CSC547: Type Systems for OO Languages

Object-based languages

Abadi and Cardelli, Chapter 4

Objects without classes

Delegation

Embedding

Other stuff
Objects without classes

In Java we can create ‘classless objects’:

```java
interface Cell {
    int get ();
    void set (int n);
}

Cell myCell = new Cell () {
    int contents = 0;
    int get () { return this.contents; }
    void set (int n) { this.contents = n; }
}
```

These are called *anonymous inner classes* in Java.

What would Java be like if it had no classes, only interfaces and objects?

This is the topic of *object-based* (rather than *class-based*) languages.
Objects without classes

Even without classes, we can simulate constructors using functions:

```java
Cell newCell (int m) {
    return new Cell () {
        int contents = m;
        int get () { return this.contents; }
        void set (int n) { this.contents = n; }
    }
}

Cell myCell = newCell (37);
```

How would we do the classic Point example?

So why not do away with classes altogether?
Prototypes and clones

Some languages use prototyping for replacing constructors:

Cell prototypeCell = new Cell () {
    int contents = 0;
    int get () { return this.contents; }
    void set (int n) { this.contents = n; }
}

Cell newCell (int m) {
    Cell result = clone prototypeCell;
    result.set (m);
    return result;
}

This uses shallow cloning with clone.

How would we do the classic Point example now?
Prototypes and clones

We need some way to do inheritance:

```java
interface ReCell extends Cell {
    void restore ();
}

ReCell prototypeReCell = new ReCell () {
    int contents = 0;
    int backup = 0;
    int get () {
        return this.contents;
    }
    void set (int n) {
        this.backup = this.contents;
        this.contents = n;
    }
    void restore () {
        this.contents = this.backup;
    }
}

ReCell newReCell () {
    return clone prototypeReCell;
}
```

What is wrong with this picture?

How would we do the classic ColorPoint example now?
Explicit embedding

One way to simulate inheritance:

```java
ReCell prototypeReCell = new ReCell () {
    int contents = prototypeCell.contents;
    int backup = 0;
    int get () = prototypeCell.get;
    void set () {
        this.backup = this.contents;
        embed prototypeCell.get ();
    }
    void restore () { this.contents = this.backup; }
};

How would we do ColorPoint now?
```
Implicit embedding

Another way to simulate inheritance:

```java
ReCell prototypeReCell extends prototypeCell {
    int backup = 0;
    void set () {
        this.backup = this.contents;
        embed prototypeCell.get ();
    }
    void restore () { this.contents = this.backup; }
};
```

Note this is a lot like the equivalent class declaration!

How would we do ColorPoint now?
Embedding with method override

Yet another way to simulate inheritance:

```java
ReCell prototypeReCell extends prototypeCell {
   int backup = 0;
   void restore () { this.contents = this.backup; }
};
prototypeReCell.set = method () {
   this.backup = this.contents;
   embed prototypeCell.get ();
};
```

This is a lot like ML programming with first-class functions.

How would we do ColorPoint now?
Embedding with method override

Even yet another way to simulate inheritance:

```java
ReCell prototypeReCell extends prototypeCell {
    void restore () { }
}
prototypeReCell.set = method (int n) {
    int current = this.contents;
    this.contents = n;
    this.restore = method () { this.contents = current; }
}

Very weird!
```
Delegation

Delegation is like embedding, but includes a *pointer* to the parent object, rather than copying it.

```java
ReCell prototypeReCell
childof prototypeCell {
    int backup = 0;
    void set () {
        this.backup = this.contents;
        delegate prototypeCell.get ();
    }
    void restore () { this.contents = this.backup; }
};
ReCell newReCell (int n) {
    ReCell result = clone prototypeReCell;
    result.set (n);
    return result;
}
```

What is the difference between this and embedding?

What goes wrong here?

```java
ReCell aReCell = newReCell (5);
ReCell bReCell = newReCell (9);
print (aReCell.get ());
```

How do we do ColorPoint now?
Delegation

Fields need to be treated differently from methods!

```java
ReCell prototypeReCell
childof prototypeCell {
    int contents = prototypeCell.contents;
    int backup = 0;
    void set () {
        this.backup = this.contents;
        delegate prototypeCell.get ();
    }
    void restore () { this.contents = this.backup; }
}
ReCell newReCell (int n) {
    ReCell result = clone prototypeReCell;
    result.set (n);
    return result;
}
```

Now what happens?

```java
ReCell aReCell = newReCell (5);
ReCell bReCell = newReCell (9);
print (aReCell.get ());
```

How do we do ColorPoint now?

What are the tradeoffs between embedding and delegation?
Dynamic inheritance

In a delegation model, objects have pointers to their parent.

Should this pointer be fixed (final in Java terminology)?

Some languages allow mode switching (change the parent methods but not fields).
Traits

In some object-based languages, we distinguish between:

- Normal objects
- Prototype objects (are cloned to create normal objects)
- Traits (containing method suites)

We’re back to objects and classes again!