Programming Languages and Techniques (CIS120)

Lecture 22
March 22, 2019

Java: Objects, Interfaces, Static Members
Chapters 19 & 20
Announcements

• HW06: Pennstagram
  – Available very soon
  – Due: Tuesday, April 2 at 11:59:59pm
  – Java programming
  – We encourage using Eclipse (TAs will do a walkthrough in recitation);
    You can use Codio if you prefer

Midterm 2: Friday, March 29 during lecture time.

• Midterm Course Survey
  – Look for a piazza post.
Object Oriented Programming
The OO Style

• Core ideas:
  – Objects: state encapsulated with operations
  – Dynamic dispatch: “receiver” of method call determines behavior
  – Classes: “templates” for object creation
  – Subtyping: grouping object types by common functionality
  – Inheritance: creating new classes from existing ones
OO programming

OCaml (part we've seen)

• Explicitly create objects using a record of higher order functions and hidden state

Java (and C, C++, C#)

• Primitive notion of object creation (classes, with fields, methods and constructors)

• Flexibility through composition: objects can only implement one interface

• Flexibility through extension: Subtyping allows related objects to share a common interface

```ocaml
type button =
  widget *
  label_controller *
  notifier_controller
```

```java
class Button extends Widget {
  /* Button is a subtype of Widget */
}
```
• **Object**: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*

• **Class**: a template for creating objects

• The class of an object specifies...
  – the types and initial values of its local state (*fields*)
  – the set of operations that can be performed on the object (*methods*)
  – one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)

• Every (Java) object is an *instance* of some class
  – Instances are created by invoking a constructor with the `new` keyword
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
    public int dec () {
        r = r - 1;
        return r;
    }
}

public class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println(c.inc());
    }
}

Objects in Java

CIS120
Encapsulating local state

```java
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
    public int dec () {
        r = r - 1;
        return r;
    }
}
```

Constructor and methods can refer to `r`.

```java
public class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println( c.inc() );
    }
}
```

Other parts of the program can only access public members.
Encapsulating local state

• *Visibility modifiers* make the state local by controlling access

• Basically:
  – public: accessible from anywhere in the program
  – private: only accessible inside the class

• Design pattern — first cut:
  – Make *all* fields private
  – Make constructors and non-helper methods public

(Java offers a couple of other protection levels — “protected” and “package protected”. The details are not important at this point.)
What is the value of \textit{ans} at the end of this program?

When poll is active, respond at \url{PollEv.com/120fall18}

Text \texttt{120FALL18} to \texttt{22333} once to join

```java
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
```

Counter x;
x.inc();
\textbf{int} ans = x.inc();
What is the value of ans at the end of this program?

```
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
Counter x;
x.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Program raises NullPointerException

Answer: Program raises NullPointerException
What is the value of `ans` at the end of this program?

```java
public class Counter {

    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }
}

Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

Program raises `NullPointerException`
What is the value of ans at the end of this program?

```java
public class Counter {
    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }
}

Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Program raises NullPointerException

Answer: 3
“Objects” in OCaml vs. Java

(* The type of “objects” *)

type point = {
  getX : unit -> int;
  getY : unit -> int;
  move : int*int -> unit;
}

(* Create an "object" with hidden state: *)
type position = {
  mutable x : int;
  mutable y : int;
}

let new_point () : point =
  let r = {x = 0; y=0} in {
    getX = (fun () -> r.x);
    getY = (fun () -> r.y);
    move = (fun (dx,dy) ->
      r.x <- r.x + dx;
      r.y <- r.y + dy)
  }

public class Point {
  private int x;
  private int y;

  public Point () {
    x = 0;
    y = 0;
  }

  public int getX () {
    return x;
  }

  public int getY () {
    return y;
  }

  public void move (int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
}

Type is separate from the implementation

Class specifies both type and implementation of object values
Which programming style is the best?

1. Functional
2. Imperative
3. Object-oriented
4. To every thing there is a season...
Interfaces

Working with objects abstractly
Interfaces

• Give a *type* for an object based on how it can be *used*, not on how it was *constructed*

• Describe a *contract* that objects must satisfy

• Example: Interface for objects that have a position and can be moved

```java
public interface Displaceable {
    public int getX();
    public int getY();
    public void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!
Implementing the interface

- A class that *implements* an interface provides appropriate definitions for the methods specified in the interface.
- The class fulfills the contract implicit in the interface.

```java
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```
Another implementation

```java
public class Circle implements Displaceable {
    private Point center;
    private int radius;
    public Circle(Point initCenter, int initRadius) {
        center = initCenter;
        radius = initRadius;
    }
    public int getX() { return center.getX(); }
    public int getY() { return center.getY(); }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
}
```

Delegation: move the circle by moving the center

Objects with different local state can satisfy the same interface
Another implementation

```java
class ColoredPoint implements Displaceable {
    private Point p;
    private Color c;
    ColoredPoint (int x0, int y0, Color c0) {
        p = new Point(x0,y0);
        c = c0;
    }
    public void move(int dx, int dy) {
        p.move(dx, dy);
    }
    public int getX() { return p.getX(); }
    public int getY() { return p.getY(); }
    public Color getColor() { return c; }
}
```

Flexibility: Classes may contain more methods than interface requires
Interfaces are types

• Can declare variables of interface type

```java
void m(Displaceable d) { ... }
```

• Can call m with any Displaceable argument...

```java
obj.m(new Point(3,4));
obj.m(new ColoredPoint(1,2,Color.Black));
```

• ... but m can only operate on d according to the interface

```java
d.move(-1,1);
...
... d.getX() ...  ⇒  0
... d.getY() ...  ⇒  3
```
Using interface types

- Interface variables can refer *dynamically*, i.e. during execution, to objects of any class implementing the interface.
- Point, Circle, and ColoredPoint are all *subtypes* of Displaceable.

```java
Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2,3), 1);
d2 = new ColoredPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
d2.move(-2,0);
...
... d0.getX() ... ⇒ -1
... d1.getX() ... ⇒ 0
... d2.getX() ... ⇒ -3
```

The class that created the object value determines which `move` code is executed: *dynamic dispatch*.
Abstraction

- The interface gives us a single name for all the possible kinds of “moveable things.” This allows us to write code that manipulates arbitrary Displaceable objects, without caring whether it’s dealing with points or circles.

class DoStuff {
    public void moveItALot (Displaceable s) {
        s.move(3,3);
        s.move(100,1000);
        s.move(1000,234651);
    }

    public void dostuff () {
        Displaceable s1 = new Point(5,5);
        Displaceable s2 = new Circle(new Point(0,0),100);
        moveItALot(s1);
        moveItALot(s2);
    }
}
Multiple interfaces

• An interface represents a point of view
  ...but there can be multiple valid points of view

• Example: Geometric objects
  – All can move (all are Displaceable)
  – Some have Color (are Colored)
• Contract for objects that have a color
  – Circles and Points don’t implement Colored
  – ColoredPoints do

```java
public interface Colored {
    public Color getColor();
}
```
public class ColoredPoint implements Displaceable, Colored {

    Point center;
    private Color color;

    public Color getColor() {
        return color;
    }

    ...

}
**Datatypes** in Java

**OCaml**

```ocaml
type shape =
  | Point of ...
  | Circle of ...

let draw_shape (s:shape) =
  begin match s with
  | Point ... -> ...
  | Circle ... -> ...
  end
```

**Java**

```java
interface Shape {
    public void draw();
}

class Point implements Shape {
    ...
    public void draw() {
        ...
    }
}

class Circle implements Shape {
    ...
    public void draw() {
        ...
    }
}
```
Recap

• **Object**: A collection of related *fields* (or *instance variables*) and *methods* that operate on those fields

• **Class**: A template for creating objects, specifying
  – types and initial values of fields
  – code for methods
  – optionally, a *constructor* that is run each time a new object is created from the class

• **Interface**: A “signature” for objects, describing a collection of methods that must be provided by classes that *implement* the interface

• **Object Type**: Either a class or an interface (meaning “this object was created from a class that implements this interface”)
Static Methods and Fields

functions and global state
Java Main Entry Point

```java
class MainClass {
    public static void main (String[] args) {
        ...
    }
}
```

- Program starts running at `main`
  - `args` is an array of `Strings` (passed in from the command line)
  - must be public
  - returns `void` (i.e. is a command)

- What does `static` mean?
How familiar are you with the idea of "static" methods and fields?

1. I haven't heard of the idea of "static".
2. I've used "static" before without really understanding what it means.
3. I have some familiarity with the difference between "static" and "dynamic".
4. I totally get it.
Static method example

```java
public class Max {
    public static int max (int x, int y) {
        if (x > y) {
            return x;
        } else {
            return y;
        }
    }

    public static int max3(int x, int y, int z) {
        return max(max(x,y), z);
    }
}
```

```java
public class Main {
    public static void main (String[] args) {
        System.out.println(Max.max(3,4));
        return;
    }
}
```

- Closest analogue of top-level functions in OCaml, but must be a member of some class.
- Internally (within the same class), call with just the method name.
- Main method must be static; it is invoked to start the program running.
- Externally, prefix with name of the class.
mantra

Static == Decided at *Compile Time*
Dynamic == Decided at *Run Time*