CS100M

Introduction to Computer Programming

Spring 2004
Lectures 23-24
Java Data Structures

Announcements

- A6 is last assignment
- Prelim 3: Today!
  - A-G in OH 255
  - H-N in HO B14
  - O-Z in UP B17
- Remaining work:
  - Chapters 6, 10
  - Chapter 7

Motivation

- Problem solving with OOP
- Encapsulation
- What's next?
  - Removing redundancy
  - Example:

```
Worker w1, w2, w3, ....
```

  - Better solution for many variables of same type?
    - Static data structures (arrays)
    - Dynamic data structures (lists, trees, ...)

Data Structure

- Reminder: ***Abstract data type***
  - Data and associated actions
  - Essentially a type
  - OOP: make a class
- Special kind of ADT:
  - ***Data structure***
  - Organized collection of data with methods for building, accessing, analyzing
  - Examples:
    - Arrays
    - Lists: Friend of a friend...
    - others?

Static Data Structure

- ***Static data structure***:
  - Can create once for a particular size
  - Good when you know the amount of data
  - Typical implementation of this ADT: arrays

Reference Values

- Arrays in Java – basic rules!
  - arrays are objects
  - indexing uses **[]** and starts at **0**
  - arrays get default values of "zeros"
  - arrays have default field length
- The gist:

```
int[] x;
x = new int[3];
System.out.println(x[2]);
x[0] = 10 + 20;
System.out.println(x.length);
```

- How to view? (see below)

More Detailed Rules

- Arrays are objects
- Java has built-in array class that you can't see but...
- ***pretend*** that Java has the following class:

```
public class Type[] {
   public final int length;
   public Type Type[0];
   public Type Type[1];
   .
   .
   .
   public Type Type[length-1];
   public Type[] [int size] {
      length = size;
   }
   // built in Object methods (e.g.,toString)
}
```

- Java really does think of an array as a special object

Rules: Array Declaration

- All arrays in Java are 1-D
  - Array variables have a special type for declaration:

```
Type[] a; // 1-D array
Type a[]; // 1-D array
```

  - both are OK: usually 1st is better style
  - ***Type*** is any valid type!
  - scope of reference variable?
    - is it a local variable?
    - is it a field?
    - is it a method parameter?
  - all elements must have same type!
    (inheritance will modify this rule a bit)
- How to do multidimensional?

Rules: Multidim. Declaration

- Multidim?
  - Think: `Type[]`, where `Type` is also `Type[]`
  - Kind of like `(Type[])[]`
- Java shortens the notation:

```
Type[][] a; // 2-D array
___________ ; // multidim? Pattern?
```

  - really an array of arrays
- How to create?

9

Create 1-D Array

- Create array
  - Must create 1 dimension at a time!

```
a = new Type[size];
```

  - ***size*** is a valid integer expression
  - size represents the length!
  - looks like a constructor?
  - array filled with default "zeros"
- primitive example:

```
int[] x = new int[3]; // picture?
```

- class example:

```
Person[] p = new Person[2]; // picture?
```

10

Accesing/Inserting Elements

- Syntax:
  - access: ***Type[index]***
  - insert: ***Type[index] = expression;***
- Notes:
  - ***index*** must be an expression of type `int`
  - inserted ***expression*** must have same type as array

11

Some Examples

- Examples:

```
int[] x = new int[2];
x[0] = 10;
x[1] = 20;
// x[2] = 30; // ?
```

```
Person[] p = new Person[2];
p[0] = new Person("Arcturus");
System.out.print(p[0].getName());
// note precedence of [] and .
```

12

Application: Arrays in a class

```
class Data {
   private int[] x;
   public Data(int size) {
      x = new int[size];
   }
   // other methods
}

public class Test {
   public static void main(String[] args) {
      Data d = new Data(4);
      // access d.something(...)
   }
}
```


Create 2-D Array

- Multidim: can avoid doing 1-D arrays one at a time:

```
Type[][] a = new Type[dim1][dim2];
```

- Primitive type example:

```
int[][] x = new int[3][2];
for (int i=0;i<x.length;i++)
   for (int j=0;j<x[0].length;j++)
      x[i][j]=MyMath.randInt(0,1);
```

- Visualization:

More 2-D Arrays

- class example:
  - remember that arrays always fill with defaults!
  - so, you need to create objects and put them in array!

```
Person[][] p = new Person[3][2];
System.out.println(p[3][2]);
// System.out.println((p[3]).getName());
for (int i=0;i<p.length;i++)
   for (int j=0;j<p[0].length;j++)
      p[i][j] = new Person();
```

- Explanation:

Initializer Lists

- ***Initializer list***:

```
Type[] var = {e1, e2, ..., en} ;
```

  - must declare and create on same line!
  - Example:

```
int[] x = {2, 1, -2, 1};
```

- More formal—***anonymous array***:
  - handy for returning an array
  - examples

```
int[][] x = new int[][] {1,3,2};

return new int[][] {0,1,2};
```

Ragged Arrays

- consequence of 1-D array:
  - not all dimensions must be the same
  - can leave a dimension unspecified
- Must remember order of indicies:
  - outermost/leftmost
  - inner/right
  - innermost/rightmost
- Visualization:

Example

```
public class Ragged {
  public static void main(String[] args) {
    int[] a = {1,2,3};
    int[] b = {4,5,6,7,8};
    int[][] x = new int[2][];
    x[0] = a;
    x[1] = b;
    print(x);
  }

  public static void print(int[][] x) {
    for (int i=0;i<x.length;i++) {
      for (int j=0;j<x[i].length;j++)
        System.out.print(x[i][j] + " ");
      System.out.println();
    }
  }

}
```


Dynamic Data Structures

- Arrays:
  - The good:
    - Convenient, clear
    - Quick storage retrieval
  - The bad:
    - Once created, cannot grow any further
  - The ugly
    - To change size, need to create another array (space expense in creating array, time expense in copying)
    - Could force programmer to specify really big, wasteful array at the beginning and force user to stay within size
- When do you need a structure to change?

Fundamental ADTs

- ADT reminder:
  - data
  - operations
- For data structure:
  - ADT organizes data
  - Operations to put, get, find, sort, ...
- Fundamental ADTs
  - Strings
  - Arrays
  - Lists
  - Trees

Sequence Structures

- Basic Operations:
  - put
  - get
- Stack
- Queue
- Priority queue

Search Structures

- Basic operations:
  - insert
  - delete
  - search
- Sorted arrays, sorted lists
- Binary search trees

Vector

- Easy, useful dynamic data structure
- `Vector` is not a linear algebra vector
  - provides access with ops as an array
  - can grow/shrink ***dynamically***
- Already provided by Java, but need to learn a bit more before you can use it....

More Java To Learn: API

- Must use API:
  - see link on course website
  - `Vector` is part of `java.util`
  - except for classes in `java.lang` and your `CLASSPATH`, you must import classes!

```
import classname;
import path.classname;
import path.* ; // everything in package
```

More Java: `Object`

- Many `Vector` headers require objects of type `Object`
  - inheritance: class types can be related
  - allows mixing of types as long they're related
  - think promotion and casting
  - consequence: variables of class `Object` can be aliases to any object of any type!
- Examples:

```
Object x = new Person();
Worker w = new Worker();
x = w;
Integer i = new Integer(4); // java.lang
Object o = i;
System.out.println(i);
```


Vector Example

```
import java.util.Vector;

public class TestVector {
   public static void main(String[] args) {

      Vector v = new Vector(0);
      v.add(new Integer(2));
      v.add(new Integer(-1));
      System.out.println(v);
      System.out.println(v.get(0));

   }
}
```


Lists

- Designing your own "vector"
- Need to use notion of references:
  - Person has a friend
  - that friend has a friend
  - eventually run out friends, so last friend is `null`
- Visualization:

List ADT Design

- data
  - node
    eg, Person has a name, ID, age, ...
  - reference to another object of related type
    eg, friend, next, prev, parent, child, ...
- operations
  - create
  - put
  - get
  - size
  - others? for ideas, look in API! (`java.util.List`)

Example List Class

```
class PersonL {
   private String name;
   private PersonL next;

   public PersonL(String name, PersonL next) {
      this.name = name;
      this.next = next;
   }

   public String getName() { return name; }
   public PersonL getNext() { return next; }
   public void setName(String name) { this.name = name; }
   public void setNext(PersonL next) { this.next = next; }
   public String toString() { return name+"->"+next; }
}
```

29

Example Continued

```
public class Friendster {
   public static void main(String[] args) {

      PersonL c = new PersonL("C",null);
      PersonL b = new PersonL("B",c);
      PersonL a = new PersonL("A",b);

      System.out.println(a);
      System.out.println(size(a));

   }
```

30

Example Still Continued

```
   public static int size(PersonL p) {

      int size = 0;
      while (p != null) {
         size++;
         p = p.getNext();
      }
      return size;
   }

}
```

31