Programming Languages and Techniques (CIS120)

Lecture 24
October 29, 2018

Arrays, Java ASM
Chapter 21 and 22
Announcements

• HW6: Java Programming (Pennstagram)
  – Due TOMORROW at 11:59pm

• Reminder: please complete mid-semester survey
  – See post on Piazza

• Upcoming: Midterm 2
  – Friday, November 9th in class
  – Coverage: mutable state, queues, deques, GUI, Java
Design Exercise: Resizable Arrays

Arrays that grow without bound.
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() { ... }
}

Object Invariant: extent is 1 past the last nonzero value in data
(can be 0 if the array is all zeros)
public class ResArray {

    private int[] data = {};

    /** 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 idx) {
        if (idx >= data.length) {
            return 0;
        } else {
            return data[idx];
        }
    }

    ...
}
public class ResArray {

    ... 

    private int[] data = {}; 

    /** Modify the array at position i to contain the value v. */ 
    public void set(int idx, int val) {
        if (idx >= data.length) {
            int[] newdata = new int[idx+1]
            for (int i=0; i < data.length; i++) {
                newdata[i] = data[i];
            }
            data = newdata;
        }
        data[idx] = val;
    } 

    public int[] values() {
        return data;
    }
}

public class ResArray {

    private int[] data = {};

    /** Modify the array at position i to contain the value v. */
    public void set(int idx, int val) {
        if (idx >= data.length) {
            int[] newdata = new int[Math.max(idx+1, data.length*2)]
            for (int i=0; i < data.length; i++) {
                newdata[i] = data[i];
            }
            data = newdata;
        }
        data[idx] = val;
    }

    public int[] values() {
        return data;
    }
}
private int extent = 0;
   /* INVARIANT: extent = 1+index of last nonzero
    * element, or 0 if all elements are 0. */

/** Modify the array at position i to contain the value v. */
public void set(int idx, int val) {
   if (idx < 0) {
      throw new IllegalArgumentException();
   }
   grow(idx);
   data[idx] = val;
   if (val != 0 && idx+1 > extent) {
      extent = idx+1;
   }
   if (val == 0 && idx+1 == extent) {
      while (extent > 0 && data[extent-1] == 0) {
         extent--;
      }
   }
}

/** Return the extent of the array. */
public int getExtent() {
   return extent;
}
Revenge of the Son
of the
Abstract Stack Machine
The Java Abstract Stack Machine

Objects, Arrays, and Static Methods
Java Abstract Stack Machine

• Similar to OCaml Abstract Stack Machine
  – 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

• Key differences:
  – Everything, including stack slots, is mutable by default
  – Objects store what class was used to create them
  – Arrays store type information and length
  – New component: Class table (coming soon)
Java Primitive Values

- The values of these data types occupy (less than) one machine word and are stored directly in the stack slots.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bit</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>16-bit integer</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>32-bit integer</td>
<td>-2³¹ to 2³¹ - 1</td>
</tr>
<tr>
<td>long</td>
<td>64-bit integer</td>
<td>-2⁶³ to 2⁶³ - 1</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE floating point</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE floating point</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>true false</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unicode character</td>
<td>'a' 'b' '\u0000'</td>
</tr>
</tbody>
</table>
Heap Reference Values

Arrays

- Type of values that it stores
- Length
- Values for all of the array elements

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

Objects

- Name of the class that constructed it
- Values for all of the fields

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

- Length
- Values for all of the array elements

<table>
<thead>
<tr>
<th>int[]</th>
<th>length 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- fields may or may not be mutable
- public/private not tracked by ASM

<table>
<thead>
<tr>
<th>Node</th>
<th>elt 1</th>
<th>next null</th>
</tr>
</thead>
</table>

- length never mutable; elements always mutable
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></td>
</tr>
</tbody>
</table>
```

Workspace

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

Workspace

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

Workspace

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

Stack

Heap

ResArray

data
extent 4

int[]
length 4
0 0 0 2
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
Workspace

ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
ResArray x = new ResArray();
x.set(3, 2);
x.set(4, 1);
x.set(4, 0);
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() { ... }
}
Values Method

```java
public int[] values() {
    int[] values = new int[extent];
    for (int i=0; i<extent; i++) {
        values[i] = data[i];
    }
    return values;
}

public int[] values() {
    return data;
}
```

Or maybe we can do it more straightforwardly? ...
This optimized implementation of values correctly encapsulates the state of the ResArray object.

```java
public int[] values() {
    return data;
}
```
This optimized implementation of values correctly encapsulates the state of the ResArray object.

```java
public int[] values() {
    return data;
}
```

1. True
2. False

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

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

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 from methods.
Objects on the ASM
What does the heap look like at the end of this program?

```java
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```

```java
public class Counter {
    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }
}
```
What does the ASM look like at the end of this program?

```java
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
```

```java
Counter[] a = { new Counter(), new Counter() };
Counter[] b = { a[0], a[1] };
a[0].inc();
b[0].inc();
int ans = a[0].inc();
```
public class Node {
    public int elt;
    public Node next;
    public Node(int e0, Node n0) {
        elt = e0;
        next = n0;
    }
}

public class Test {
    public static void main(String[] args) {
        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 = 9;
        System.out.println(n1.elt);
    }
}
What does the following program print?
1 – 9
or 0 for "NullPointerException"

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

class Test {
    public static void main(String[] args) {
        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 = 9;
        System.out.println(n1.elt);
    }
}

Answer: 9
```
<table>
<thead>
<tr>
<th>Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node n1 = new Node(1,null);</td>
</tr>
<tr>
<td>Node n2 = new Node(2,n1);</td>
</tr>
<tr>
<td>Node n3 = n2;</td>
</tr>
<tr>
<td>n3.next.next = n2;</td>
</tr>
<tr>
<td>Node n4 = new Node(4,n1.next);</td>
</tr>
<tr>
<td>n2.next.elt = 9;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
</table>
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 = 9;

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.
Workspace

Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```java
Node n2 = null;
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 9;
```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4, n1.next);
n2.next.elt = 9;
Workspace
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
Node n4 = new Node(4, n1.next);
n2.next.elt = 9;
Node n4 = ;
n2.next.elt = 9;
n2.next.elt = 9;
n2.next.elt = 9;
Workspace
\n\texttt{n2.next.elt = 9;}

Stack
\n\begin{align*}
\text{n1} & : \text{Node} \\
\text{n2} & : \text{Node} \\
\text{n3} & : \text{Node} \\
\text{n4} & : \text{Node}
\end{align*}

Heap
\n\begin{align*}
\text{Node} & : \text{elt = 9} \\
\text{Node} & : \text{elt = 2} \\
\text{Node} & : \text{elt = 4}
\end{align*}
OOooooo programming

OO

Subtypes
Interfaces

- Give a type for an object based on what it *does*, not on how it was constructed
- Describes a contract that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```java
public interface Displaceable {
    public int getX();
    public int getY();
    public void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!
Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface.
- That class fulfills the contract implicit in the interface.

```java
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```
Another implementation

```java
public class Circle implements Displaceable {
    private Point center;
    private int radius;
    public Circle(Point initCenter, int initRadius) {
        center = initCenter;
        radius = initRadius;
    }
    public int getX() { return center.getX(); }
    public int getY() { return center.getY(); }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
}
```

Objects with different local state can satisfy the same interface

*Delegation*: move the circle by moving the center
The following snippet of code typechecks:

```java
public void moveItALot (Displaceable s) {
    ...
} //omitted

... // elsewhere
Circle c = new Circle(10,10,10);
moveItAlot(c);
```

1. True
2. False

Answer: True
Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces

Types can have many different supertypes / subtypes
1. Interface extension
2. Class extension (Simple inheritance)
Interface Extension

- Build richer interface hierarchies by extending existing interfaces.

```java
public interface Displaceable {
    double getX();
    double getY();
    void move(double dx, double dy);
}
```

```java
public interface Area {
    double getArea();
}
```

```java
public interface Shape extends Displaceable, Area {
    Rectangle getBoundingBox();
}
```

Note the use of the “extends” keyword.

The Shape type includes all the methods of Displaceable and Area, plus the new `getBoundingBox` method.
• Shape is a subtype of both Displaceable and Area.
• Circle and Rectangle are both subtypes of Shape, and, by transitivity, both are also subtypes of Displaceable and Area.
• Note that one interface may extend several others.
  – Interfaces do not necessarily form a tree, but the hierarchy has no cycles.
Class Extension: Inheritance

• Classes, like interfaces, can also extend one another.
  – Unlike interfaces, a class can extend only one other class.

• The extending class inherits all of the fields and methods of its superclass, may include additional fields or methods.
  – This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).
  – Class extension should never be used when “is a” does not relate the subtype to the supertype.
public class DisplaceableImpl implements Displaceable {
    private Point pt;
    public DisplaceableImpl(int x, int y) {
        this.pt = new Point(x,y);
    }
    public int getX() { return pt.getX(); }
    public int getY() { return pt.getY(); }
    public void move(int dx, int dy) {
        pt.move(dx, dy);
    }
}

public class Circle extends DisplaceableImpl {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x,y);
        this.radius = radius;
    }

    public int getRadius() { return radius; }
}
Simple Inheritance

• In *simple inheritance*, the subclass only *adds* new fields or methods.
• Use simple inheritance to *share common code* among related classes.
• Example: Circle, and Rectangle have *identical* code for `getX()`, `getY()`, and `move()` methods when implementing Displaceable.
Subtyping with Inheritance

- Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.
  - e.g. Circle is a subtype of Area, but Point is not.
Example of Simple Inheritance

See: Main2.java
• Constructors cannot be inherited
  – They have the wrong names!
  – A subclass invokes the constructor of its super class using the keyword `super`.
  – **Super** *must* be the first line of the subclass constructor
    • if the parent class constructor takes no arguments, it is OK to omit the call to `super`.
      – It is then called implicitly.
public class DisplaceableImpl implements Displaceable {
    private Point pt;
    public DisplaceableImpl(int x, int y) {
        this.pt = new Point(x, y);
    }
    public int getX() { return pt.getX(); }
    public int getY() { return pt.getY(); }
    public void move(int dx, int dy) {
        pt.move(dx, dy);
    }
}

public class Circle extends DisplaceableImpl {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    public int getRadius() { return radius; }
}
Other forms of inheritance

- Java has other features related to inheritance (some of which we will discuss later in the course):
  - A subclass might override (re-implement) a method already found in the superclass.
  - A class might be abstract – i.e. it does not provide implementations for all of its methods (its subclasses must provide them instead).

- These features are hard to use properly, and the need for them arises only in somewhat special cases
  - Making reusable libraries
  - Special methods: equals and toString

- We recommend avoiding all forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition.

Especially: avoid overriding.
Subtype Polymorphism*

Main idea:

Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```java
// in class C
public static void leapIt(DisplaceableImpl c) {
    c.move(1000,1000);
}
// somewhere else
C.leapIt(new Circle (10));
```

If B is a subtype of A, it provides all of A’s (public) methods.

Due to dynamic dispatch, the behavior of a method depends on B’s implementation.
  – Simple inheritance means B's method is inherited from A
  – Otherwise, behavior of B should be “compatible” with A’s behavior

*polymorphism = many shapes
public class Object {
    boolean equals(Object o) {
        ... // test for equality
    }
    String toString() {
        ... // return a string representation
    }
    ... // other methods omitted
}

• Object is the root of the class tree.
  – Classes that leave off the “extends” clause implicitly extend Object
  – Arrays also implement the methods of Object
  – This class provides methods useful for all objects to support

• Object is the highest type in the subtyping hierarchy.
Recap: Subtyping

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes “by fiat” of Object