Programming Languages and Techniques (CIS120)

Lecture 25
March 18, 2013

Subtyping and Dynamic Dispatch
Announcements

- HW07 due tonight at midnight
  - Weirich OH cancelled today
  - Help your TAs make the most effective use of OH
- HW08 (Text Adventure) is due March 25\textsuperscript{th} at 11:59:59pm
- \textit{Midterm 2 is Friday, March 29\textsuperscript{th} in class}
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in OCaml)
  - Arrays in (Java)
  - Subtyping & Inheritance (in Java)
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

Sharing declarations and definitions between related types
• Build rich interface hierarchies by extending existing interfaces.

```java
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.
• 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.
Interface Extension Demo

See: Main1.java
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, and may include additional fields or methods.

```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); }
}
```
Simple Inheritance

• In *simple inheritance*, the subclass only *adds* new fields or methods

• Use simple inheritance to *share common code* among related classes
  – Example: Point, Circle, and Rectangle have *identical* code for `getX()`, `getY()`, and `move()` methods when implementing Displaceable
  – Share this common code in a class “DisplaceableImpl”. The classes Point, Circle, Rectangle should inherit fields and methods from this class (see Main2.java)

• Inheritance 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
Subtyping with Inheritance

Classes

- Displaceable
- Area
- Shape
- Circle
- Rectangle
- Point

Interfaces

- DisplaceableImpl

- Extends
- Implements

- 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
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 (override these!)

• Object is also the top type of the subtyping hierarchy.
Subtyping

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object if implicit)
- Interface types are “subtypes by fiat” of Object
Example of Simple Inheritance

See: Main2.java
Inheritance: Constructors

- Constructors *cannot* be inherited (they have the wrong names!)
  - Instead, a subclass invokes the constructor of its super class using the keyword ‘super’.
  - Super *must* be the first line of the subclass constructor, unless the parent class constructor takes no arguments, in which it is OK to omit the call to super (it is called implicitly).

```java
class D {
    private int x;
    private int y;
    public D (int initX, int initY) {
        x = initX;
        y = initY;
    }
    public int addBoth() { return x + y; }
}

class C extends D {
    private int z;
    public C (int initX, int initY, int initZ) {
        super(initX, initY);
        z = initZ;
    }
    public int addThree() { return (addBoth() + z); }
}
```
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 in 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 (see Main3.java).

*Especially avoid overriding.*
The Java Abstract Stack Machine

1. Class tables
2. Constructors and “this”
3. Dynamic dispatch
How do method calls work?

• What code gets run in a method invocation?
  
  \[ o.\text{move}(3,4); \]

• When that code is running, how does it access the fields of the object that invoked it?
  
  \[ x = x + dx; \]

• When does the code in a constructor get executed?

• What if the method was inherited from a superclass?
The class table contains:
  • The code for each method,
  • Back pointers to each class’s parent, and
  • The class’s static members.
The ‘this’ Reference

• Inside a non-static method, the variable this is a reference to the object itself.

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

```java
class C {
    private int f;

    public void copyF(C other) {
        this.f = other.f;
    }
}
```
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 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();
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>Decr (d = \text{new Decr}(2)); (d).dec(); (\text{int } x = d).get();</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Object</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>String toString() { ... }</td>
<td>boolean equals...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Counter</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>extends</td>
<td>Counter() { (x = 0;) }</td>
<td>void incBy(int (d)) {...}</td>
<td>int get() {return (x;)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Decr</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>extends</td>
<td>Decr(int init(y)) { ... }</td>
<td>void dec() {incBy(-(y));}</td>
<td></td>
</tr>
</tbody>
</table>
Allocating Space on the Heap

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

Workspace

```java
super();
this.y = initY;
```

Stack

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

Heap

```
Decr

\[
\begin{array}{c|c}
\text{x} & 0 \\
\text{this} & \\
\text{y} & 0 \\
\text{initY} & 2 \\
\end{array}
\]
```

Class Table

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

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

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

CIS120 / Spring 2013
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.

CIS120 / Spring 2013
Abstract Stack Machine

Workspace

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

Stack

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

```
this
```

```
initY 2
```

```
_; this.y = initY;
```

```
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);}
```
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.

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

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

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

Stack

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

Heap

```
Decr
x 0
this
Y 0
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);}
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...)

CIS120 / Spring 2013
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

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

Stack

Heap

Counter

extends Object

Counter()

void incBy(int d){...}

int get()

{return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}
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 often omit the bold boxes and just assume the contents can be modified.

CIS120 / Spring 2013