Generic Programming and Inner classes

Goal

- First version of linear search
  - Input was array of int
- More generic version of linear search
  - Input was array of Comparable
- Can we write a still more generic version of linear search that is independent of data structure?
  - For example, work even with 2-D arrays of Comparable, or List, or ArrayList, etc.

"Obvious" linear search code

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

Code in red relies on data being stored in a 1-D array.
For-loop also implicitly assumes that data is stored in 1-D array.

This code will not work if data is stored in a more general data structure such as a 2-D array.

Minor rewrite of linear search

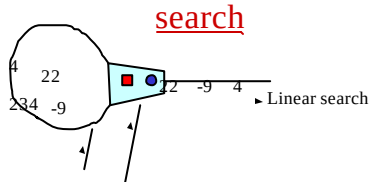
```
boolean linearSearch (Comparable[] a, Object v) {
  int i = 0;
  while (i < a.length) {
    if (a[i].compareTo(v) = = 0) return true;
    else i++;
  }
  return false;
}
```

Intuitively, linear search needs to know
- are there more elements to look at?
- if so, get me the next element

Key ideas in solution

- Iterator interface
- Linear search written once and for all using Iterator interface
- Data class that wants to support linear search must implement Iterator interface
- Implementing Iterator interface
  - We look at several approaches

Intuitive idea of 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
}
```

This interface is predefined in Java.
Linear search is written using this interface.
Data class must provide an implementation of this interface.

Generic Linear Search

Iterator version

```
boolean linearSearch(Iterator a, Object v) {
    while (a.hasNext())
        if ((Comparable) a.next()).compareTo(v) == 0)
            return true;
    return false;
}
```

Compare with Array version

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


How does data class implement Iterator interface?

Let us look at a number of solutions.

1. Adapter code is part of data class
2. Taking adapter code out of the data class
3. a – d : Putting it back in

Adapter (version 1)

```
class Crock1 implements Iterator {
        private Comparable[ ] a;
        private int cursor = 0; //index of next element to be enumerated

        public Crock1(…) { …store data in array a… }

        public boolean hasNext( ) {
                return (cursor < a.length);
        }

        public Object next( ) {
                // may throw ArrayIndexOutOfBoundsException
                return a[cursor++];
        }

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


Critique

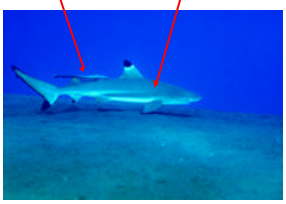
- As shown, client class can only enumerate elements once!
  - No way to reset the cursor
- Making the data class implement Iterator directly is something of a crock because its concern should be with data, rather than enumeration of data.
- However, this works for other data structures such as 2-D arrays.
  - 2-D arrays: data class can keep two cursors
    - one for row
    - one for column
    - standard orders of enumeration: row-major/column-major

- One solution to resetting the cursor:
  - Data class implement a method `void reset()` which resets all internal cursor(s)
  - Method must be declared in Iterator interface
- But we still cannot have multiple enumerations of elements going on at the same time
  - Remember: only one cursor per data array…
- Problem: cannot create new cursors on demand
- To solve this problem, cursor must be part of a different class that can be instantiated any number of times for a single data object.

Sharks and remoras

Iterator implementation is like a remora.

Data class is like shark



Single shark must allow us to hook many remoras to it.

Adapter (version 2)

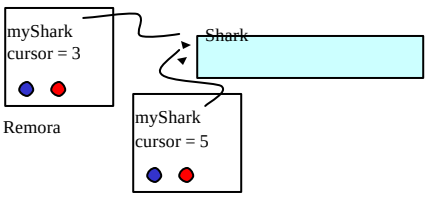
```
class Shark {
        protected Comparable[] a;
        public Shark(…) {…get data into a…}
}
```

```
class Remora implements Iterator {
    private Shark myShark;
    private int cursor = 0;

    public Remora(Shark s) { myShark = s; }

    public boolean hasNext() {
        // a in Shark is protected, so accessible
        return (cursor < myShark.a.length);
    }

    public Object next( ) { return myShark.a[cursor++]; }

    public void remove( ) {…}
}
```

Remora teeth

Client code:

```
Shark s = new Shark(…data…);
Remora r1 = new Remora(s);
Remora r2 = new Remora(s);
linearSearch(r1, v);
```



Remora

Remora

Critique

- Good:
  - Shark class focuses on data, Remora class focuses on enumeration
- Bad:
  - Remora code relies on being able to access Shark variables (e.g. array a must be made *protected*)
    - What if a was declared private?
    - Protected access is less secure than private.
  - Remora is specialized to Shark but code appears outside Shark class
    - 2-D array Shark will require a different Remora
    - We may change Shark class and forget to update Remora.

Slightly better code: Shark object creates Remoras in request

```
class Shark {
    protected Comparable[] a;
    public Shark(…) {…get data into a…}

    public Iterator makeRemora( ) {
        // Shark code mentions Remora class
        return new Remora(this);
    }
}
```

```
class Remora implements Iterator {
  private Shark myShark;
  …
}
```

```
// client code

Shark s = new Shark(…data…);

// let shark make a Remora
Iterator r1 = s.makeRemora( );

// or make our own
Remora r2 = new Remora(s);

linearSearch(r1, v);
```

see iterator-2-outside files

Critique

- Good:
  - Shark code mentions Remora, so person modifying Shark code is at least aware that Remora code depends on this class.
- Bad:
  - Clients can still create Remoras without invoking makeRemora method
    - Better to have language construct to enforce such a convention
    - Better to force them to use Iterator, not Remora
  - Code in separate files, but closely intertwined:
    - Remora used in exactly one place: inside Shark.makeRemora()

Idea: Move Remora Closer

- Want Remora to be as close as possible to the one single place that uses it.
- What does this mean?
  - In same file?
  - In same class?
  - In same method?
  - On same line?

19

An Old Idea: In Same File

- Problem: Compiler prefers to have one class per file
  - How could it know where to find Remora code during compilation?

20

Better Idea: In Same Class

- Let's make Remora class a "member" of Shark class
  - Should it be static or not?
  - Should it be public or not?

21

Make it Static: Nested Classes

```
class Shark {
  private Comparable[ ] a;
  public Iterator makeRemora( ) {
     return new Remora(this);
  }

  public static class Remora  //nested class
  implements Iterator {
     Shark myShark;
     int cursor = 0;
     // same code, but has access
     // to private member myShark.a
     public Object next( ) {
        return  myShark.a[cursor++];
     }
  }
}
```

```
// client code

Shark s = new Shark(…data…);

// let shark make it for us
Iterator r2 = s.makeRemora( );

// make our own Remora
Remora r1 = new Shark.Remora();

linearSearch(r2, v);
```

see iterator-3a-nested files

22

Critique

- Good:
  - Shark code and remora code in the same file
    - Class is Shark.Remora so compiler knows where to find it.
  - Users know that Remora is specific to Shark class.
  - Can now have Dolphin.Remora, Whale.Remora, etc…
  - Data is private again
- Bad:
  - Clients can still create Remoras on their own
  - Each Remora "belongs" to exactly one Shark
    - Java language could enforce this
  - Remora code must keep writing " myShark.blah"
    - It could be implicit, since *everything* in Remora refers to the Shark

23

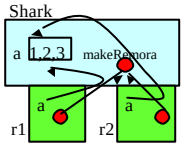
Make it Non-Static: Inner Classes

```
class Shark {
  private Comparable[ ] a;
  public Iterator makeRemora( ) {
     return new Remora();
  }

  public class Remora  // inner class
  implements Iterator {
     int cursor = 0;
     // same code, but "myShark" is implicit
     public Remora( ) { }
     public Object next( ) {
        return  a[cursor++];
     }
  }
}
```

```
// client code

Shark s = new Shark(…data…);

// let shark make it for us
Iterator r2 = s.makeRemora( );

// make our own Remora
Remora r1 = s.new Remora();

linearSearch(r2, v);
```

see iterator-3b-inner files

24



Points to note

- Nested & Inner class can be declared to be public, private, or protected
  - class name is visible accordingly
- Nested class instantiated: new Shark.Remora(…);
  - class Remora is a static member of Shark
- Inner class is instantiated: s.new Remora(…);
  - class Remora is an instance member of Shark
- Instances of inner class is "bound" to an instance of the class that contains it.
  - the " myShark" is implicit
  - any member not in the inner class is assumed to be in the containing class

What is *this* in Inner Class?

- Keyword this in Remora class refers to Remora object, not the outer Shark object.
- How do we get a reference to Shark object?
  - Alternative 1: Go back to using "myShark" in each Remora
  - Alternative 2: Put a "self" member in Shark

```
class Shark {
    private self;
    public Shark(…) {
        self = this; …
    }
    class Remora {  // inner class
        void debug( ) {
            System.out.println(this);
            System.out.println(self);
        }
}
```


Critique

- Good:
  - Remora instance bound to a single shark instance.
    - No need for " myShark" nonsense
- Bad (?):
  - Clients can still create Remoras on their own (if public)
  - Remora code is still kind of far away from the only place it is every used

Closer: Method-Local Classes

```
class Shark {
  private Comparable[ ] a;
  public Iterator makeRemora( ) {
    class Remora  // method-local class
    implements Iterator {
      // same code
      public Object next( ) {
        return  a[cursor++];
      }
    }
    return new Remora( );
  }

}
```

```
// client code

Shark s = new Shark(…data…);

// let shark make it for us
Iterator r2 = s.makeRemora( );

// can't make our own Remora
//Remora r1 = new ???

linearSearch(r2, v);
```

see iterator-3c-local files

Critique

- Good:
  - Nothing outside makeRemora can see the class Remora anymore
- Bad (?):
  - Remora name is defined, then used once
    - Like saying " int a = 3; return a;"
    - Instead of just "return 3;"

Closer: Anonymous Classes

```
class Shark {
  private Comparable[ ] a;
  public Iterator makeIterator( ) {
    return new Iterator( ) {
      // same code, but no constructor
      public Object next( ) {
        return a[cursor++];
      }
    }
  }

}
```

```
// client code

Shark s = new Shark(…data…);

// let shark make it for us
Iterator r2 = s.makeIterator( );

// can't make our own Remora
//Remora r1 = new ???

linearSearch(r2, v);
```

see iterator-3d-anonymous files

Critique

- Good:
  - Iterator class (previously known as Remora) defined on-the-fly, on precisely the line that it is needed.
    - No more "Remora" name any more…
- Bad:
  - Confusing as anything
  - Wacko syntax

Anonymous classes

- Class declaration has usual body but
  - becomes 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
  - For "extends" case, you can call super-class constructor of your choice:
    - return new Point(2, 3) { … override Point methods here … }
    - Overriding methods makes sense… but how would you call a newly defined method?

Adapter classes

- 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.
  - Frequently uses anonymous classes (defined at the point they are used).

Conclusions

- Generic code:
  - works on data collections without much regard to type of data elements or type of data structure
- Writing generic code:
  - Iterator interface is very useful
  - use inner classes to implement Iterator
- C++ Standard Template Library:
  - more complex iterators