have access to all members of containing outer-class instance
  • Even members declared `private`
• Some inner-class syntax is weird
  • Inner classes that are `public` can be instantiated by `outerObjectInstance.new InnerClass()`
  • E.g., `myShark.new Remora()`
  • Note that `new Shark.Remora()` does not work
  • If you find yourself needing this syntax, you are probably using a bad design

Inner Classes & this

• Keyword `this` in Remora class refers to Remora object-instance, not outer Shark object-instance
• How do we get a reference to Shark from Remora?
  • Here's one way:
  ```java
  class Shark {
    private kahuna;
    public Shark() { kahuna = this; }
    class Remora { //inner class
      ...kahuna... //inner class can access variable
    }
  }
  ```
  • Here's another way: `Shark.this` refers to the outer Shark object-instance

Anonymous Classes

• To permit programmers to write inner classes compactly, Java permits programmers to write anonymous classes
  • Class does not have a name
  • Must be instantiated at the point where it is defined

Anonymous Class Example

```java
class Shark implements Iterable {
    public Object[] data;
    public Shark(Object[] a) { data = a; }
    public Iterator iterator() {
        return new Remora();
    }

    class Remora implements Iterator {
        private int index = 0;
        public boolean hasNext() { return (index < data.length); }
        public Object next() {
            return data[index++];
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```
Anonymous Class Example

```java
class Shark implements Iterable {
    public Object[] data;
    public Shark(Object[] a) { data = a; }
    public Iterator iterator() {
        return new Iterator {
            private int index = 0;
            public boolean hasNext() {
                return (index < data.length);
            }
            public Object next() {
                return data[index++];
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Anonymous-Class Properties

- An anonymous class is an inner class with the usual class body, but
  - No class name
  - No access specifier (i.e., no public/private/protected)
  - No constructor
  - No explicit use of extends or implements
    - It either extends one class or implements one interface

```
new classOrInterfaceName {...body...}
```

Anonymous Class Examples

- To specify an anonymous class (call it A) that extends class P
  - `new P { ... }`; //create instance of A
  - `new P(42) { ... }`; //calls different P-constructor
  - `P x = new P { ... };`; //assignment

- To specify an anonymous class (call it A) that implements interface I
  - `new I { ... }`; //create instance of A
  - `I y = new I { ... };`; //assignment

- Anonymous class can override methods of superclass P or implement interface methods of I
- All other methods and fields are effectively private
  - Because there is no way to invoke them from outside!

Enhanced for-loop

- As of Java 5, a for-loop works with
  - Any array type
  - Anything that implements the Iterable interface

**Iterator version**

```java
boolean linearSearch(Iterator a, Object v) {
    while (a.hasNext()) {
        if (a.next().equals(v)) return true;
    }
    return false;
}
```

**Iterable version**

```java
boolean linearSearch(Iterable b, Object v) {
    for (Object x: b) {
        if (x.equals(v)) return true;
    }
    return false;
}
```

Conclusions

- Iterator interface allows one to write generic code
  - Works on data collections without regard to type of elements or data structure

- Inner classes are the best way to write an Iterator

- The for-each construct (i.e., enhanced for-loop) makes for more compact code, but
  - Cannot use if need access to array indices, for instance
  - Cannot use if need to use remove-operation of Iterator