Programming Languages
and Techniques
(CIS120)

Lecture 26
March 23, 2016

Inheritance and Dynamic Dispatch
Chapter 24
Inheritance Example

```java
public 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();
```

What is the value of x at the end of this computation?

1. -2
2. -1
3. 0
4. 1
5. 2
6. NPE
7. Doesn't type check

Answer: -2
Announcements

• Exam grades will be available (late) Friday

• Homework 6 available, due Tuesday
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 times2(Counter c) {
    c.incBy(c.get());
}
// somewhere else
C.times2(new Decr(3));
```

• 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
The Object Class
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 if implicit)
- Interface types (and arrays) are subtypes “by fiat” of Object
When do constructors execute?
How are fields accessed?
What code runs in a method call?
Revenge of the Son of the Abstract Stack Machine
How do method calls work?

- What code gets run in a method invocation?
  
  \texttt{o.move(3,4);}

- When that code is running, how does it access the fields of the object that invoked it?
  
  \texttt{x = x + dx;}

- When does the code in a constructor get executed?

- What if the method was inherited from a superclass?
ASM refinement: The Class Table

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

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

The class table contains:
- the code for each method,
- references to each class’s parent, and
- the class’s static members.
• Inside a non-static method, the variable `this` is a reference to the object on which the method was invoked.

• References to local fields and methods have an implicit “this.” in front of them.

class C {
    private int f;
    public void copyF(C other) {
        this.f = other.f;
    }
}
An Example

```java
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();
d.dec();
int x = d.get();
```
public class Counter extends Object {
    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();
d.dec();
int x = d.get();
Constructing an Object

Workspace

```java
Decr d = new Decr(2);
```

```java
d.dec();
```

```java
int x = d.get();
```

Stack

Heap

Class Table

```java
Object
String toString(){...}
boolean equals...
...

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

Decr
extends
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
Calling super

Call to super:
• The constructor (implicitly) calls the super constructor
• Invoking a method/constructor pushes the saved workspace, the method params (none here) and a new this pointer.

```java
super();
this.y = initY;
Decr d = _;
d.dec();
int x = d.get();
this
initY 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);}
```
Abstract Stack Machine

**Workspace**

```java
super();
this.x = 0;
```

**Stack**

```java
Decr d = _;
d.dec();
int x = d.get();
this.x = 0;
```

**Heap**

<table>
<thead>
<tr>
<th>Decr</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

**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.)
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.
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.
Continue in the `Decr` class’s constructor.
Concrete Stack Machine

**Workspace**

```
this.y = 2;
```

**Stack**

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

**Heap**

```
Decr

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**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);}
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 \texttt{Decr} constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the \texttt{this} pointer).
Decr d = __;  
d.dec();  
int x = d.get();

Continue executing the program.
Allocate a stack slot for the local variable `d`. Note that it’s mutable... (bold box in the diagram).

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

Invoke the `dec` method on the object. The code is found by “pointer chasing” through the class hierarchy.

This process is called *dynamic dispatch*: Which code is run depends on the dynamic class 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).
Decr extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}

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

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

Heap
Decr
x 0
y 2
this

Stack
d
int x = d.get();
this

Workspace
this.incBy(-y);
Dynamic Dispatch, Again

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

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

It takes a few steps...

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

Workspase: `this.x = this.x + d;`

Stack:
- `d`
- `int x = d.get();`
- `this`
- `_;
- `d -2`
- `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);}`
After a few more steps…

Workspace
int x = d.get();

Stack
d

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

Workspace

Stack

Heap

Class Table

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* class.
  - The dynamic class, represented as a pointer into the class table, 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 of *dynamic dispatch* is the heart of OOP!

- 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 & Java ASM
Based on your understanding of the ‘this’ parameter, is it possible to refer to ‘this’ in a static method?

1. No
2. Yes
3. I’m not sure
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()
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.
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;

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)
  - Generally not a good idea!
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.