Semantics of Objects

17-363/17-663: Programming Language Pragmatics

Reading: PLP chapter 10

Ben Titzer

Prof. Jonathan Aldrich
Object-Oriented Programming (OOP)

Three key aspects:

• Encapsulation
  • An object is a grouping of state and behavior, and hides its implementation choices from the outside world

• Inheritance
  • Objects are related, and we can capture shared behavior in a way that multiple kinds of objects can use it without defining it themselves

• Dynamic dispatch
  • The same operation can be implemented in different ways; each object knows what implementation to use for each of its operations

This aspect is special—it’s the only one present in ALL object-oriented languages
Encapsulation

• An object is a grouping of state and behavior

```javascript
let setImpl = {
    members: [1, 2, 3],
    isMember: function(x) {
        return this.members.includes(x);
    },
    add: function(x) {
        if (!isMember(x))
            this.members.push(x);
    }
};
setImpl.add(4); // uses the object
setImpl.isMember(4); // returns true
```
• We can hide some of the object’s state

```typescript
interface IntSet {
    isMember : (x: number) => boolean
    add : (x: number) => void
}

let setImpl = { ... };
let set : IntSet = setImpl;
set.add(4);
set.isMember(4);
set.members
```

interface `IntSet` leaves out the members field. We can change that later without affecting clients.

Assigning to a variable of type `IntSet` hides everything that’s not in the interface. It’s a type error to access members that are not exposed in the interface.
Classes

• A class is a template for objects. It defines structure & behavior used by all instances of the class

```javascript
class IntSetClass {
  members : number[];
  constructor(m:number[]) {
    this.members = m;
  }
  isMember(x:number):boolean {
    return this.members.includes(x);
  }
  // add(x:number):void {
  ...
}

let set2 : IntSetClass = new IntSetClass([1, 2]);
set2.add(5);
set2.isMember(5); // returns true
```
Dynamic Dispatch

• Every object knows its method implementations (whether defined in the object, or in that object’s class)
• When we invoke a method, the code for that object is run

class Dog {
  talk() { console.log("woof!"); }
}
class Cat {
  talk() { console.log("meow!"); }
}
let animals = [new Dog(), new Cat()];
for (let a of animals)  
a.talk(); // prints woof! meow!
Inheritance

- Inheritance lets us reuse code from one class in another
  - Prototype: a variant where you reuse code from another object (see JavaScript)

```javascript
class Collection {
    constructor(ms) { this.members = ms; }
    isMember(x) { return this.members.includes(x); }
    add(x) { this.members.push(x); }
    addAll(a) { for (let x of a) this.add(x); }
}

class Set extends Collection {
    constructor(ms) { super(ms); }
    add(x) { if (!this.isMember(x)) { super.add(x); } }
}

let set = new Set([]);
set.add(3);
set.addAll([3, 4]);
set.isMember(4);
```
Exercise

- Draw the frames on the runtime stack when 4 is added to the set in the call set.addAll([3, 4]). Show all methods that are in from main() through push()

```javascript
class Collection {
    constructor(ms) { this.members = ms; }
    isMember(x) { return this.members.includes(x); }
    add(x) { this.members.push(x); }
    addAll(a) { for (let x of a) this.add(x); }
}

class Set extends Collection {
    constructor(ms) { super(ms); }
    add(x) { if (!this.isMember(x)) { super.add(x); } }
}
let set = new Set([]);
set.add(3);
set.addAll([3, 4]);
```
Why Objects Matter

- Encapsulation (not specific to objects)
  - Separate reasoning about a single module enhances correctness & finding bugs
  - Ability to change the internals of a module without affecting others enhances software evolution

- Inheritance
  - Some code patterns are difficult to reuse in any other way
    - Typically when you have a reusable part and a customizable part, and they both call each other
  - That said, many uses of inheritance can (and should) be replaced with composition
    - Common guideline: prefer composition to inheritance

- Dynamic dispatch
  - Architecturally important – support multiple independent & interoperating implementations of a common interface
  - Examples all over the place: mobile phone apps, Linux device drivers, graphical user interfaces, MapReduce, web frameworks
Semantics

• We can use the static and dynamic semantics techniques we have learned to model objects

Source: Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. OOPSLA 1999.