CS 17700
Object Oriented Design

Week 12
Announcements

• Project 3 will be out on Wednesday 12\textsuperscript{th} and due on Dec. 3\textsuperscript{rd}
Table of Contents

- Overview of Object Oriented Programming (OOP) terminology
- Global and Shared data
- Procedural approach vs. Object oriented design
- Bank account system
Overview of OOP Terminology

- **Class**: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.
Overview of OOP

Terminology

**Data member:** A class variable or instance variable that holds data associated with a class and its objects.

**Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
Object: A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
Global and Shared data

We can see that one of the principle differences is that procedural systems make use of **shared** and **global data**, while object oriented systems lock their data privately away in objects.

Let's consider a scenario where you need to change a shared variable in a procedural system. Perhaps you need to rename it, change it from a string to a numeric, change it from a struct to an array, or even remove it completely.
Global and Shared data

In a procedural application you would need to find and change each place in the code where that variable is referenced. In a large system this can be a widespread and difficult change to make.

In an object oriented system we know that all variables are inside objects and that only functions within those objects can access or change those variables. When a variable needs to be changed then we only need to change the functions that access those variables.
Procedural approach
Object Oriented Design
Bank account System

- Suppose we want to model a bank account with support for deposit and withdraw operations.

- There are two approaches: Global state, Object oriented design.
Bank account System (Global state)

- One way to do that is by using global state as shown in the following example.

```python
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

`balance += amount` is equivalent to `balance = balance + amount`
Bank account System (Global state)

- The previous example (Global state) is good enough only if we want to have just a single account. Things start getting complicated if we want to model multiple accounts.
class BankAccount:
    def __init__(self, initial_balance):
        """Creates an account with the given balance."""
        self.money = initial_balance
        self.penalty = 0

    def deposit(self, amount):
        """Deposits the amount into the account."""
        self.money += amount
        return self.money

    def withdraw(self, amount):
        """
        Withdraws the amount from the account. Each withdrawal resulting in a
        negative balance also deducts a penalty fee of 5 dollars from the balance.
        """

        if self.money - amount < 0:
            self.money -= amount+5
            self.penalty += 5
        else:
            self.money -= amount
        return self.money

    def get_balance(self):
        """Returns the current balance in the account."""
        return self.money

    def get_fees(self):
        """Returns the total fees ever deducted from the account."""
        return self.penalty
Bank account System (OOP)

```python
Python 3.2 (r32:88445, Feb 20 20
32
Type "copyright", "credits" or 
>>> ============
>>> account1 = BankAccount(10)
>>> account2 = BankAccount(20)
>>> account1.deposit(30)
40
>>> account2.deposit(40)
60
>>> account1.withdraw(10)
30
>>> account2.withdraw(20)
40
>>> account1.withdraw(40)
-15
>>> |
Inheritance

• Now we use inheritance to provide an **InterestAccount** that adds interest (we'll assume 3%) on every deposit. It will be identical to the standard BankAccount class except for the deposit method. So we simply override that.

• **ChargingAccount** is again identical to a standard BankAccount class except that this time it charges $3 for every withdrawal. As for the InterestAccount we can create a class inheriting from BankAccount and modifying the withdraw method.
class InterestAccount(BankAccount):
    def deposit(self, amount):
        BankAccount.deposit(self, amount)
        self.money = self.money * 1.03
        return self.money

class ChargingAccount(BankAccount):
    def __init__(self, initialAmount):
        BankAccount.__init__(self, initialAmount)
        self.fee = 3

    def withdraw(self, amount):
        BankAccount.withdraw(self, amount+self.fee)
        return self.money
Inheritance

```python
>>> Interest_AC1 = InterestAccount(100)
>>> Interest_AC1.deposit(100)
206.0
>>> Chargin_AC1 = ChargingAccount(100)
>>> Chargin_AC1.withdraw(10)
87
>>> Interest_AC1.get_balance()
206.0
>>> Chargin_AC1.get_balance()
87
```
References

- http://www.codeskulptor.org
- http://objectorientedcoldfusion.org/
ANY QUESTIONS?