Programming Languages and Techniques (CIS120)

Lecture 23

Mar 14, 2012

Java ASM & Encapsulation
Announcements

• HW07 is available on the web
  – Image processing in Java
  – Due tomorrow, March 15\textsuperscript{th} at 11:59:59pm

• HW08 will be available on Friday
  – Due Monday, March 26\textsuperscript{th} at 11:59:59pm

• Midterm 1 regrade deadline on Friday

• Midterm 2 on Friday, March 30\textsuperscript{th}
The Java Abstract Stack Machine

Objects, Arrays and Static Methods
Java Abstract Stack Machine

• Similar to OCaml Abstract Stack Machine
  – Distinction between “primitive” and “reference values”

• Workspace
  – Contains the currently executing code

• Stack
  – Remembers the values of local variables and "what to do next" after function/method calls

• Heap
  – Stores reference types: objects and arrays

• Differences:
  – Everything, including stack slots, is mutable by default
  – Special reference value: null
  – Heap objects store dynamic class information
  – Code stored in Class Table
Heap Values

Objects
• Name of the class that constructed it
• Values for all of the fields

Arrays
• Type of values that it stores
• Length
• Values for all of the elements

class Node {
    private int elt;
    private Node next;
    ...
}

int [] a = { 0, 0, 7, 0 };

fields may or may not be mutable

length never mutable
elements always mutable
Object Aliasing example

```java
public class Node {
    private int elt;
    private Node next;
    public Node(int e0, Node n0) {
        elt = e0;
        next = n0;
    }

    public static int m() {
        Node n1 = new Node(1,null);
        Node n2 = new Node(2,n1);
        Node n3 = n2;
        n3.next.next = n2;
        Node n4 = new Node(4,n1.next);
        n2.next.elt = 17;
        return n1.elt;
    }
}
```
### ASM Example

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.m();</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Static* method call, similar to OCaml function call:

- Save the workspace to the stack
- Look up method named ‘m’ in the class table
- Put method parameters on the stack
- Put method body in the workspace
Node n1 = new Node(1, null);
Node n2 = new Node(2, n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 17;
return n1.elt;

We’ll omit this in the rest of the example.
ASM Example

Workspace

Stack

Heap

Node n1 = \texttt{new Node(1, null)};
Node n2 = \texttt{new Node(2, n1)};
Node n3 = n2;
n3.next.next = n2;
Node n4 = \texttt{new Node(4, n1.next)};
n2.next.elt = 17;
return n1.elt;

Note: we’re skipping details here about how the constructor works. We’ll fill them in next week. For now, assume the constructor allocates and initializes the object in one step.
Node n1 = ;
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
return n1.elt;

Local variable definition: as in OCaml, add to the stack. Unlike OCaml, the value is mutable.
Node n2 = new Node(2, n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 17;
return n1.elt;
Node n2 = ;
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
return n1.elt;
Workspace

Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 17;
return n1.elt;

Stack

Heap
MutaGng a field

Workspace

n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
return n1.elt;

Stack

Heap

Node

elt 1
next null

Node

elt 2
next null
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
return n1.elt;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 17;
return n1.elt;
Node n4 = new Node(4, n1.next);
n2.next.elt = 17;
return n1.elt;
Node n4 = new Node(4, null);
n2.next.elt = 17;
return n1.elt;
Node n4 = ;
n2.next.elt = 17;
return n1.elt;
Mutating a field

Workspace

```
n2.next.elt = 17;
return n1.elt;
```

Stack

```
n1
n2
n3
n4
```

Heap

```
Node
elt 1
next
```

```
Node
elt 2
next
```

```
Node
elt 4
next
```
n2.next.elt = 17;
return n1.elt;
n2.next.elt = 17;
return n1.elt;
return n1.elt;
Design Exercise: Resizable Arrays

Arrays that grow without bound.
public class ResArray {

    /** Constructor, takes no arguments. */
    public ResArray() { … }

    /** Access the array at position \textit{i}. If position \textit{i} has not yet * been initialized, return 0. */
    public int get(int i) { … }

    /** Modify the array at position \textit{i} to contain the value \textit{v}. */
    public void set(int i, int v) { … }

    /** Return the extent of the array. */
    public int getExtent() { … }

}

Object Invariant: extent is always 1 past the last nonzero value in data (or 0 if the array is all zeros)
ResArray x = new ResArray();
x.set(3,2);
x.set(4,1);
x.set(4,0);
ResArray ASM

Workspace

```java
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```

Stack

Heap

<table>
<thead>
<tr>
<th>ResArray</th>
<th>int[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>length</td>
</tr>
<tr>
<td>extent</td>
<td>0</td>
</tr>
</tbody>
</table>

CIS120 / Spring 2012
Workspace

```java
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```
ResArray ASM

Workspace

```java
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```

Stack

Heap

```
<table>
<thead>
<tr>
<th>ResArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
</tr>
<tr>
<td>extent</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>int[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>0 0 0 2</td>
</tr>
</tbody>
</table>
```
ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```

Stack

Heap

```
ResArray x

<table>
<thead>
<tr>
<th>data</th>
<th>extent</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

```
int[]
length 8
0 0 0 0 2 1 0 0 0 0
```

```
int[]
length 4
0 0 0 2
```
ResArray ASM

Workspace

```
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```

Stack

Heap

```
ResArray x

<table>
<thead>
<tr>
<th>data</th>
<th>extent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
int[]

<table>
<thead>
<tr>
<th>length</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 2 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```
Workspace

```
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
```

```
Stack
```

```
Heap
```

```
int[]
```

```
<table>
<thead>
<tr>
<th>length</th>
<th>8</th>
</tr>
</thead>
</table>

```
0 0 0 0 2 0 0 0 0 0
```

CIS120 / Spring 2012
Demo

ResArray.java
ResArrayTest.java
public class ResArray {

    /** Constructor, takes no arguments. */
    public ResArray() { … }

    /** Access the array at position i. If position i has not yet
     * been initialized, return 0.
     */
    public int get(int i) { … }

    /** Modify the array at position i to contain the value v. */
    public void set(int i, int v) { … }

    /** Return the extent of the array. */
    public int getExtent() { … }

    /** The smallest prefix of the ResArray
     * that contains all of the nonzero values as a normal array.
     */
    public int[] values() { … }
}

Object Invariant: extent is always 1 past the last nonzero value in data
(or 0 if the array is all zeros)
public int[] values() {
    int[] values = new int[extent];
    for(int i=0; i<extent; i++) {
        values[i] = data[i];
    }
    return values;
}

public int[] values() {
    if (data.length == extent) {
        return data;
    }
    int[] values = new int[extent];
    for(int i=0; i<extent; i++) {
        values[i] = data[i];
    }
    return values;
}
ResArray ASM

Workspace

ResArray x = new ResArray();
x.set(3, 2);
int[] y = x.values();
y[3] = 0;

Stack

Heap

ResArray

data
extent 4

int[]
length 4
0 0 0 2
ResArray ASM

Workspace

```java
ResArray x = new ResArray();
x.set(3, 2);
int[] y = x.values();
y[3] = 0;
```

Diagram:
- Stack: `x` and `y`
- Heap:
  - `ResArray` with:
    - `data`: empty
    - `extent`: 4
  - `int[]`: length 4, values: 0, 0, 0, 2
ResArray ASM

Workspace

```java
ResArray x = new ResArray();
x.set(3, 2);
int[] y = x.values();
y[3] = 0;
```

Stack

Heap

```
ResArray

<table>
<thead>
<tr>
<th>data</th>
<th>extent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
int[]

<table>
<thead>
<tr>
<th>length</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Invariant violation!
Object encapsulation

- All modification to the state of the object must be done using the object's own methods.

- Use encapsulation to preserve invariants about the state of the object.

- Enforce encapsulation by not returning aliases. Make a COPY of a data structure if necessary.
Mutable Queue ML Interface

module type QUEUE =
  sig
    (* type of the data structure *)
    type 'a queue

    (* Make a new, empty queue *)
    val create : unit -> 'a queue

    (* Add a value to the end of the queue *)
    val enq : 'a -> 'a queue -> unit

    (* Remove the front value and return it (if any) *)
    val deq : 'a queue -> 'a

    (* Determine if the queue is empty *)
    val is_empty : 'a queue -> bool

    (* Remove the first occurrence of the value. *)
    val remove : 'a -> 'a queue -> unit
  end