Introduction to Object-Oriented Programming

Arrays, Part 2 of 2

Christopher Simpkins
chris.simpkins@gatech.edu

A few more array topics

- Variable arity parameters
- Multi-dimensional arrays
- Partially filled arrays

Variable Arity Parameters

- The *arity* of a method is its number of formal parameters.
- So far, all the methods we've written have fixed arity.
- The last parameter to a method may be a *variable arity parameter*, a.k.a. *var args* parameter (JLS §8.4.1), whose syntax is simply to add an ellipse (`...`) after the type name.
- The var args parameter is accessed as an array inside the method.

For example:

```
public static int max(int ... numbers) {
    int max = numbers[0];
    for (int i = 1; i < numbers.length; ++i) {
        if (numbers[i] > max) max = number;
    }
    return max;
}
```

Multidimensional Arrays

You can create arrays of any number of dimensions simply by adding additional square brackets for dimensions and sizes. For example:

```
char[][] grid;
```

The declaration statement above:

- Declares a 2-dimensional array of `char`.
- As with one-dimensinal arrays, `char` is the base type.
- Each element of `grid`, which is indexed by two `int` expressions, is a `char` variable.

Initializing Multidimensional Arrays

Initialization of 2-dimensional arrays can be done with `new`:

```
grid = new char[10][10];
```

or with literal initialization syntax:

```
char[][] grid = {{' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '},
                 {' ', '*', '*', ' ', ' ', ' ', ' ', '*', '*', ' '},
                 {' ', '*', '*', ' ', ' ', ' ', ' ', '*', '*', ' '},
                 {' ', ' ', ' ', ' ', '*', '*', ' ', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', '*', '*', ' ', ' ', ' ', ' '},
                 {' ', '*', ' ', ' ', ' ', ' ', ' ', ' ', '*', ' '},
                 {' ', ' ', '*', ' ', ' ', ' ', ' ', '*', ' ', ' '},
                 {' ', ' ', ' ', '*', '*', '*', '*', ' ', ' ', ' '},
                 {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}};
```

Notice that a 2-dimensional array is an array of 1-dimensional arrays (and a 3-dimensional array is an array of 2-dimensional arrays, and so on).

Visualizing Multidimensional Arrays

Our 2-dimensional `grid` array can be visualized as a 2-d grid of cells.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
grid[0]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '
grid[1]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '
grid[2]	' '	'*'	'*'	' '	' '	' '	' '	'*'	'*'	' '
grid[3]	' '	'*'	'*'	' '	' '	' '	' '	'*'	'*'	' '
grid[4]	' '	' '	' '	' '	'*'	'*'	' '	' '	' '	' '
grid[5]	' '	' '	' '	' '	'*'	'*'	' '	' '	' '	' '
grid[6]	' '	'*'	' '	' '	' '	' '	' '	' '	'*'	' '
grid[7]	' '	' '	'*'	' '	' '	' '	' '	'*'	' '	' '
grid[8]	' '	' '	' '	'*'	'*'	'*'	'*'	' '	' '	' '
grid[9]	' '	' '	' '	' '	' '	' '	' '	' '	' '	' '

And an individual cell can be accessed by supplying two indices:

```
grid[3][2] == '*'; // true
```

Traversing Multidimensional Arrays

Traverse 2-dimensional array by nesting loops. The key to getting it right is to use the right `length`s.

```
for (int row = 0; row < grid.length; ++row) {
    for (int col = 0; col < grid[row].length; ++col) {
        System.out.print(grid[row][col]);
    }
    System.out.println();
}
```

Note that the for loops above traverse the grid in row-major order. We can traverse the grid in column-major order by reversing the nesting of the for loops:

```
for (int col = 0; col < grid[0].length; ++col) {
    for (int row = 0; row < grid.length; ++row) {
        System.out.print(grid[row][col]);
    }
    System.out.println();
}
```

See Smiley.java

Ragged Arrays

It's possible to create *ragged arrays* by creating nested arrays of variable length. For example:

```
double [][] ragged = new double[3][];
ragged[0] = new double[5];
ragged[1] = new double[10];
ragged[2] = new double[4];
```

Can we traverse array `ragged` in row-major order?
Can we traverse array `ragged` in column-major order?

Partially Filled Arrays

Sometimes we only use part of an array:

```
int[] assignments = new int[10];
int lastAssignment = 0;
assginments[lastAssignment++] = 100;
// As more assignments are graded, more of assignments[] is used ...
```

Note that we had to keep track of the last used index in the array.

Now that we know how to define classes, we can do better ...

A Partial Int Array ADT

ADT: `PartialIntArray`
Data:

- **elements: int[]** - an array of `int` elements
- **size: int** - the number of elements currently in use

Operations:

- **new** - construct a new `PartialIntArray`
- **add(element: int)** - add an element to this `PartialIntArray`
- **get(i: int)** - get the `i`th element of this `PartialIntArray`
- **size** - get the size of this `PartialIntArray`

PartialIntArray

Constructors:

```
public class PartialIntArray {
    private int[] elements;
    private int size;

    public PartialIntArray() {
        this(10);
    }
    public PartialIntArray(int initialCapacity) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        this.elements = new int[initialCapacity];
    }
    // ...
}
```

Note how the no-arg constructor delegates to the other constructor with `this(...)`.

Adding Elements to PartialIntArray

Assuming we don't need to automatically "grow" our array-backed data structure when needed (like `java.util.ArrayList` does), we can add elements like this:

```
public class PartialIntArray {

  // ...

    public void add(int e) {
        elements[size++] = e;
    }

}
```

Accessing Elements of PartialIntArray

We can provide access to elements of our PartialIntArray with:

```
public class PartialIntArray {
    // ...

    public int get(int index) {
        return elements[index];
    }
}
```

Note that we're providing access to individual elements, not the entire underlying array. The underlying array is an implementation detail.

Traversing a PartialIntArray

To allow clients of `PartialIntArray` to traverse its elements, we need one more method in our API - size:

```
public class PartialIntArray {
    // ...
    public int size() {
        return this.size;
    }
}
```

Now we can add elements to a `PartialIntArray` and traverse it in a manner similar to regular arrays:

```
PartialIntArray pia = new PartialIntArray();
pia.add(1);
// add more ...
for (int i = 0; i < pia.size(); ++i) {
    System.out.println(pia.get(i));
}
```

Why did we define a `size()` a method rather than making the `size` instance variable public?

Encapsulation and Information Hiding

Our `PartialIntArray` class demonstrates two important concepts in software engineering: encapsulation and information hiding.

- The `elements` instance variable was private and never exposed in its entirety to clients.
- All access to `elements` was provided through instance methods, so we can ensure data consistency by enforcing invariants, validating input, etc.
- We could have called our class `RandomAccessIntList`, because the fact that an array was used is an implementation detail. Client code need not be aware of implementation details (to an extent ...).

`PartialIntArray` and `java.util.ArrayList`

Our `PartialIntArray` example was inspired by Java's standard ArrayList class. Try these exercises at home:

- Consider what happens if a user of `PartialIntArray` supplies an out of bounds index to `get`, i.e., < 0 or $\geq$ `size`. Is this desirable? If not, how would you improve it?
- Add a `remove(int index)` method that removes the element at `index`.
- Make `PartialIntArray` automatically resize, that is, expand its capacity if you add a `size`th element. How would you do this?
- Look at the source for `java.util.ArrayList`.

As in any craft, study the work of masters to improve your own skillz.