Programming Languages and Techniques (CIS120e)

Lecture 26
Nov. 10, 2010

Java Abstract Stack Machine

The Java Abstract Stack Machine

1. Class tables
2. Constructors and “this”
3. Dynamic dispatch
4. Static members

Announcements

• Homework 8 (SpellChecker) is due Nov 15th.
• Midterm 2 is this Friday, November 12th

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.
**An Example**

```java
class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

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();
```

**Example with Explicit this and super**

```java
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; }
}

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**

```java
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

**Allocating Space on the Heap**

When a constructor is invoked:
- 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.

Assignment to a Field
Assignment 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.

### Abstract Stack Machine

- **Workspace**: this.y = 2;
- **Stack**: 
  - Counter
    - Counter() { x = 0; }  
    - void incBy(int d) {...}
  - int get() {return x;}
- **Heap**: 
  - Decr
    - extends Counter
      - Decr(int initY) { ...}
    - void dec(){incBy(-y);}
  - int x = 0;
  - this
  - initY = 2
  - x
  - y

### Assigning to a field

- **Workspace**: this.y = 2;
- **Stack**: 
  - Counter
    - Counter() { x = 0; }  
    - void incBy(int d) {...}
  - int get() {return x;}
- **Heap**: 
  - Decr
    - extends Counter
      - Decr(int initY) { ...}
    - void dec(){incBy(-y);}
  - int x = 0;
  - this
  - initY = 2
  - x
  - y

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...)

### 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);}
  - int x = 0;
  - this
  - initY = 2
  - x
  - y

### 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);}
  - int x = 0;
  - this
  - initY = 2
  - x
  - y
Done with the call

Represented by the diagram:

1. **Workspace**: Empty.
2. **Stack**:
   - `Decr d = new Decr(initY);` (Create a new `Decr` object with `initY` as parameter.

3. **Heap**:
   - `d`: An object reference to the newly created `Decr`.
   - `this`: A reference to the `Decr` object.
   - `initY`: An integer initialized to 2.

4. **Class Table**:
   - `Decr`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x` by the `d`'s `initY`.
     - `get()`: Get the `x`.
   - `Counter`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x`.
     - `get()`: Get the `x`.

5. **Done with the call**:
   - The call to the `Decr` constructor is complete. The stack is now empty, and the heap contains the `Decr` object with `initY` initialized.

6. **Continue executing the program**:
   - The `incBy(-y)` method is called, which increments `x` by `-initY`.
   - The `toString(...)` method is called, printing the object's string representation.

---

Allocating a local variable

Represented by the diagram:

1. **Workspace**: Empty.
2. **Stack**:
   - `g.dec(); int x = d.get();` (Allocate a stack slot for local variable `d`.

3. **Heap**:
   - `d`: An object reference to the `Decr` object.
   - `this`: A reference to the `Decr` object.
   - `initY`: An integer initialized to 2.

4. **Class Table**:
   - `Decr`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x` by the `d`'s `initY`.
     - `get()`: Get the `x`.
   - `Counter`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x`.
     - `get()`: Get the `x`.

5. **Allocate a local variable**:
   - A new `Decr` object is allocated on the stack.
   - `x` is initialized to the `get()` method of `d`.

6. **Aside**:
   - Since fields are mutable by default, we may omit the bold box.

---

Dynamic Dispatch: Finding the Code

Represented by the diagram:

1. **Workspace**: Empty.
2. **Stack**:
   - `dec();` (Invoke the `dec` method on the `Decr` object.)

3. **Heap**:
   - `d`: An object reference to the `Decr` object.
   - `this`: A reference to the `Decr` object.
   - `initY`: An integer initialized to 2.

4. **Class Table**:
   - `Decr`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x` by the `d`'s `initY`.
     - `get()`: Get the `x`.
   - `Counter`:
     - `toString()`: String representation.
     - `equals(...)`: Equality check.
     - `incBy(...)`:
       - Increment the `x`.
     - `get()`: Get the `x`.

5. **Invoke the dec method on the object**:
   - Look up the `dec` method in the `Decr` class.

6. **This process is called dynamic dispatch**:
   - The code to be executed is determined at runtime.

---

CSI120 / Fall 2010
Dynamic Dispatch: Finding the Code

Call the method, remembering the current workspace and pushing the this pointer and any arguments (none in this case).

Call this.incBy(-this.y);

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

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

workspace
stack
heap
class table
This incBy(-this.y);

Decr
d
x 0
this
y 2

workspace
stack
heap
class table

Workspace
Stack
Heap
Class Table

Lookup this

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.

workspace
stack
heap
class table

Dynamic Dispatch, Again

workspace
stack
heap
class table

CIS120e / Fall 2010

CIS120e / Fall 2010

CIS120e / Fall 2010

CIS120e / Fall 2010
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

After a few more steps...
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...
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);
}
```

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;

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

- 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`<code>== 3.141592653589793</code>
  - `Math.sin`, `Math.cos`
  - `Math.sqrt`
  - `Math.pow`
  - etc.
**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**

```java
public class C {
    public void printTest() {
        if (test()) { System.out.println("passed"); }
        else { System.out.println("failed"); }
    }
    public boolean test() { return true; }
}

public class D extends C {
    public boolean test() { return false; }
}

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

- 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”.

Example List Code

```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;
- n2.next.next = n2;

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

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.