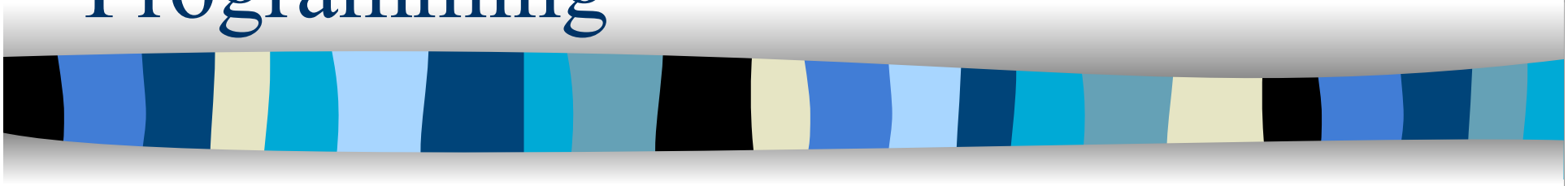
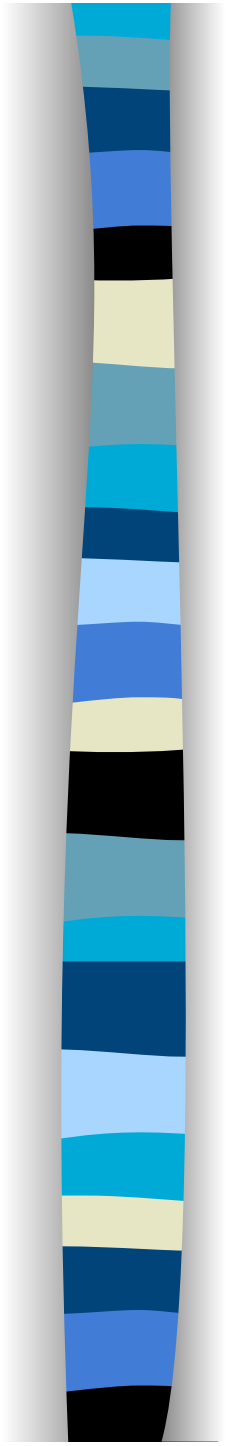
IS12 - Introduction to Programming Lecture 11: Object-Oriented Programming



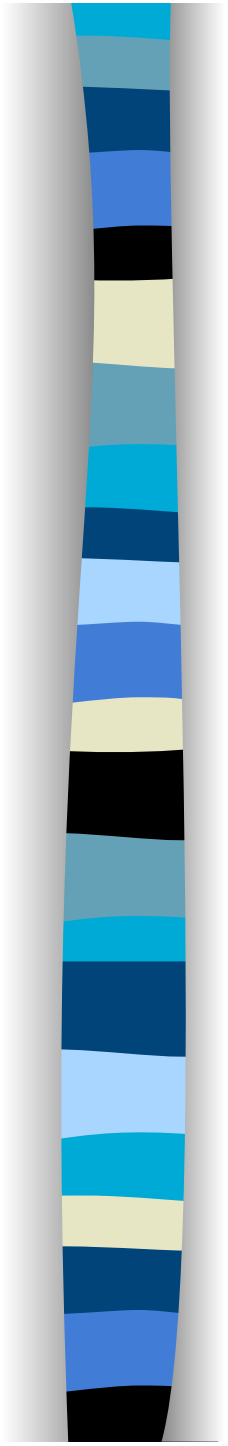
Peter Brusilovsky

http://www.pitt.edu/~peterb/0012-161/



Outline

- Encapsulation
  - Hiding, get_ and set_
- Object evaluation - __str__
- Data processing with classes and objects
- Inheritance

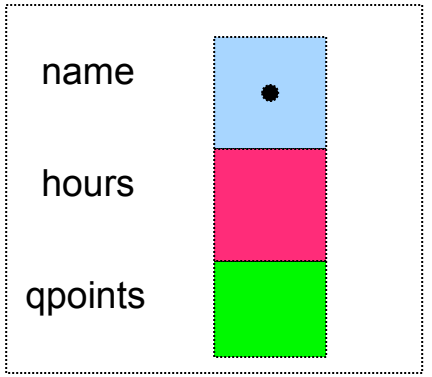


Why Objects and OOP?

- Heterogeneous data aggregates
- Encapsulation of actions on data objects
- Inheritance and more efficient programming
- More efficient GUI programming

Example: Max GPA: Data

```
class Student:
# name - student name
# hours - credit hours
# qpoints quality points
```




```
class Student:
  def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = float(hours)
        self.qpoints = float(qpoints)
```

Example: Max GPA: Methods

```
class Student:
  def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = hours
        self.qpoints = qpoints

  def gpa(self):
        if self.hours > 0:
         return self.qpoints/self.hours
        else:
         return 0
```

Example: Max GPA: Functions

```
def makeStudent(infoStr):
    # infoStr is a tab-separated line:
    # name hours qpoints
    # returns a corresponding Student object

    name, hours, qpoints = infoStr.split("\t")
    s = Student(name, float(hours), \
  float(qpoints))
    return s
```

Example: Max GPA: Main

```
def main():
    # open the input file for reading
    filename = input("Enter name the grade file: ")
    infile = open(filename, 'r')

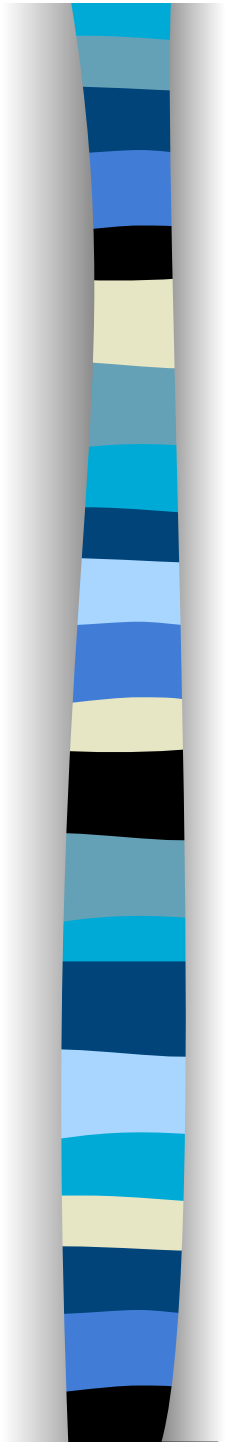
    # set best to the record for the
    # first student in the file
    best = makeStudent(infile.readline())

    # process subsequent lines of the file
    for line in infile:
        # turn the line into a student record
        s = makeStudent(line)
        # if this student is best so far, remember it.
        if s.gpa() > best.gpa():
            best = s
```

Example: Max GPA: Main (cont)

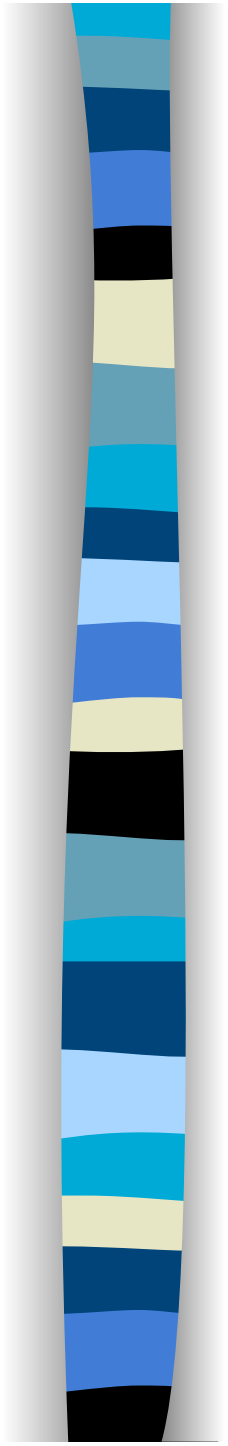
```
 infile.close()

# print information about the best
print("The best student is:", best.name)
print("hours:", best.hours)
print("GPA:", best.gpa())
```



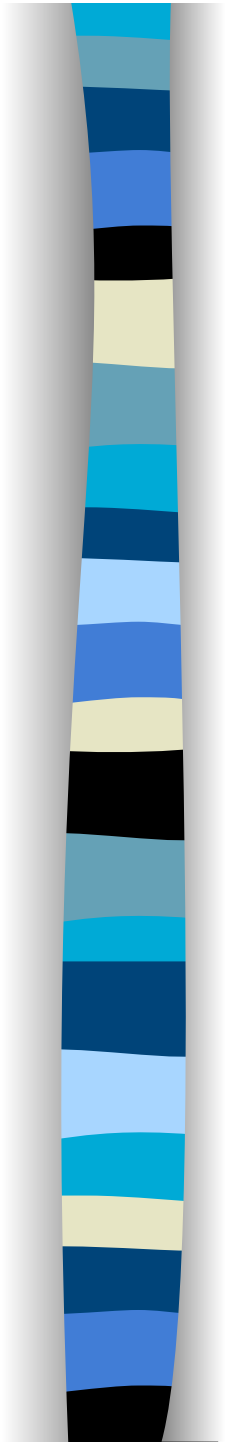
Why Objects and OOP?

- Heterogeneous data aggregates
- Encapsulation of actions on data objects
- Inheritance and more efficient programming
- More efficient GUI programming



Accessing Object Fields

- We have accessed object's attributes outside the class
  - Quick approach
  - Needs knowledge of implementation details
- Why it might be bad?
  - Allows inconsistent changes
  - Complicates possible changes/expansions of internal representation



Data Encapsulation / Hiding

- Data Hiding
  - To avoid inconsistent changes we can protect fields from being accessed outside of the class
- Data Encapsulation
  - To avoid freezing internal representation, we expose "conceptual representation"
- Data Abstraction
  - Data Hiding + Data Encapsulation

Proper Attribute Access

- Attributes should not be accessed directly outside the class
- Class should provide methods to access attributes
  - Getters: methods to get the values of attributes
  - Setters: methods to set the values of attributes

Example: Max GPA: Getters

```
def getName(self):
        return self.name

def getHours(self):
        return self.hours

def getQPoints(self):
        return self.qpoints
```

Example: Max GPA: Main (cont)

```
    infile.close()

# This is the changed part of main
# Access to attributes is by getters
# print information about the best
    print("The best student is:", \
                        best.getName())
    print("hours:", best.getHours())
    print("GPA:", best.gpa())
```

Example: Max GPA: Setters?

- We might not be willing to provide direct setting of all attributes
  - To avoid changing special attributes
  - To avoid inconsistent changes: Two or more attributes should be changed in sync
- Consistent change of qpoints and hours when adding a course

```
def addCourse(self, number, c_hours, grade):
      self.hours += c_hours
      self.qpoints += grade * c_hours
```

Data Protection/Hiding in Python

- It is in the name!
- name Public
  - Public attributes can be freely used inside or outside of a class definition.
- _name Protected
  - Protected attributes should not be used outside of the class definition, unless inside of a subclass definition.
- __name Private
  - Private attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except *inside of the class definition itself*.

Example: Public/Private/Protected

```
class Mytest():

    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"

    def getPriv(self):
        return self.__priv

    def getProt(self):
        return self._prot

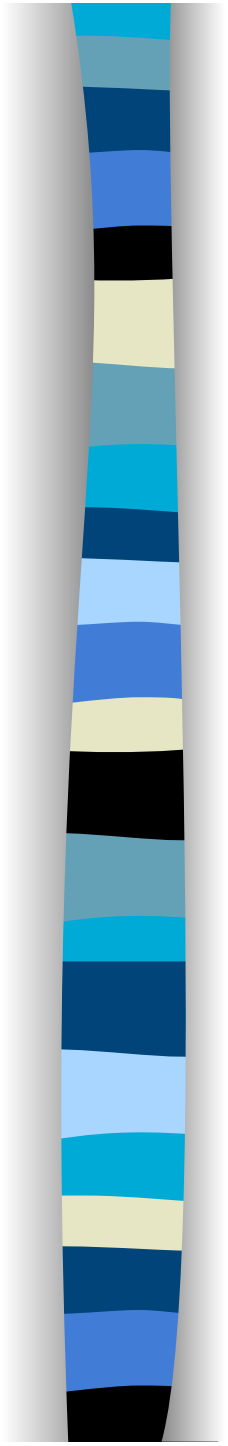
    def getPub(self):
        return self.pub
```

```
def main():
        a = Mytest()
        print(a.getPub())
        print(a.getProt())
        print(a.getPriv())
        print(a.pub)
        print(a._prot)
        print(a.__priv)
```

Object as a String

- The secret behind s = str(a) and print(a)
- If you apply *str* or *print* an object, Python is looking for a corresponding method __str__ in the class definition of the object
- If the method does exist, it will be called
- If not, Python will print a default object representation

```
<__main__.Test object at 0x103035198>
```



Example: Bank Account

- Conceptual view
  - Bank Account has number, owner's name, balance (we do not care about the rest)
- We need getters for conceptual fields
- We need consistent ways to change these fields, not simple setters
  - deposit, withdraw, transfer
- We need __str__

Class BankAccount: constructor

```
class BankAccount:

    # __init__ Method creates a new bank account.
    # Account number and the
    # owner name (strings) are given as parameters.

    def __init__(self, number, owner):
        self.account_number = number
        self.name = owner
        self.balance = 0.0
```

Class BankAccount: getters

```
class BankAccount:

    # __init__ Method creates a new bank account.
    # Account number and the
    # owner name (strings) are given as parameters.

    def __init__(self, number, owner):
        self.account_number = number
        self.name = owner
        self.balance = 0.0
```

Class BankAccount: core

```
class BankAccount:

    # __init__ Method creates a new bank account.
    # Account number and the
    # owner name (strings) are given as parameters.

    def __init__(self, number, owner):
        self.account_number = number
        self.name = owner
        self.balance = 0.0

    def __str__ (self):
        str = "Acct {:s}, Owner {:s}, Balance ${:.2f}".format(
                self.account_number, self.name, self.balance)
        return str
```

Class BankAccount: getters

```
# This method returns a bank account number.

    def get_number(self):
        return self.account_number

# This method returns the name of the owner account.

    def get_owner(self):
        return self.name

# The method returns the account balance (decimal).

    def get_balance(self):
        return self.balance
```

Class BankAccount: operations

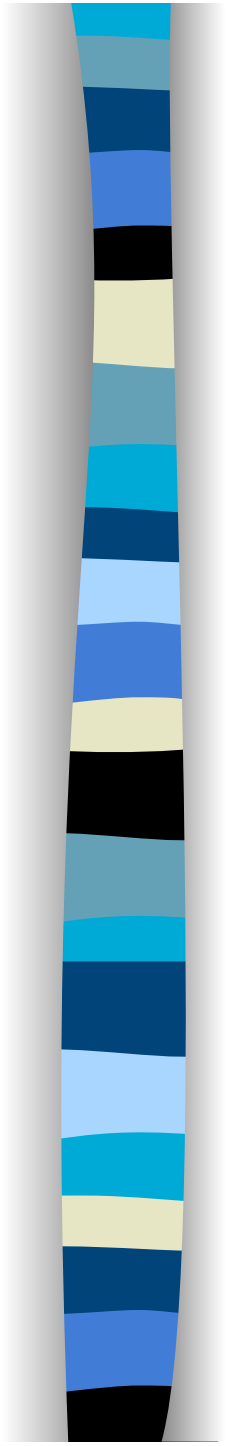
```
def deposit(self, amount):
        if amount > 0.0:
            self.balance += amount

def withdraw(self, amount):
        if amount > 0.0 and amount <= self.balance:
            self.balance -= amount
            return amount
        else:
            return 0.0

def transfer(self, amount, account):
        if self.withdraw(amount) > 0.0:
            account.deposit(amount)
            return True
        else:
            return False
```

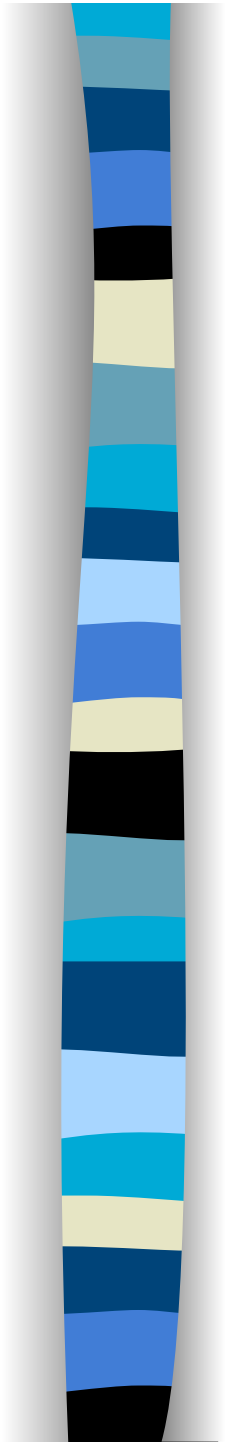
Lab Work with Bank Account

- Create a new bank account (object) with the account number "5577" and the account holder John Doe. Please note that the account number is a string.
- Add $429.40 to the created account using *deposit* method.
- Print account balance with two decimal places by using the *get_balance* method.
- Withdraw $180.00 from this account by using the *withdraw* method and print and the amount withdrawn.
- Ask the user to enter another account number and read it
- Ask the user to enter another account owner's name and read it.
- Create the second account (object), using the account number and the owner's name entered by the user
- Ask the user to enter the amount of money to deposit to the new account, read the number, and deposit it to the account
- Print the second account balance with two decimal places by using the *get_balance* method.
- Prints details of the first and second account by using *__str__* method. Note: do not call *__str__* method directly, simply print the objects. This will cause Python to call this method and print the returned string.
- Ask the user to enter the amount of money to be transferred from the second account to the first account created by the program (John Doe's).
- Print whether the transfer was successful.
- Prints details of the first and second account by using *__str__* method.



Why Objects and OOP?

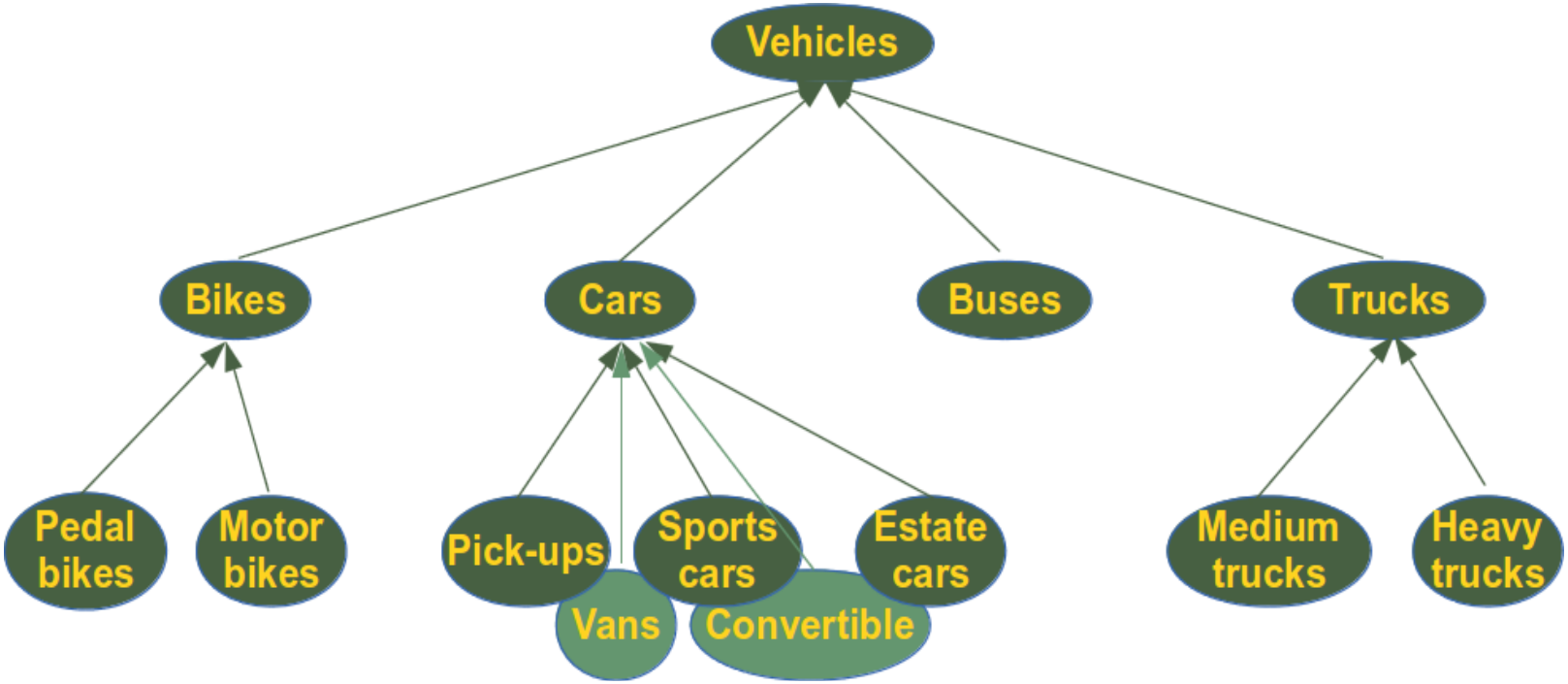
- Heterogeneous data aggregates
- Encapsulation of actions on data objects
- Inheritance and more efficient programming
- More efficient GUI programming

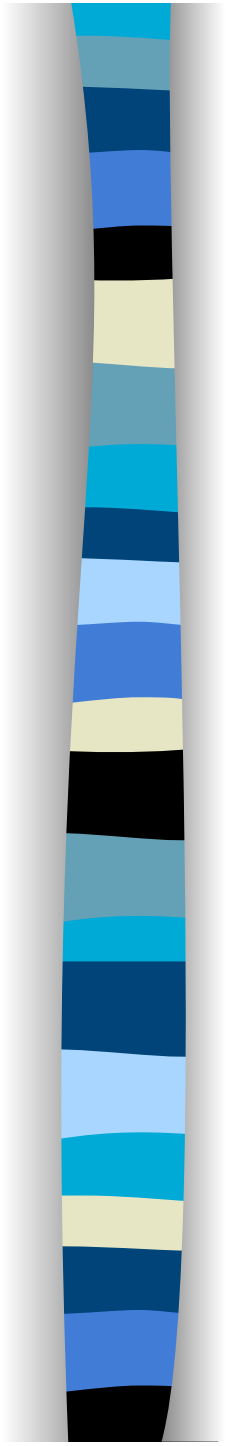


Inheritance

- Classes can inherit from other classes.
- A class can inherit attributes and behavior methods from another class, called the *superclass (ancestor* class*)*
- A class which inherits from a superclass is called a *subclass* (*descendant*, *heir*, or *child* class).
- There is a hierarchy relationship between classes.

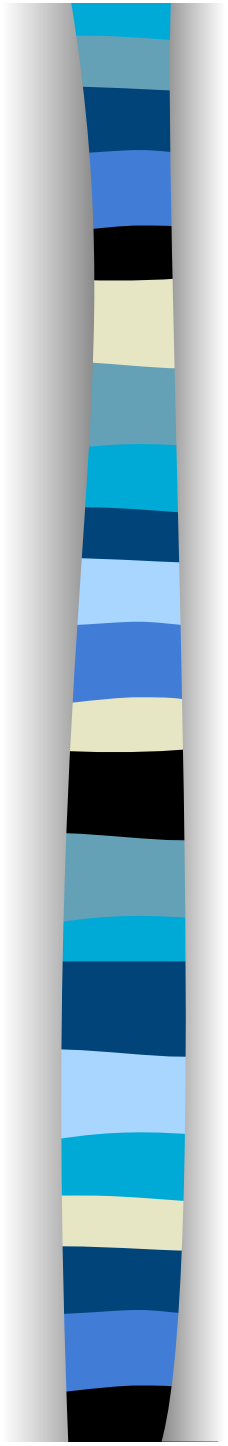
Class Hierarchy





Subclass Structure

- Subclass usually allows to store more information and to do more than the base class
- It might have *additional* attributes and *additional* methods
- Sometimes superclass methods should be changed to allow working with new data
- In this case subclass provides methods with the same name that override the superclass methods



Subclass Structure

```
class SubclassName(SuperclassName):
    <overriding methods>
    <additional methods>
```

Example: TA

- Let's create a subclass of the class student to represent teaching assistants
- Teaching assistant is a student, but can TA a class – we need attribute to represent it
- Also, you can only TA a class that you took in the past with A or better grade. To check TA eligibility, we need to add another attribute: grades for all past taken courses

TA: Overridden Methods

```
class TA(Student):

#   Subclass student adds a dictionary with grades for past courses
#   It is important, since a TA should took the class before with A
#    __init__ method should be overridden to initialize .grades and .courseTA
    def __init__(self, name, hours, qpoints):
        self.name = name
        self.hours = hours
        self.qpoints = qpoints
        self.grades = {}
        self.courseTA = ""

#   This method should also be overridden to add new course to the dict
    def addCourse(self, c_name, c_hours, grade):
                self.hours += c_hours
                self.qpoints += grade * c_hours
                self.grades[c_name] = grade
```

TA: New Methods

```
#   This new method checks whether a student can be a TA in general
    def canTA(self):
        if self.gpa() > 3.5:
            return True
        else:
            return False

#   This method appoints a student as a TA if the course has been taken
#   before with grade A or better. Returns True if success, False otherwise
    def appointTA(self, course):
        if course in self.grades and self.grades[course] >= 4 :
            self.courseTA = course
            return True
        else:
            return False
```

Using TA: Main – TA interview

```
#   Main function similates a TA interview
def main():
    print("This is a TA interview!")
    name = input ("What is your name? ")
    candidate = TA(name, 0, 0)
    addcourses(candidate)
    print("Thank you, your GPA is", candidate.gpa())
    if candidate.gpa() > 3:
        print("With this GPA, you can be a TA!")
        course = input ("For which course you want to be a TA? ")
        if candidate.appointTA(course):
            print("You are appointed a TA for", course)
        else:
            print("You can't be a TA for", course)
    else:
        print("With this GPA, you can't be a TA!")
```

Using TA: Adding Courses

```
#   This function ask the user about taken courses and add it to the record
def addcourses (student):
        course_number = int (input ("How many courses you took in the past?"

        for i in range(course_number):
                number = input ("Next course number: ")
                grade = float(input ("Grade for this course: "))
                hours = float(input ("Hours for this course: "))
                student.addCourse(number, hours, grade)
```