Programming Languages and Techniques (CIS120e)

Lecture 27
Nov. 13, 2010

Random-Access Data

Announcements

• Homework 8 due today
• Homework 9 available on Wednesday (due next Wednesday, Nov 24)
• Final homework assignment out next Wednesday (due Dec 10)
• Final exam: Thursday, December 16th, noon - 2PM (Moore 212)

• There will be class next Wednesday (day before Thanksgiving)

Midterm Results

Digression: Generic Methods
Generic Interfaces (review)

```java
class LinkedList<X> {
    private X myField;
    GenericClass(X init) {
        myField = init;
    }
    public X m() {
        return myField;
    }
}

class LinkedList<X> {
    private X myField;
    GenericClass(X init) {
        myField = init;
    }
    public X m() {
        return myField;
    }
}
```

Generic Classes (review)

```java
public interface List<E> {
    void add(E x);
    E next();
    boolean hasNext();
    // ...
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
    // ...
}
```

Generic Methods

```java
class GenericMethod {
    public static <X> X first(List<X> myList) {
        return myList.get(0);
    }
}
```

List<String> myList = new LinkedList<String>();
myList.add("foo");

Random-Access Data

```java
List<String> myList = new LinkedList<String>();
myList.add("foo");
GenericMethod.first(myList);
```
A Taxonomy of Collections

- **Sequential access** (lists, stacks, queues)
  - retrieve elements *in order*
- **Associative access** (sets, maps)
  - retrieve elements by *content*
- **Random access** (vectors, arrays)
  - retrieve elements by *index*
  - reach any element in *constant time*
  - useful for data that is naturally organized by some numerical index (e.g., histograms, spatial data, maps, images)

Random-Access Data Structures in Java

- **arrays**
  - very fast access
  - number of elements must be known in advance; cannot be resized later
- **Vector**
  - not quite as fast
  - but dynamically resizable
- **ArrayList**
  - similar to Vector
  - a little faster, but not thread-safe
  - (you can ignore this one for now)
- **BitSet**
  - special “packed” implementation of a vector of booleans
  - same operations, but uses much less memory
- **String**
  - read-only array of chars
- **...**

Vector Class and Friends

- Interface
  - Collection
    - List
      - LinkedList
    - Set
      - Vector
    - SortedSet
      - ArrayList

(The get and set operations are available on all Lists, but are inefficient on instances of LinkedList.)

JAVA ARRAYS
Java Arrays: Indexing

- Index elements from 0

- \texttt{a[i]} is the \texttt{i}th element of array \texttt{a}
- \texttt{a[i] = e} assigns \texttt{e} to \texttt{i}th element of array \texttt{a}
- \texttt{a.length} is the number of elements in \texttt{a}

Java Arrays: Creation

- Create an array \texttt{a} of size \texttt{n} with elements of type \texttt{type}:

\[
\text{type[]} \ a = \text{new type}[n];
\]

Java Arrays: Initialization

\begin{verbatim}
int[] myArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
String[] yourArray = {"foo", "bar", "baz"};
Point[] herArray = {new Point(1,3), new Point(5,4)};
\end{verbatim}

Java Arrays: Aliasing

- Arrays live in the \textit{heap}
- Variables of array type are \textit{pointers}
ITERATION

Array Iterators

- The class Arrays provides a static method `asList` for converting an array to a List:

  ```java
  static <T> List<T> asList(T[] a)
  ```

- This makes it easy to build an iterator over the elements of an array...

  ```java
  String[] arr = {"foo", "bar"};
  Iterator<String> i = Arrays.asList(arr).iterator();
  ```

For-Each over Arrays

The “for each” looping construct actually works with *either* an `Iterable`...

```java
static int totalLength(String[] a) {
    int total = 0;
    for (String s : Arrays.asList(a))
        total += s.length();
    return total;
}
```

or an array...

```java
static int totalLength(String[] a) {
    int total = 0;
    for (String s : a) total += s.length();
    return total;
}
```

Limitations of “For Each” with Arrays

- the “for each” idiom is good when we need to do something with all of the values in the array (like add them up or find their max)
- but...
  - doesn’t give us access to the array itself
    - no way to modify the elements as we see them
  - no easy way to process just part of the array
  - no way to process elements in a particular order (except left-to-right)
### For loops

A lower-level control structure for iteration...

```
for (int i = 0; i < a.length; i++)
    total += a[i];
```

### An Updating For Loop

```
static void clip (double[] a) {
    for (int i = 0; i < a.length; i++)
        if (a[i] < 0.0)
            a[i] = 0.0;
}
```

### Multi-Dimensional Arrays

A 2-d array is just an array of arrays...

```
String[][] names = {{"Mr. ", "Mrs. ", "Ms. 
"},
                   {"Smith", "Jones"});
```

```
int[][] products = new int[5][];
for(int row = 0; row <= 5; row++) {
    products[row] = new int[row+1];
    for(int col = 0; col <= row; col++)
        products[row][col] = row * col;
}
```
If you build a straight fence 100m long with posts 10m apart, how many posts do you need?

```java
static double[] subarray(double[] a, int m, int n) {
    double[] r = new double[n-m];
    for (int i = m; i <= n; i++)
        r[i-m] = a[i];
    return r;
}
```

Fencepost errors come from counting things rather than the spaces between them, or vice versa, or by neglecting to consider whether one should count one or both ends of a row.

— from the *Jargon File*