CS61B Lecture #7: 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:

```
<table>
<thead>
<tr>
<th>Function-based</th>
<th>Object-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>Account</td>
</tr>
<tr>
<td>deposit</td>
<td>deposit</td>
</tr>
<tr>
<td>withdraw</td>
<td>withdraw</td>
</tr>
<tr>
<td>balance: 1420</td>
<td>balance: 1420</td>
</tr>
<tr>
<td>Exported methods</td>
<td>Exported field</td>
</tr>
<tr>
<td>account</td>
<td>account</td>
</tr>
<tr>
<td>account</td>
<td>account</td>
</tr>
</tbody>
</table>
```
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.
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

myAccount = Account(1000)
print(myAccount.balance)
myAccount.deposit(100)
myAccount.withdraw(500)
You Also Saw It All in CS61AS

(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)

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,

  ```
  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 `Account.balance = 1000000` is an error outside `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()`. However, it’s probably better to choose differing names to avoid confusion.)
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 to `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.”)
Calling Instance Method

/** (Fictional) equivalent of deposit instance method. */
static int deposit(final Account this, int amount) {
    this.balance += amount; funds += amount;
    return this.balance;
}

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

• Inside a real instance method, as a convenient abbreviation, one can leave off the leading 'this.' on field access or method call if not ambiguous. (Unlike Python)
'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!
    // (Whose balance?)
}
```

• 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

\[
\begin{align*}
L & = \text{new IntList}(1, \text{null}) \implies \\
& \quad \{ \text{tmp = pointer to } \text{0} \} \\
& \quad \text{tmp.IntList}(1, \text{null}) \\
& \quad L = \text{tmp};
\end{align*}
\]
Multiple Constructors and Default Constructors

- **All** classes have constructors. In the absence of any explicit constructor, get **default constructor**, as if you had written:

  ```java
  public class Foo {
    public Foo() {
    }
  }
  ```

- **Multiple overloaded constructors** possible, and they can use each other (although the syntax is odd):

  ```java
  public class IntList {
    public IntList(int head, IntList tail) {
      this.head = head; this.tail = tail;
    }

    public IntList(int head) {
      this(head, null);  // Calls first constructor.
    }
    ...
  }
  ```
Constructors and Instance Variables

• Instance variables initializations are moved inside all constructors:

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

⇐ ⇒

class Foo {
    int x;
    Foo() {
        x = 5;
        DoStuff();
    }
}
## Summary: Java vs. Python

<table>
<thead>
<tr>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
</table>
| ```java
class Foo {
    int x = ...;
    Foo(...) {
        ...
    }
    int f(...) {
        ...
    }
    static int y = 21;
    static void g(...) {
        ...
    }
}
``` | ```python
class Foo: ...
x = ...
def __init__(self, ...):
    ...
def f(self, ...):
    ...
y = 21  # Referred to as Foo.y
@staticmethod
def g(...):
    ...
``` |
| aFoo.f(...)                                | aFoo.f(...)                                 |
| aFoo.x                                     | aFoo.x                                      |
| new Foo(...)                               | new Foo(...)                                |
| this                                       | this # (typically)                          |