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

*Rough* cutoffs:
- lowest A: 31
- lowest B: 25
- lowest C: 14
Digression: Generic Methods
List<String> myList = new LinkedList<String>();

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
    ...
}

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

```
... List<String> myList = new LinkedList<String>();
myList.add("foo");
```
class GenericClass<X> {
    X myField;

    GenericClass (X init) {
        myField = init;
    }

    public X m () {
        return myField;
    }
}

...  
GenericClass<String> g = new GenericClass<String>("foo");  
String s = g.m();  
...
class GenericMethod {
    public static <X> X first (List<X> myList) {
        return myList.get(0);
    }
}

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

actual type parameter (String) is implicit

formal type parameter (to method) ...used in result type
...used in argument type
Random-Access Data
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
- ...
(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

- \( a[i] \) is the \( i \)th element of array \( a \)
- \( a[i] = e \) assigns \( e \) to \( i \)th element of array \( a \)
- \( a.length \) is the number of elements in \( a \)
Java Arrays: Creation

• Create an array `a` of size `n` with elements of type `type`:

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

```java
Int[] a = new int[4];
a[2] = 7;
```

```
a 0 0 7 0
```
Java Arrays: Initialization

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)};
Java Arrays: Aliasing

- Arrays live in the *heap*
- Variables of array type are *pointers*

```java
int[] a = new int[4];
int[] b = a;
a[2] = 7;
System.out.println(b[2]);
```

![Array aliasing diagram]
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...

double sum(double[] a) {
    double total = 0;
    for (int i = 0; i < a.length; i++)
        total += a[i];
    return total;
}

**Initialization**

**Loop condition**

**Update**

\[
\text{for (int } i = 0; i < \text{a.length}; i++)
\]

\[
\text{total } += \text{a}[i];
\]
static void clip (double[] a) {
    for (int i = 0; i < a.length; i++)
        if (a[i] < 0.0)
            a[i] = 0.0;
}
An Updating For Loop
A 2-d array is just an array of arrays...

```java
String[][] names = {
    {"Mr. ", "Mrs. ", "Ms. ", "Smith", "Jones"},
    {"Smith", "Jones"}
};
System.out.println(names[0][0] + names[1][0]); // --> Mr. Smith
System.out.println(names[0][2] + names[1][1]); // --> Ms. Jones
```

String[][] just means (String[])[]
names[1][1] just means (names[1])[1]

More brackets → more dimensions
Multi-Dimensional Arrays

• A 2-d array need not be rectangular
• For example, this code fragment creates a “triangular” matrix...

```java
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;
}
```

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
```
Programming with Arrays
The Problem

• One of the most common difficulties when manipulating numerical indices into arrays is *off-by-one errors*
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*