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

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 three solutions
  - Inner classes provide elegant solution

Recall linear search code

```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;
}
```

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

```java
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

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**

```java
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**

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

**Compare with Array version**

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

**Adapter (version 1)**

```java
class Crock1 implements Iterator {
    protected Comparable[] a;
    protected 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() {
        return a[cursor++];
    }
    public void remove() {} //unimplemented
}
```

**Critique**

- As shown, client class can only enumerate elements once!
  - How do we 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….
- 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 a 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{
    int cursor;
    Shark myShark;
    public Remora(Shark s) {
        myShark = s;
        cursor = 0;
    }
    public boolean hasNext() {
        return (cursor < myShark.a.length); // a in Shark is protected, so accessible
    }
    public Object next() {
        return myShark.a[cursor++];
    }
    public void remove() {} // unimplemented
}

Client code:

...Shark s = new Shark(); // object containing data
...new Remora(s)... Object v = ...;
boolean b = linearSearch( new Remora(s), v);
...

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 array a
    • 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(){
        return new Remora(this); // Shark code contains mention of Remora class
    }
}
class Remora implements Iterator{
    int cursor;
    Shark myShark;
    public Remora(Shark s) {
        myShark = s;
        cursor = 0;
    }
    public boolean hasNext() {
        return (cursor < myShark.a.length); // a in Shark is protected, so accessible
    }
    public Object next() {
        return myShark.a[cursor++];
    }
    public void remove() {} // unimplemented
}

Client code

...Shark s = new Shark(); // object containing data
...s.makeRemora()...
Object v = ...;
boolean b = linearSearch(s.makeRemora(), v);
...
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 solution: 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 inner class

class Shark{
    private int i;
    public Shark(int arg){
        i = arg;
    }
    public Remora makeRemora(){
        return new Remora();
    }
    public class Remora{
        public void see(){
            System.out.println(i);//inner class has access to i
        }
    }
}
class Client{
    public static void main(String[] args){
        Shark jaws1 = new Shark(7);
        Shark jaws2 = new Shark(-90);
        Shark.Remora r1 = jaws1.makeRemora();//create instance of inner class
        Shark.Remora r2 = jaws2.new Remora();//alternate syntax
        r1.see();//should print 7
        r2.see();//should print -90
        jaws1.makeRemora().see();//should print 7
    }
}

Points to note

• Inner class can be declared to be public, private, or protected
  – Inner class name is visible accordingly
• Inner class is instantiated by invoking method of containing class or by outerObj.new InnerClass()
  – new jaws1.Remora() does not work
• Instances of inner class have access to all members of containing outer class instance
  – In our example, member i of jaws1 is visible to r1 even though it is private

Keyword this in Remora class refers to Remora object, not the 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;//constructor of outer object initializes variable
        ……;
    }
}
class Remora {//inner class
    … kahuna…}//inner class simply accesses variable
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.
  - Adapter class
  - 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 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

Client code: same as before

Shark s = new Shark(); //object containing data
Object v = ...
boolean b = linearSearch(s.makeRemora(), v);

Back to linear search: Data class with inner class

class Shark{
    protected Comparable[] a;
    public Shark() {...get data into a...}
    protected class Remora implements Iterator{
        int cursor = 0;
        public boolean hasNext() {
            return (cursor < a.length);
        }
        public Object next() {
            return a[cursor++];
        }
        public void remove() {} //unimplemented
    }
}

Intuitive idea

import java.util.*;
class Shark{
    private int i;
    public Shark(int arg){
        i = arg;
    }
    public Remora makeRemora(){
        return new Remora();
    }
    //inner class
    public class Remora{
        public void see(){
            System.out.println(i);//inner class has access to i
        }
    }
}
class Client{
    public static void main(String[] args){
        Shark jaws1 = new Shark(7);
        Shark jaws2 = new Shark(-90);
        Shark.Remora r1 = jaws1.makeRemora();//create instance of inner class
        Shark.Remora r2 = jaws2.makeRemora();//alternate syntax
        r1.see();//should print 7
        r2.see();//should print -90
        jaws1.makeRemora().see();//should print 7
    }
}

Anonymous classes

- Creating an instance of anonymous class A:
  - If class A is extending a superclass P
    - new P (body of A); //creates instance of anon class
      • Can invoke appropriate constructor of P by passing arguments to P as in new P(79) (body of A);
    - Assignment: P x = new P (body of A);
  - Assignment: P x = new P (body of A);
  - Assignment: I foo = new I (body of A);
    - Think: anonymous class should only implement interface methods, and not any other methods.
  - If it did, how would you invoke these methods?
    - Something like x.coolMethod(); ???
  - What would the type checker do??
Anonymous class

interface IRemora

void see();
}

class Shark

private int i;
public Shark(int arg)
    i = arg;

public IRemora makeRemora()
    return new IRemora()
    public void see()
        System.out.println(i);

}

class Client

public static void main(String[] args)
    Shark jaws1 = new Shark(7);
    Shark jaws2 = new Shark(90);
    IRemora r1 = jaws1.makeRemora();
    IRemora r2 = jaws2.makeRemora();
    r1.see(); //should print 7
    r2.see(); //should print 90
    jaws1.makeRemora().see(); //should print 7

}

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