Programming Languages and Techniques (CIS120)

Lecture 26

March 28, 2014

Extension

Announcements

- HW08 (GUI Programming II) is due next Tuesday at 11:59:59pm
- *Midterm 2 is Friday, April 4th in class*
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in OCaml and Java)
  - Arrays (in Java)
  - Subtyping (in Java)
- Everything up through *today's lecture and HW08* is fair game

Anonymous Inner Classes

First-class functions via inner classes

```
public interface Fun {
   public String apply (String x);
}
```

```
public static StringList transform (Fun f, StringList pl) {
   if (pl.isNil()) {
     return new Nil();
   } else {
     return new Cons (f.apply(pl.hd()), transform(f, pl.tl()));
   }
}
```

```
StringList z = transform
                  (new Fun() {
                    public String apply (String x) {
                      return x.toUpperCase();
                    }},
                   x);
```

```java
interface Fun {
   public int apply(int x);
}

class A {
   Fun g = new Fun () {
      public int apply(int x) {
         return x + 1;
      }
   };

   int m() {
      int x = g.apply(1);
      int y = g.apply(x);
      return g.apply(y);
   }

}
```

What is the result of a method call to m?

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. NullPointerException
7. Infinite loop
```

```java
interface Fun {
   public int apply(int x);
}

class B {
   Fun g = new Fun () {
      public int apply(int x) {
         return x + 1;
      }
   };
   Fun h = new Fun () {
      public int apply(int x) {
         return x + 2;
      }
   };
   int m() {
     int x = g.apply(1);
     g = h;
     int y = g.apply(x);
     Fun k = g;
     return k.apply(y);
   }
}
```

What is the result of a method call to m?

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. NullPointerException
7. Infinite loop
```

```java
interface Fun {
   public int apply(int x);
}

class C {
    private int z = 1;

    Fun g = new Fun () {
       public int apply(int x) {
          z = z + 1;
          return z;
       }
    };

    int m() {
       int w = g.apply(1);
       return g.apply(w);
    }
 }
```

What is the result of a method call to m?

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. NullPointerException
7. Infinite loop
```

```java
class D {

   int m(final int z) {

        final int w = 1;

        Fun g = new Fun () {
          public int apply(int x) {
           return x + z + w;
          }
        };

        return g.apply(2);
    }

 }
```

What is the result of a method call m(1)?

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. NullPointerException
7. Infinite loop
```

Anonymous Inner class

- New *expression* form: define a class and create an object from it all at once

keyword "new"

```
new InterfaceOrClassName() {
    public void method1(int x) {
        // code for method1
    }
    public void method2(char y) {
        // code for method2
    }

}
```

Normal class definition, no constructors allowed

Static type of the expression is the Interface/superclass used to create it

Dynamic class of the created object is anonymous! Can't really refer to it.

Inner Classes

- Useful in situations where two objects require "deep access" to each other's internals
- Replaces tangled workarounds like "owner objects"
  - Solution with inner classes is often easier to read
  - No need to allow public access to instance variables of outer class
- Anonymous inner classes can only refer to variable stored on the *stack* marked **final**
  - class instance variables are stored in the heap

Like first-class functions

- Anonymous inner classes are the Java equivalent of Ocaml first-class functions
- Both create "delayed computation" that can be stored in a data structure and run later
  - Code stored by the event / action listener
  - Code only runs when the button is pressed
  - Could run once, many times, or not at all
- Both sorts of computation can refer to variables in the current scope
  - Java: only instance variables (fields) and variables marked final
  - OCaml: Any available variable (all stack variables are final)

First class functions for real!

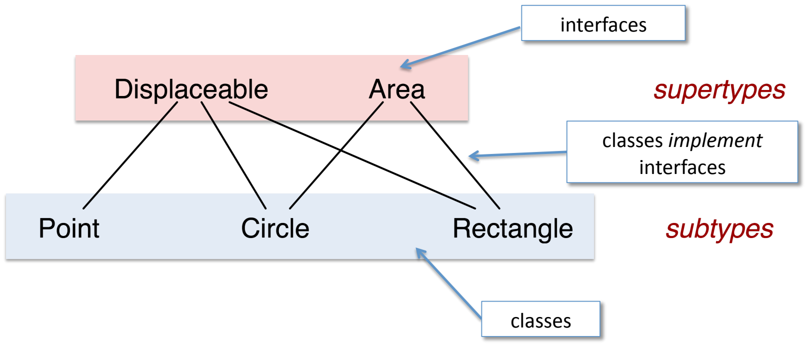
- The recent Java 8 release includes better notation for first-class functions, including anonymous abstractions
- :-)

Extension

Sharing code

between related types

Interfaces and Subtyping



Types can have many different supertypes / subtypes

Where do these classes and interfaces come from?
Can we make it easier to define them?

Interface Extension

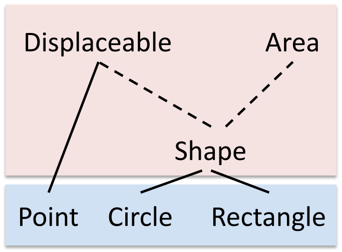
- Build rich interface hierarchies by *extending* existing interfaces.

```
public interface Displaceable {
  int getX();
  int getY();
  void move(int dx, int 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 “extends” keyword.

Interface Hierarchy



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

- 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.

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 and methods.

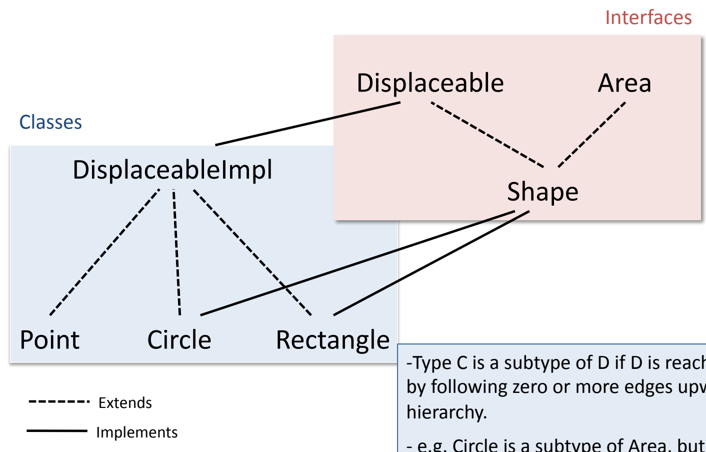
```
public class Counter {
  private int x;
  public Counter () { x = 0; }
  public void incBy(int d) { x = x + d; }
  public int get() { return x; }
}

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

Same “extends” keyword, different form of extension

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
- 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



-Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.

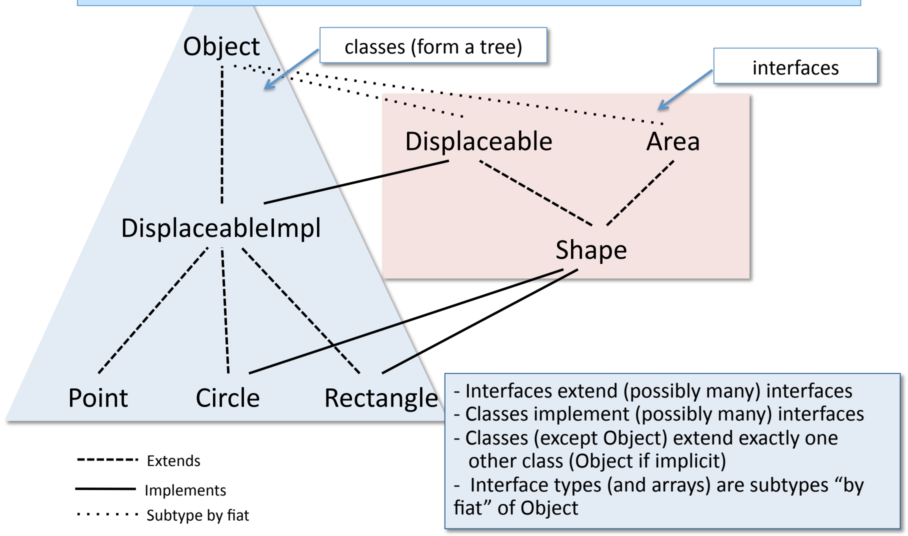
- e.g. Circle is a subtype of Area, but Point is not

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.

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

Inheritance: Constructors

- Contructors *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).

```
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.

  Especially avoid overriding.