Generic Programming and Inner Classes

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[][]`

Key Ideas

- `Iterator` interface
- Linear search written once and for all using `Iterator` interface
- Any data structure that wants to support linear search must implement `Iterator`
- Implementing `Iterator` interface
  - We will look at three implementations
  - Anonymous inner classes provide an elegant solution

Linear Search

```
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, ...

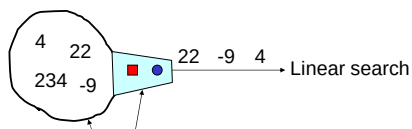
Linear Search

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

All linear search needs to know is:

1. are there more elements to look at?
2. 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 to be enumerated?
  - `Object next()`: if so, give me a new element that has not been enumerated so far

Iterator Interface

```
interface Iterator {
   boolean hasNext();
   Object next();
   void remove(); //we will not use this
}
```

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

- predefined in Java
- linear search can be written using **`Iterator`** interface
- any data class that wishes to allow searching using this code can do so by implementing **`Iterable`** (i.e., by providing an **`Iterator`**

Enumeration Interface

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

- similar functionality to **`Iterator`** (no **`remove`** method)
- **`Iterator`** is preferred

Generic Linear Search

Array version

```
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

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

How Do You Produce an Iterator?

Some possibilities:

1. Adapter is a separate class from the data class
2. Adapter is an inner class of the data class
3. Adapter is an anonymous inner class

Adapter (Version 1)

```
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++];
   }
}
```

Using the Adapter

```
String[] a = {"Hello", "world"};

Iterator iter = new ArrayIterator(a);
while (iter.hasNext()) {
  System.out.println(iter.next());
}

iter = new ArrayIterator(a);
if linearSearch(iter,"world") {
  System.out.println("found!");
}
```

Features

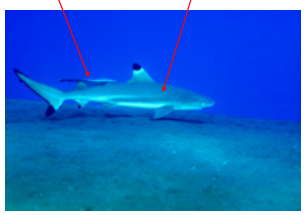
- Can create as many iterators as needed
- Works for other data structures
  - 2D arrays: keep two cursors, one for row, one for column
  - standard orders of enumeration:
    - row-major
    - column-major

```
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();
   }
}
```

Sharks and Remoras

Iterator implementation is like a remora

Data class is like shark



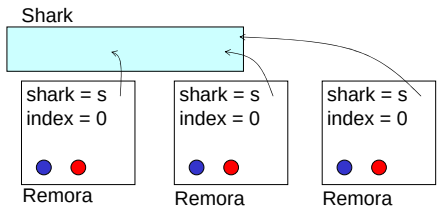
A single shark must allow many remoras to hook to it

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

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++];
  }
}
```

Remora teeth

Client Code

```
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

- Good:
  - Shark class focuses on data, Remora class focuses on enumeration
- Bad:
  - Remora code relies on being able to access Shark variables such as **`data`** array
    - What if **`data`** was declared private?
  - Remora is specialized to Shark, but code appears outside Shark class
    - 2D array Shark will require a different Remora
    - 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: Inner Classes

- 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
    - Called anonymous classes
- Let us focus on member-level inner classes

Example of an Inner Class

```
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++];
    }
  }
}
```

Client Code

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

Observations

- Inner class can be declared public, private, or protected
  - Inner class name is visible accordingly
- Inner class can also be instantiated by **`outerObject.new InnerClass()`**
  - e.g., **`Shark.new Remora()`**
  - but **`new Shark.Remora()`** does not work
- Instances of inner class have access to all members of containing outer class instance, even if declared private

- Keyword **`this`** in Remora class refers to Remora object, not outer Shark object
- How do we get a reference to Shark object from Remora? Here's one way:

```
class Shark {
  private kahuna;
  public Shark() { kahuna = this; }

  class Remora{ //inner class
    ...kahuna... //inner class simply accesses variable
  }
}
```

Adapter Classes

- An inner class is like an adapter that permits client code to work with class containing data without modifying the data class itself
- This is a very general design pattern that shows up in many contexts (e.g., GUI's)
- To permit programmers to write adapters 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 Classes

```
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++];
    }
  }
}
```

Anonymous Classes

```
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++];
    }
  };
}
```

Anonymous Classes

```
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++];
      }
    };
}
```

Anonymous Classes

- Class declaration has usual body, but
  - inner class
  - no name
  - no access specifier: public/private/protected
  - no explicit extends or implements
    - it either extends one class or implements one interface
  - no constructor

Creating an Instance of Anonymous Class A

- To specify that A extends superclass P
  - `new P { ... }; //creates instance of A`
  - `new P(42) { ... }; //calls a different constructor of P`
  - `P x = new P { ... }; //assignment`
- To specify that A implements interface I
  - `new I { ... }`
  - `I foo = 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
  - No way to invoke them from outside!

Conclusions

- Generic code
  - works on data collections without regard to type of elements or data structure
- Writing generic code
  - **`Iterator`** interface is very useful
  - use inner classes to implement it