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.

Resizable Arrays

```
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)

ResArray (review)

```
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];
      }
  }
…
```

ResArray (review)

```
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;
  }
}
```

ResArray (review)

```
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;
  }
}
```

Adding *extent*

```
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.

Type	Description	Values
`byte`	8-bit	`-128 to 127`
`short`	16-bit integer	`-32768 to 32767`
`int`	32-bit integer	-2^{31} to $2^{31} - 1$
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit IEEE floating point	
`double`	64-bit IEEE floating point	
`boolean`	true or false	`true false`
`char`	16-bit unicode character	`'a' 'b' '\u0000'`

Heap Reference Values

Arrays

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

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

length *never* mutable; elements *always* mutable

Objects

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

```
class Node {
    private int elt;
    private Node next;
    …
}
```

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

ResArray ASM

Workspace

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

Stack

Heap

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);
```

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);
```

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);
```

Resizable Arrays

Object Invariant: extent is always 1 past the last nonzero value in data
(or 0 if the array is all zeros)

```
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

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

Or maybe we can do it more straightforwardly? …

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

This optimized implementation of values correctly encapsulates the state of the ResArray object.

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

True

False

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

This optimized implementation of values correctly encapsulates the state of the ResArray object.

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

1. True
2. False

Answer: False

ResArray ASM

Workspace

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

ResArray ASM

Workspace

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

ResArray ASM

Workspace

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

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?

```
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 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?

```
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 Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

What does the following program print?

```
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);
  }

}
```

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

What does the following program print?

1 – 9

or 0 for "NullPointerException"

```
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);
  }

}
```

Answer: 9

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 = 9;
```

Stack

Heap

Workspace

```
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;
```

Stack

Heap

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;
```

Workspace
Node n2 = ;
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
Stack
n1
Heap
Node
elt
1
next
null
Node
elt
2
next

Workspace

```
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;
```

Workspace
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
null
Node
elt
2
next

Workspace

```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Workspace

```
Node n4 = new Node(4,n1.next);
n2.next.elt = 9;
```

Workspace
Node n4 =;
n2.next.elt = 9;
Stack
n1
n2
n3
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

Workspace

```
n2.next.elt = 9;
```

Workspace
n2.next.elt = 9;
Stack
n1
n2
n3
n4
Heap
Node
elt
1
next
Node
elt
2
next
Node
elt
4
next

Workspace
n2.next.elt = 9;
Stack
n1
n2
n3
n4
Heap
Node
elt
9
next
Node
elt
2
next
Node
elt
4
next

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

keyword

```
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

```
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;
  }
}
```

interfaces implemented

methods required to satisfy contract

Another implementation

```
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:

```
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

Extension

1. Interface extension
2. Class extension (Simple inheritance)

Interface Extension

- Build richer interface hierarchies by *extending* existing interfaces.

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

public interface Area {
  double getArea();
}

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

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

Note the use of the “extends” keyword.

Interface Hierarchy

```
class Point implements Displaceable {
  … // omitted
}
class Circle implements Shape {
  … // omitted
}
class Rectangle implements Shape {
  … // omitted
}
```

- 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.

Class Extension: Inheritance

```
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

Inheritance: Constructors

- Contructors *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.

Class Extension: Inheritance

```
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.

```
// 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

Object

```
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

Extends

Implements

Subtype by fiat