

CS61B Lecture #5: Objects: Instance Methods and Constructors

- Homework/Lab assignment handout today.
- Readings for Monday: *PIJ* We'll still be on *PIJ* 1.11 and 1.12! We'll get to 1.13 and 1.14 later in the week, so you might want to read ahead.
- Readings on language details: We've looked at Java by example. To fully understand it, you need a more detailed treatment. Here are readings that correspond to what we've read or covered in lecture so far, but that go into more detail:

PIJ 2.1-2.2, 5.1-5.5, 5.7-5.8, 6.1-6.3.2, 6.5-6.8, 7.1-7.5.

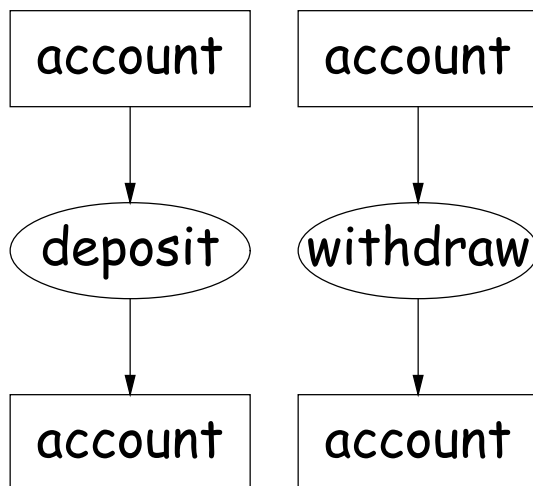
You should try to finish this reading as soon as possible—certainly by the end of next week. You're certain to have questions; feel free to ask!

Object-Based Programming

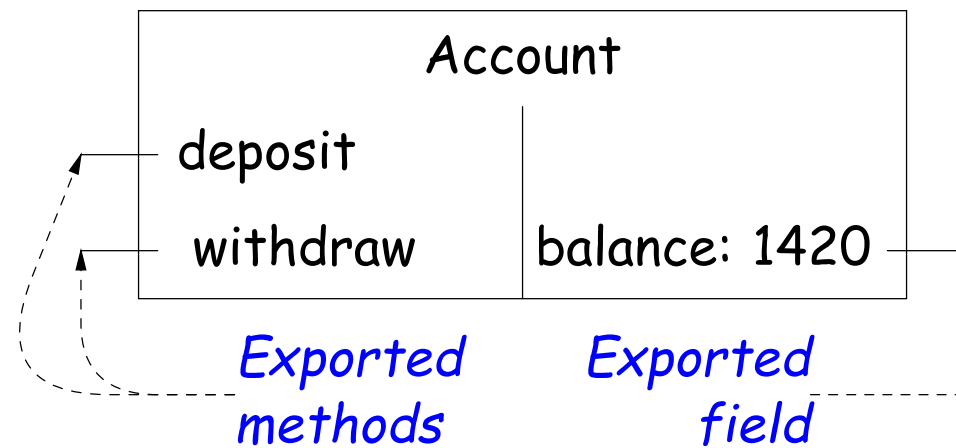
Basic Idea.

- *Function-based programs* are organized primarily around the functions (methods, etc.) that do things. Data structures (objects) are considered separate.
- *Object-based programs* are organized around the types of objects that are used to represent data; methods are grouped by type of object.
- Simple banking-system example:

Function-based



Object-based



Philosophy

- Idea (from 1970s and before): *An abstract data type is*
 - a set of possible values (a *domain*), plus
 - a set of *operations* on those values (or their containers).
- In `IntList`, for example, the domain was a *set of pairs*: `(head, tail)`, where `head` is an `int` and `tail` is a pointer to an `IntList`.
- The `IntList` operations consisted only of assigning to and accessing the two fields (`head` and `tail`).
- In general, prefer a purely *procedural interface*, where the functions (methods) do everything—no outside access to fields.
- That way, implementor of a class and its methods has complete control over behavior of instances.
- In Java, the preferred way to write the “operations of a type” is as *instance methods*.

You Saw It All in CS61A

- Here's the account class from CS61A, slightly modified to make it more compatible with Java:

```
(define-class (account balance0)
  (instance-vars (balance 0))
  (initialize
    (set! balance balance0))
  (method (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
        (error "Insufficient funds")
        (begin
          (set! balance (- balance amount))
          balance)))) )
```

**Class
Defn**

```
(define my-account (instantiate account 1000))
(ask my-account 'balance)
(ask my-account 'deposit 100)
(ask my-account 'withdraw 500)
```

Sample Use

And Here It Is in Java

```
public class Account {
    public int balance;
    public Account (int balance0) {
        balance = balance0;
    }
    public int deposit (int amount) {
        balance += amount; return balance;
    }
    public int withdraw (int amount) {
        if (balance < amount)
            throw new IllegalStateException ("Insufficient funds");
        else {
            balance -= amount; return balance;
        }
    }
}
```

```
Account myAccount = new Account (1000);
myAccount.balance
myAccount.deposit (100);
myAccount.withdraw(500);
```

The Pieces, part I

- Class declaration defines a *new type of object*, i.e., new type of structured container.
- **Instance variables** such as `balance` are the simple containers within these objects (*fields or components*).
- **Instance methods**, such as `deposit` and `withdraw` are like ordinary (static) methods that take an invisible extra parameter (called **this**).
- The **new** operator creates (*instantiates*) new objects, and initializes them using constructors.
- **Constructors** such as the method-like declaration of `Account` are special methods that are used only to initialize new instances. They take their arguments from the **new** expression.
- **Method selection** picks methods to call. For example,

```
myAccount.deposit(100)
```

tells us to call the method named `deposit` that is defined for the object pointed to by `myAccount`.

Getter Methods

- Slight problem with Java version of Account: anyone can assign to the balance field
- This reduces the control that the implementor of Account has over possible values of the balance.
- Solution: allow public access only through methods:

```
public class Account {  
    private int balance;  
    ...  
    public int balance () { return balance; }  
    ...  
}
```

- Now the balance field cannot be directly referenced outside of Account.
- (OK to use name balance for both the field and the method. Java can tell which is meant by syntax: A.balance vs. A.balance().)

Class Variables and Methods

- Suppose we want to keep track of the bank's total funds.
- This number is not associated with any particular Account, but is common to all—it is *class-wide*.
- In Java, "class-wide" \equiv static

```
public class Account {  
    ...  
    private static int funds = 0;  
    public int deposit (int amount) {  
        balance += amount; funds += amount;  
        return balance;  
    }  
    public static int funds () {  
        return funds;  
    }  
    ... // Also change withdraw.  
}
```

- From outside, can refer to either `Account.funds()` or `myAccount.funds()` (same thing).

Instance Methods

- Instance method such as

```
int deposit (int amount) {  
    balance += amount; funds += amount;  
    return balance;  
}
```

behaves sort of like a static method with hidden argument:

```
static int deposit (final Account this, int amount) {  
    this.balance += amount; funds += amount;  
    return this.balance;  
}
```

- NOTE: Just explanatory: Not real Java (not allowed to declare 'this'). (However, *final* is real Java; means "can't assign to.")
- Likewise, the instance-method call `myAccount.deposit (100)` is like a call on this fictional static method:

```
Account.deposit (myAccount, 100);
```

- Inside method, as a convenient abbreviation, can leave off leading 'this.' on field access or method call if not ambiguous.

'Instance' and 'Static' Don't Mix

- Since real static methods don't have the invisible `this` parameter, makes no sense to refer to instance variables in them:

```
public static int badBalance () {  
    return balance; // WRONG! NONSENSE!  
}
```

- Reference to `balance` here equivalent to `this.balance`,
- But this is meaningless (*whose balance?*)
- However, it makes perfect sense to access a static (class-wide) field or method in an instance method or constructor, as happened with `balance` in the `deposit` method.
- There's only one of each static field, so don't need to have a 'this' to get it.

Constructors

- To completely control objects of some class, you must be able to set their initial contents.
- A *constructor* is a kind of special instance method that is called by the **new** operator right after it creates a new object, as if

$$L = \text{new IntList}(1, \text{null}) \implies \begin{cases} \text{tmp} = \text{pointer to } \boxed{0}; \\ \text{tmp.IntList}(1, \text{null}); \\ L = \text{tmp}; \end{cases}$$

- Instance variables initializations are moved inside constructors:

```
class Foo {
    int x = 5;
    Foo () {
        DoStuff ();
    }
    ...
}

class Foo {
    int x;
    Foo () {
        x = 5;
        DoStuff ();
    }
    ...
}
```

- In absence of any explicit constructor, get *default constructor*:
`public Foo() { }.`
- *Multiple overloaded* constructors possible (different parameters).

Summary:

Java vs. CS61A OOP in Scheme

Java	CS61A OOP
class Foo ...	(define-class (Foo ...
int x = ...;	(instance-vars (x ...))
Foo(...) {...}	(initialize ...)
int f(...) {...}	(method (f ...) ...)
static int y = ...;	(class-vars (y ...))
static void g(...) {...}	(define (g...)...)
aFoo.f ...	(ask aFoo 'f ...)
new Foo (...)	(instantiate Foo ...)
this	self