CS61B Lecture #7

Announcements:

• Programming Contest coming up: 3 October. Watch for details.

• Computer Science Mentors (CSM) is holding adjunct sections that you can sign up for this semester. These are small groups of students, led by a trained mentor, that meet weekly and provide additional practice and guidance with course material. Sign-up deadline is Friday, 18 September (next week). See also Piazza post @520.

• Homework #2 was released late Wednesday.

• Project #0 will be released soon. Watch for it.
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:
    - account
      - deposit
      - withdraw
      - account
  
  - Object-based:
    - Account
      - deposit
      - withdraw
      - balance: 1420
      - export methods
      - export field
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: The Account Class

class Account:
    balance = 0
    def __init__(self, balance0):
        self.balance = balance0

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if self.balance < amount:
            raise ValueError("Insufficient funds")
        else:
            self.balance -= amount
        return self.balance

my_account = Account(1000)
my_account.balance
my_account.deposit(100)
my_account.withdraw(500)

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);
You Also Saw It All in CS61AS

```lisp
(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)))
)

(define my-account
  (instantiate account 1000))

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

```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

- **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,

  ```java
  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:

```java
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

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

  behaves sort of like a static method with hidden argument:
  
  ```java
  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'). (final is real Java; means “can’t change once set.”)

• Likewise, the instance-method call `myAccount.deposit (100)` is like a call on this fictional static method:

  ```java
  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 directly to instance variables in them:

```java
public static int badBalance (Account A) {
    int x = A.balance; // This is OK (A tells us whose balance)
    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 funds in the deposit method.

• There's only one of each static field, so don't need to have a 'this' to get it. Can just name the class.
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

\[
L = \text{new IntList}(1, \text{null}) \rightarrow \begin{cases} 
\text{tmp} = \text{pointer to } 0; \\
\text{tmp.} \text{IntList}(1, \text{null}); \\
L = \text{tmp};
\end{cases}
\]

• Instance variables initializations are moved inside constructors:

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

  \[
  \text{class Foo} \{ \\
    \text{int x;} \\
    \text{Foo} () \{ \\
      \text{x} = 5; \\
      \text{DoStuff} (); \\
    \}
    \text{...}
  \}
  \]

• In absence of any explicit constructor, get default constructor:

  \[
  \text{public Foo()} \{ \}.
  \]

• Multiple overloaded constructors possible (different parameters).
## Summary: Java vs. CS61A OOP in Scheme & Python

<table>
<thead>
<tr>
<th>Java</th>
<th>Python</th>
<th>CS61AS OOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Foo ...</td>
<td>class Foo: ...</td>
<td>(define-class (Foo args)...)</td>
</tr>
<tr>
<td>int x = ...;</td>
<td>x = ...</td>
<td>(instance-vars (x ...))</td>
</tr>
<tr>
<td>Foo(args) {...}</td>
<td>def <strong>init</strong>(self, args):...</td>
<td>(initialize ...)</td>
</tr>
<tr>
<td>int f(...) {...}</td>
<td>def f(self, ...):...</td>
<td>(method (f ...) ...)</td>
</tr>
<tr>
<td>static int y = ...;</td>
<td>y = ...</td>
<td>(class-vars (y ...))</td>
</tr>
<tr>
<td>static void g(...) {...}</td>
<td>(refer to with Foo.y)</td>
<td>(define (g...)...)</td>
</tr>
<tr>
<td>... or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aFoo.f (...)</td>
<td>@staticmethod def g(...): ...</td>
<td></td>
</tr>
<tr>
<td>aFoo.x</td>
<td>def g(...): [outside classes]</td>
<td></td>
</tr>
<tr>
<td>new Foo (...)</td>
<td>(ask aFoo 'f ...)</td>
<td></td>
</tr>
<tr>
<td>this</td>
<td>self [typically]</td>
<td>(instantiate Foo ...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>self</td>
</tr>
</tbody>
</table>