Programming Languages and Techniques (CIS120e)

Lecture 26

Nov. 10, 2010

Java Abstract Stack Machine
Announcements

• Homework 8 (SpellChecker) is due Nov 15th.

• Midterm 2 is this Friday, November 12th
The Java Abstract Stack Machine

1. Class tables
2. Constructors and “this”
3. Dynamic dispatch
4. Static members
Refinements to the Stack Machine

• Code is stored in a *class table*, which is a special part of the heap:
  – When a program starts, the JVM initializes the class table
  – Each class has a pointer to its (unique) parent in the class tree
  – A class stores the constructor and method code for its instances
  – The class also stores *static* members

• Constructors:
  – Allocate space in the heap
  – (Implicitly) invoke the super class constructor, then run the constructor body

• Objects and their methods:
  – Each object in the heap has a pointer to the class table of its dynamic type (the one it was created with via `new`).
  – A method invocation “o.m(…)” uses o’s class table to “dispatch” to the appropriate method code (might involve searching up the class hierarchy).
  – Methods and constructors take an implicit “this” parameter, which is a pointer to the object whose method was invoked.
public class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

public class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
public class Counter {
    private int x;
    public Counter () { super(); this.x = 0; }
    public void incBy(int d) { this.x = this.x + d; }
    public int get() { return this.x; }
}

public class Decr extends Counter {
    private int y;
    public Decr (int initY) { super(); this.y = initY; }
    public void dec() { this.incBy(-this.y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
### Constructing an Object

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Stack</th>
<th>Heap</th>
<th>Class Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Decr d = new Decr(2);</code></td>
<td></td>
<td></td>
<td><strong>Object</strong></td>
</tr>
<tr>
<td><code>d.dec();</code></td>
<td></td>
<td></td>
<td><strong>String toString(){}</strong></td>
</tr>
<tr>
<td><code>int x = d.get();</code></td>
<td></td>
<td></td>
<td><strong>boolean equals...</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Decr</strong></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>extends</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Decr(int initY) { ... }</code></td>
<td><strong>Counter</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>void dec() {incBy(-y);}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>extends</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Counter() { x = 0; }</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>void incBy(int d){...}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>int get() {return x;}</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Decr</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>extends</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>Decr(int initY) { ... }</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>void dec() {incBy(-y);}</code></td>
<td></td>
</tr>
</tbody>
</table>

CIS120e / Fall 2010
Allocating Space on the Heap

Workspace

super();
this.y = initY;

Stack

Decr d = _;
d.dec();
int x = d.get();

this
initY 2

Heap

Decr

x 0
Y 0

Class Table

Object

String toString(){...}
boolean equals...
...

Counter

extends Object
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr

extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}

Invoking a constructor:
• allocates space for a new object in the heap
• includes slots for all fields of all ancestors in the class tree (here: x and y)
• creates a pointer to the class – this is the object’s dynamic type
• runs the constructor body after pushing parameters and this onto the stack

Note: fields start with a “sensible” default
- 0 for numeric values
- null for references
Call to super:
• The constructor (implicitly) calls the super constructor
• Remember that invoking a method/constructor pushes the saved workspace, the method params (none here) and a new this pointer.

```java
calling super!

workspace

stack

heap

class table

Object

String toString() { ... }

boolean equals...

Counter

declares Object

Counter() { x = 0; }

void incBy(int d) {...}

int get() { return x; }

Decr

declares Counter

Decr(int initY) { ... }

void dec() { incBy(-y); }
```
Abstract Stack Machine

Workspace

super();
this.x = 0;

Stack

Decr d = _;
d.dec();
int x = d.get();

this

Heap

Decr

x

0

Y

0

Class Table

Object

String toString(){...}
boolean equals...
...

Counter

extends Object

Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }
void dec(){incBy(-y);}

(Running Object’s default constructor omitted.)
Assigning into the `this.x` field goes in two steps:
- look up the value of this in the stack
- write to the “x” slot of that object.
Assigning into the this.x field goes in two steps:
- look up the value of this in the stack
- write to the “x” slot of that object.
Done with the call to “super”, so pop the stack to the previous workspace.
Continuing

Workspace

this.y = initY;

Stack

Decr d = _;
d.dec();
int x = d.get();

this

initY 2

Heap

Decr

x 0
Y 0

Class Table

Object

String toString(){...}
boolean equals...
...

Counter

extends Object
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr

extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}

Continue in the Decr class’s constructor.
Abstract Stack Machine

Workspace

this.y = 2;

Stack

Decr d = _;
d.dec();
int x = d.get();

this

initY 2

Heap

Decr

x 0

Y 0

Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}
Assigning to a field

Assignment into the this.y field.

(This really takes two steps as we saw earlier, but we’re skipping some for the sake of brevity...)
Done with the call to the Decr constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the this pointer).
Returning the Newly Constructed Object

Workspace

Stack

Heap

Class Table

Decr d = _;
d.dec();
int x = d.get();

Decr

Object

Counter

extends Object

extends Counter

String toString() {
...
}

boolean equals...

...

Decr

d = _;
d.dec();

Counter()

{ x = 0; }

void incBy(int d){...

int get() {return x;}

Decr

Decr(int initY) { ... }

void dec() { incBy(-y); }

Continue executing the program.
Allocating a local variable

Allocate a stack slot for the local variable d. It’s mutable... (see the bold box in the diagram).

Aside: since, by default, fields and local variables are mutable, we may omit the bold boxes and just assume the contents can be modified.

CIS120e / Fall 2010
Invoke the dec method on the object. The code can be found by “pointer chasing”.

This process is called dynamic dispatch – which code is run depends on the dynamic type of the object. (In this case, Decr.)
Call the method, remembering the current workspace and pushing the this pointer and any arguments (none in this case).
Reading A Field’s Contents

Workspace

Stack

Heap

Class Table

Object
_String toString(){...}
boolean equals...
_

Counter
extends Object
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr
extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}
this.incBy(-2);

Stack:
- d
- __;
  int x = d.get();
- this

Heap:
- Decr
  - x: 0
  - y: 2

Class Table:
- Object
  - String toString(){...}
  - boolean equals...
  - ...

- Counter
  - extends Object
  - Counter() { x = 0; }
  - void incBy(int d){...}
  - int get() {return x;}

- Decr
  - extends Counter
  - Decr(int initY) { ... }
  - void dec(){incBy(-y);}
Invoke the `incBy` method on the object via dynamic dispatch.

In this case, the `incBy` method is *inherited* from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.

The search is guaranteed to succeed – Java’s static type system ensures this.
Running the body of `incBy`

It takes a few steps...

Body of `incBy`:
- reads `this.x`
- looks up `d`
- computes result `this.x + d`
- stores the answer (-2) in `this.x`

```
this.x = this.x + d;
this.x = -2;
```

### Workspace
```
this.x = this.x + d;
this.x = -2;
```

### Stack
```
d
_; int x = d.get();
this
_; d
this
```

### Heap
```
Decr
  x -2
  Y 2
```

### Class Table
```
Object
  String toString() { ... }
  boolean equals...
  ...

Counter
  extends Object
  Counter() { x = 0; }
  void incBy(int d) {...}
  int get() {return x;}

Decr
  extends Counter
  Decr(int initY) { ... }
  void dec(){incBy(-y);}
```
Now use dynamic dispatch to invoke the `get` method for `d`. This involves searching up the class tree again...
After yet a few more steps...

Workspace

Stack

Heap

Class Table

Done! (Phew!)
Summary: this and dynamic dispatch

• When object’s method is invoked, as in `o.m()`, the code that runs is determined by `o`’s *dynamic* type.
  – The dynamic type, which is just a pointer to a class, is included in the object structure in the heap.
  – If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table.
  – This process is called *dynamic dispatch*.

• Once the code for `m` has been determined, a binding for `this` is pushed onto the stack.
  – The `this` pointer is used to resolve field accesses and method invocations inside the code.
Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

```java
public class C {
    public static int x = 23;
    public static int someMethod(int y) { return C.x + y; }
    public static void main(String args[]) {
        ...
    }
}

C.x = C.x + 1;
C.someMethod(17);
```

You can do a static assignment to initialize a static field.
Access to the static member uses the class name `C.x` or `C.foo()`
Class Table Associated with C

- The class table entry for C has a field slot for x.
- Updates to C.x modify the contents of this slot: \(C.x = 17\);

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>extends Object</td>
</tr>
<tr>
<td>static x 23</td>
</tr>
<tr>
<td>static int someMethod(int y)</td>
</tr>
<tr>
<td>{ return x + y; }</td>
</tr>
<tr>
<td>static void main(String args[])</td>
</tr>
<tr>
<td>{...}</td>
</tr>
</tbody>
</table>

- A static field is a *global* variable
  - There is only one heap location for it (in the class table)
  - Modifications to such a field are globally visible (if the field is public)
  - Use with caution!
Static Methods (Details)

- Static methods do not have access to the this pointer
  - Why? There isn’t an instance to dispatch through.
  - Therefore, static methods may only directly call other static methods.
  - Similarly, static methods can only directly read/write static fields.
  - Of course a static method can create instance of objects (via new) and then invoke methods on those objects.

- Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
  - e.g. o.someMethod(17) where someMethod is static
  - Eclipse will issue a warning if you try to do this.
Example of Statics

- The java.lang.Math library provides static fields/methods for many common arithmetic operations:

  - `Math.PI == 3.141592653589793`
  - `Math.sin`, `Math.cos`
  - `Math.sqrt`
  - `Math.pow`
  - etc.
Overriding
A Subclass can *Override* its Parent

```java
public class C {
    public void printName() { System.out.println("I’m a C"); }
}

public class D extends C {
    public void printName() { System.out.println("I’m a D"); }
}

C c = new D();
c.printName(); // what gets printed?
```

- Our model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But... can be confusing and difficult to reason about if not used carefully.
Dangers of Overriding

• Overriding the `test` method can cause the behavior of `printTest` to change!
  – Overriding can break invariants/abstractions relied upon by the superclass.
When To Override?

• Only override methods when the parent class is designed specifically to support such modifications:
  – If you’re writing the code for both the parent and child class (and will maintain control of both parts as the software evolves) it might be OK to override.
  – If the library designer specifically describes the behavioral contract that the parent methods assume about overridden methods (and the child follows that contract).
  – Either way: document the design.

• Look for other means of achieving the desired outcome:
  – Use composition & delegation (i.e. wrapper objects) rather than overriding.
The final modifier

• By default, fields and local variables are mutable and methods can be overridden*.

• The final modifier changes that.

• Final fields and local variables:
  – Must be initialized (either by a static initializer or in the constructor) and cannot thereafter be modified.
  – Act like the immutable name bindings in OCaml
  – static final fields are useful for defining constants (e.g. Math.PI)

• Final methods cannot be overridden in subclasses.
  – Also useful in combination with static
  – Prevents subclasses from changing the “behavioral contract” between methods by overriding.

*Technically, fields can also be re-declared in a subclass (i.e. C has field x and D extends C and also declares a field x, not even necessarily of the same type!). Don’t do this! But be aware that you can introduce bugs by inadvertently using this “feature”.
Aliasing and Mutation in Java

This is the same story as in OCaml. But, since mutability is the default, you must be extra careful in Java.
public class Node {
    public int elt;
    public Node next;
    public Node(int elt, Node next) {
        this.elt = elt;
        this.next = next;
    }
}

...
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;

- What is the final state of the stack & heap after running this program? (ignoring the workspace and class table)