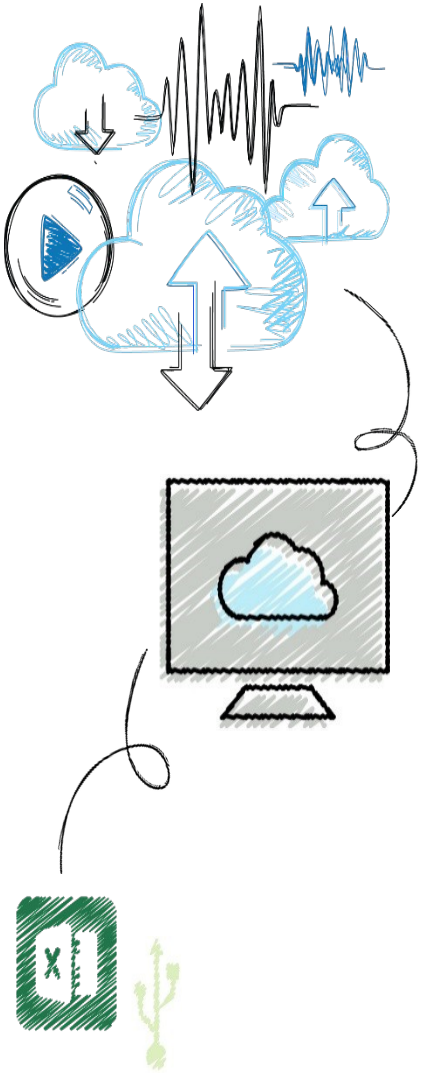
Module 5

Chapter 9

OOP: Inheritance and Polymorphism



Objective

9.7 Building a new data structure

9.8 Exploit inheritance and polymorphism when developing classes

Building a New Data Structure: The Two-Dimensional Grid

- A useful data structure: **two-dimensional grid**
- A grid organizes items by position in rows and columns
- In this section, we develop a new class called **Grid**
  - For applications that require grids

The Interface of the Grid Class (1 of 2)

- The constructor or operation to create a grid allows you to specify the width, the height, and an optional initial fill value for all of the positions
  - Default fill value is None
- You access or replace an item at a given position by specifying the row and column of that position, using the notation:
  - **grid[<row>] [<column>]**

The Interface of the Grid Class (2 of 2)

Grid Method	**What It Does**
g = Grid(rows, columns, fillValue = None)	Returns a new Grid object
g.getHeight()	Returns the number of rows
g.getWidth()	Returns the number of columns
g.__str__()	Same as str(g). Returns the string representation
g.__getitem__(row)[column]	Same as g.[row][column]
g.find(value)	Returns (row, column) if value is found, or None otherwise

The Implementation of the Grid Class: Instance Variables for the Data

- Implementation of a class provides the code for the methods in its interface
  - As well as the instance variables needed to track the data contained in objects of that class
- Next step is to choose the data structures that will represent the two-dimensional structure within a **Grid** object
  - A single instance variable named **self.data** holds the top-level list of rows
  - Each item within this list will be a list of the columns in that row
- Other two methods:
  - **_init_**, which initializes the instance variables
  - **_str_**, which allows you to view the data during testing

A “list” of lists

The Implementation of the Grid Class: Subscript and Search

- Subscript operator
  - used to access an item at a grid position or to replace it there
- In the case of access
  - The subscript appears within an expression, as in **grid[1] [2]**
- Search operation named **find** must loop through the grid’s list of lists
  - Until it finds the target item or runs out of items to examine

Structuring Classes with Inheritance and Polymorphism

- Most object-oriented languages require the programmer to master the following techniques:
  - **Data encapsulation:** Restricting manipulation of an object’s state by external users to a set of method calls (*classes*)
  - **Inheritance:** Allowing a class to automatically reuse/extend code of similar but more general classes (like we did with *EasyFrame*)
  - **Polymorphism:** Allowing several related classes to use the same general method names
    - But allow for variation in implementation
- Python’s syntax doesn’t *enforce* data encapsulation
  - But classes allow us to group data and methods together
- Inheritance and polymorphism are built into Python

Approximating Privacy

- Classes allow you to define both **data** and **methods** for that data
- User should primarily use a class’s methods
  - And **not** access data attributes directly
- But there is no way to stop them from accessing the data
  - Other OOP languages have “private” access specifiers
- In Python we use **two underscores** before the data attribute name
  - Python will “mangle” the name as “_classname__attributename”
  - Users won’t likely use the mangled name
  - But this technique has its drawbacks, too
- See *private.py*

Inheritance Hierarchies and Modeling (1 of 2)

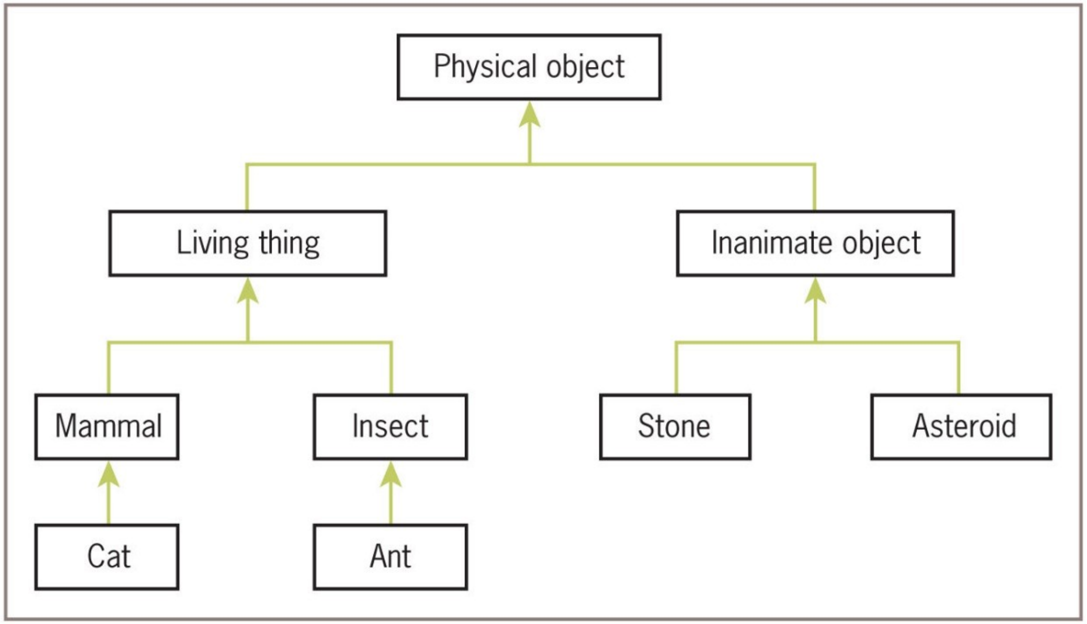


Figure 9-5 A simplified hierarchy of objects in the natural world

Inheritance Hierarchies and Modeling (2 of 2)

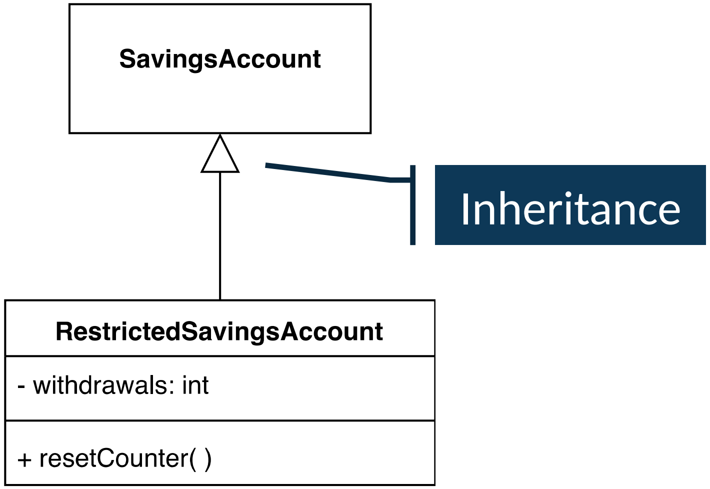
- In Python, all classes automatically extend the built-in **object** class
- It is possible to extend any existing class:

class <new class name>(<existing parent class name>):

- Example:
  - **PhysicalObject** would extend **object**
  - **LivingThing** would extend **PhysicalObject**
- Inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code

Example 1: A Restricted Savings Account

```
>>> account = RestrictedSavingsAccount(“Ken”, “1001”, 500.00)
>>> print(account)
Name: Ken
PIN: 1001
Balance: 500.0
>>> account.getBalance()
500.0
>>> for count in range(3):
account.withdraw(100)
>>> account.withdraw(50)
‘No more withdrawals this month’
>>> account.resetCounter()
>>> account.withdraw(50)
```



- To call a method in the parent class from within a method with the same name in a subclass:

<parent class name>.<method name>(self, <other arguments>)

Example 2: The Dealer and a Player in the Game of Blackjack (1 of 2)

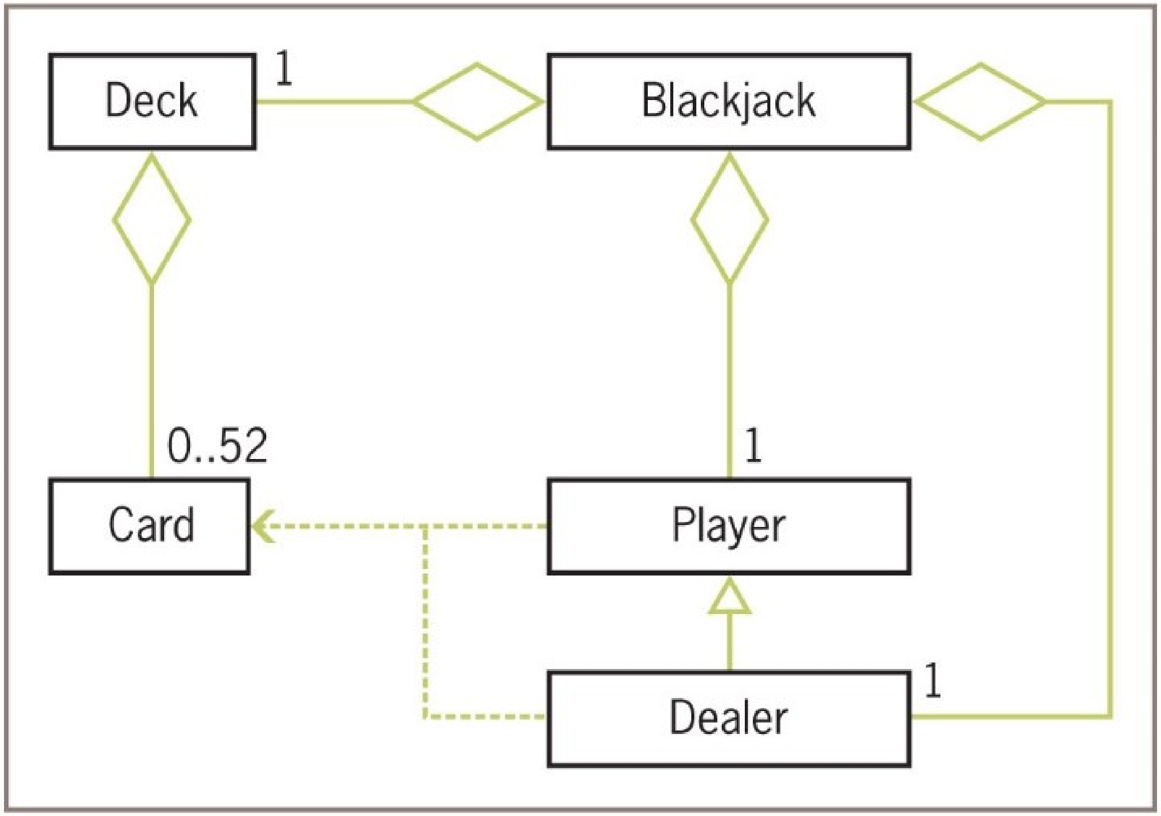


Figure 9-6 The classes in the blackjack game application

Example 2: The Dealer and a Player in the Game of Blackjack (2 of 2)

- An object belonging to **Blackjack** class sets up the game and manages the interactions with user

```
>>> from blackjack import Blackjack
>>> game = Blackjack()
>>> game.play()
Player:
2 of Spades, 5 of Spades
7 points Dealer:
5 of Hearts
Do you want a hit? [y/n]: y
Player:
2 of Spades, 5 of Spades, King of Hearts
17 points
Do you want a hit? [y/n]: n
Dealer:
5 of Hearts, Queen of Hearts, 7 of Diamonds
22 points
Dealer busts and you win
```

Which output comes from which object?

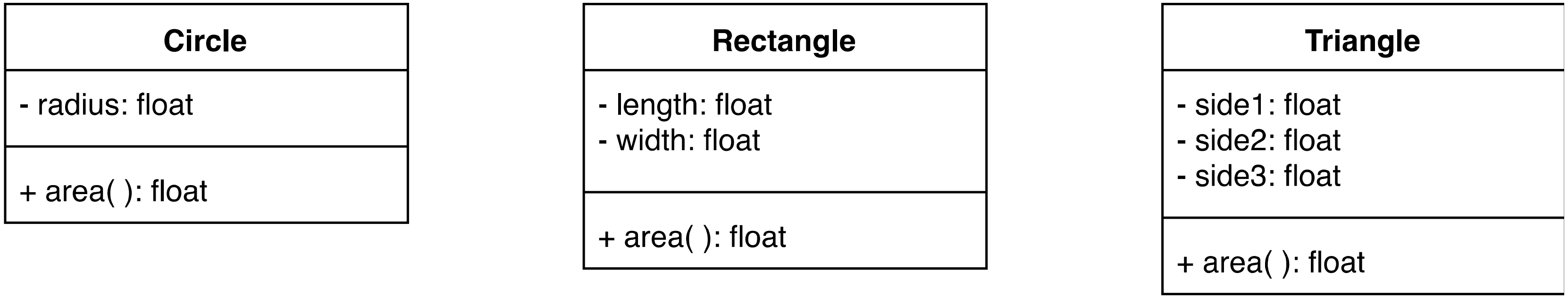
Polymorphic Methods

- We subclass when two classes share a substantial amount of **abstract behavior**
  - The classes have similar sets of methods/operations
  - A subclass can **add** something extra
  - But most often it **modifies behavior** of an operation
- The two classes may have the same interface
  - One or more methods in subclass override the definitions of the same methods in the superclass to provide specialized versions of the abstract behavior
    - **Polymorphic methods** (e.g., the **__str__** method)

A Canonical Example: Shapes

- Circle, Rectangle, Triangle, Trapezoid, Square
  - We’ll only deal with 2-dimensional shapes
- What do all of these have in common?

Shape Classes



Circle

- radius: float

+ area( ): float

Rectangle

- length: float
- width: float

+ area( ): float

Triangle

- side1: float
- side2: float
- side3: float

+ area( ): float

Design Principles

1. Don’t **R**epeat Yourself (**DRY**)
   - “One definition rule”
   - So you only have *one place* to fix/update something
2. **Separate** Things that **Change** from Things that Stay the **Same**
   - So things that need to change can
   - And things that shouldn’t change don’t
   - The **type** of an object **can’t change** → object ≠ variable
3. Program to an **Interface**
   - Interface == methods available to user programmers
   - **Not** an Implementation
   - So you are shielded from changes/fixes in implementation detail

Discussion

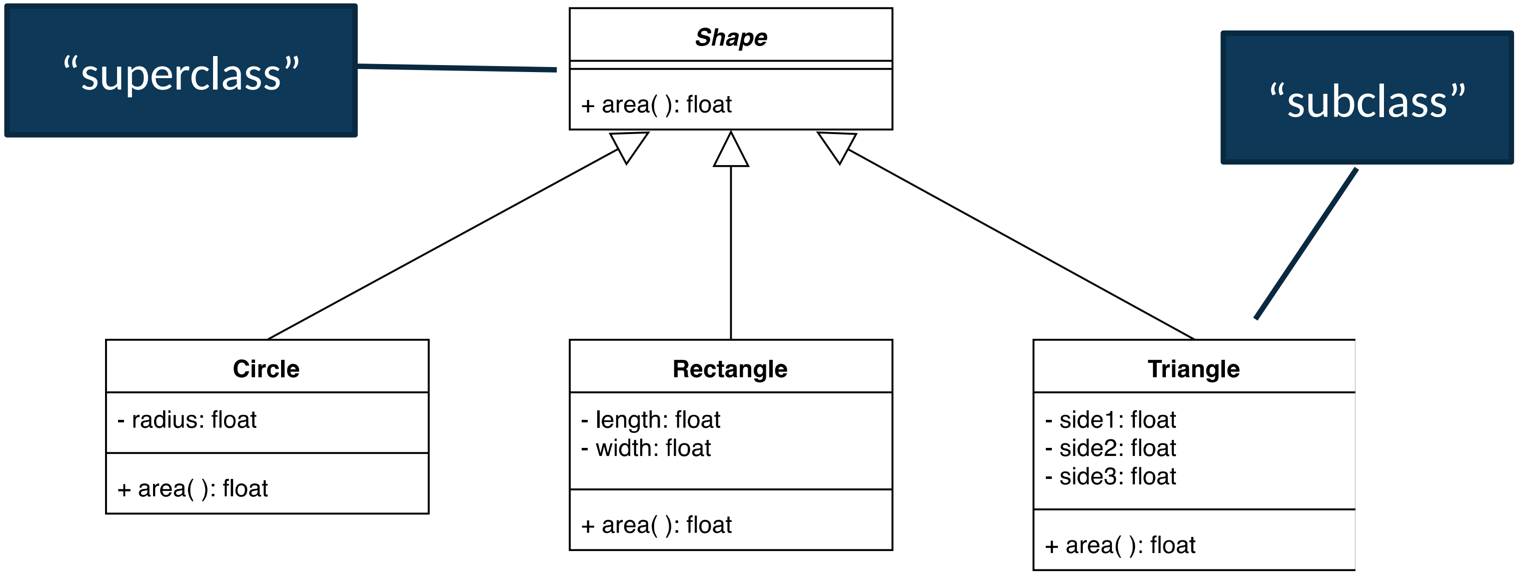
- Different shapes have different data
  - So that data will reside in each different class
- But they all have an **area** method
  - **area** is the **interface** for all shapes
- But the **implementation** of each area is **different**!

- So put the **interface** in a single, shared place
  - And the **implementations** in each **class**

- But how !!!???

Inheritance

- Allows things in common to be in one, *shared* place
- But how do we place the interface in a shared, superclass…
  - … and their implementations in each respective classes?



Polymorphism in Python: How Does It Work?

- **M**ethod **R**esolution **O**rder
  - A **lookup order** to find class attributes and methods
- 1) Looks in the class of the object itself first to find the attribute/method
  - That’s what happened here, the usual case
  - Each concrete, derived object is an instance of a class that has an **area** method
- 2) If it’s not there, it will look in the base class
  - And so on…

```
>>> print(Rectangle.__mro__)
(<class '__main__.Rectangle'>, <class '__main__.Shape’>,
<class ‘abc.ABC>, <class 'object’>)
```

Illustrating MRO

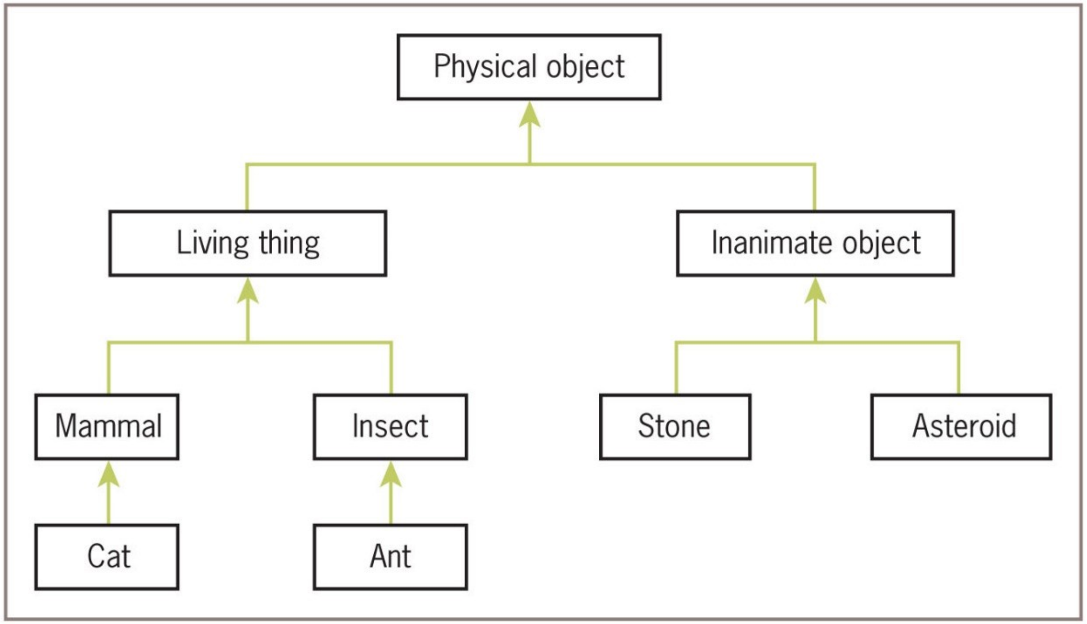


Figure 9-5 A simplified hierarchy of objects in the natural world

See mro.py

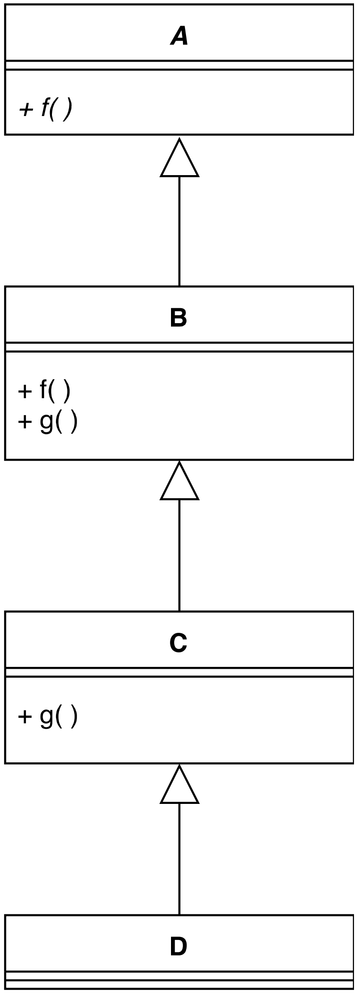
Abstract Classes

- Contain **abstract methods** for subclasses to **override**
  - They need no function bodies
  - They mainly exist as a placeholder to be overridden by subclasses
- We make the superclass abstract by subclassing **abc.ABC**
- We make methods abstract with the **abc.abstractmethod** *decorator*

- We can have a mixed list of concrete (derived) shape objects
  - And call **area** and it will just do the right thing automatically!
  - This is the power of OOP

- See *shape.py*

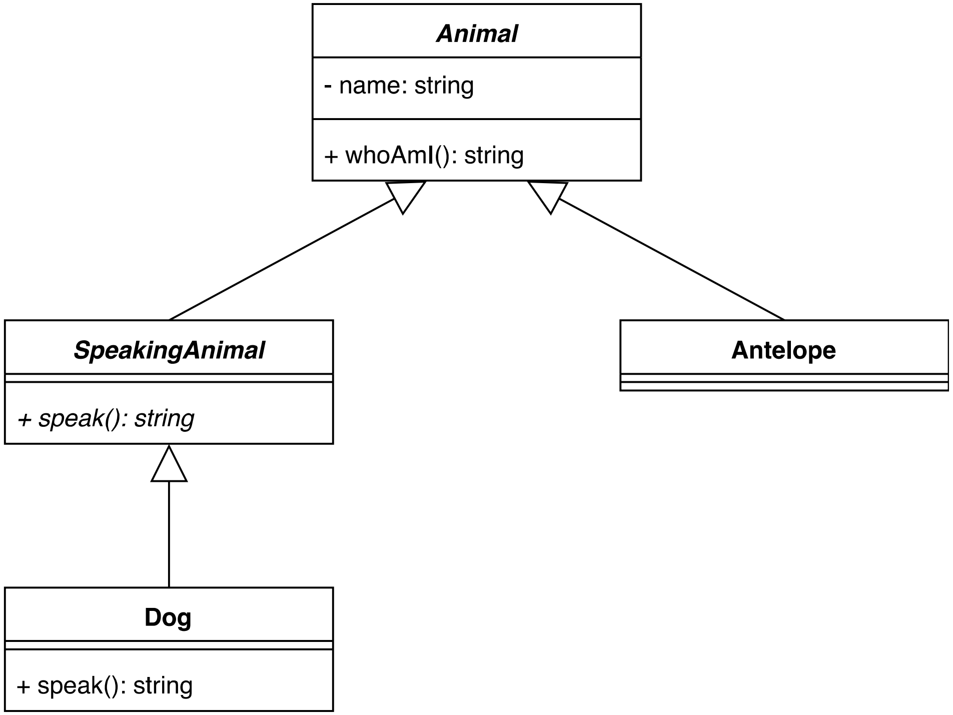
Which Method is Called?



```
d = D()
d.f()      # ?
d.g()      # ?
```

An Animal Hierarchy

- Data common to all subclasses goes in a superclass
- Subclass constructors must pass the data to **superclass constructors**
- See *animal.py*



A Case Study

A Rewards Program

- A certain hotel chain gives loyalty rewards to frequent customers.
- There are 4 **levels** of benefits for those who enroll in the rewards program:
  - **Basic** (less than 10 nights per year)
    - 100 points per stay per night, free Wi-Fi, exclusive rates, optional mobile-phone room key
  - **Silver** (10+ nights per year)
    - 10% bonus points, late checkout
  - **Gold** (25+ nights per year)
    - 25% bonus, free room upgrade
  - **Platinum** (50+ nights)
    - 50% bonus points, free welcome gift
- Each **level** enjoys all the benefits of lower levels
- **Members** can *change* levels
- Design classes to track stays, levels, and rewards of program members

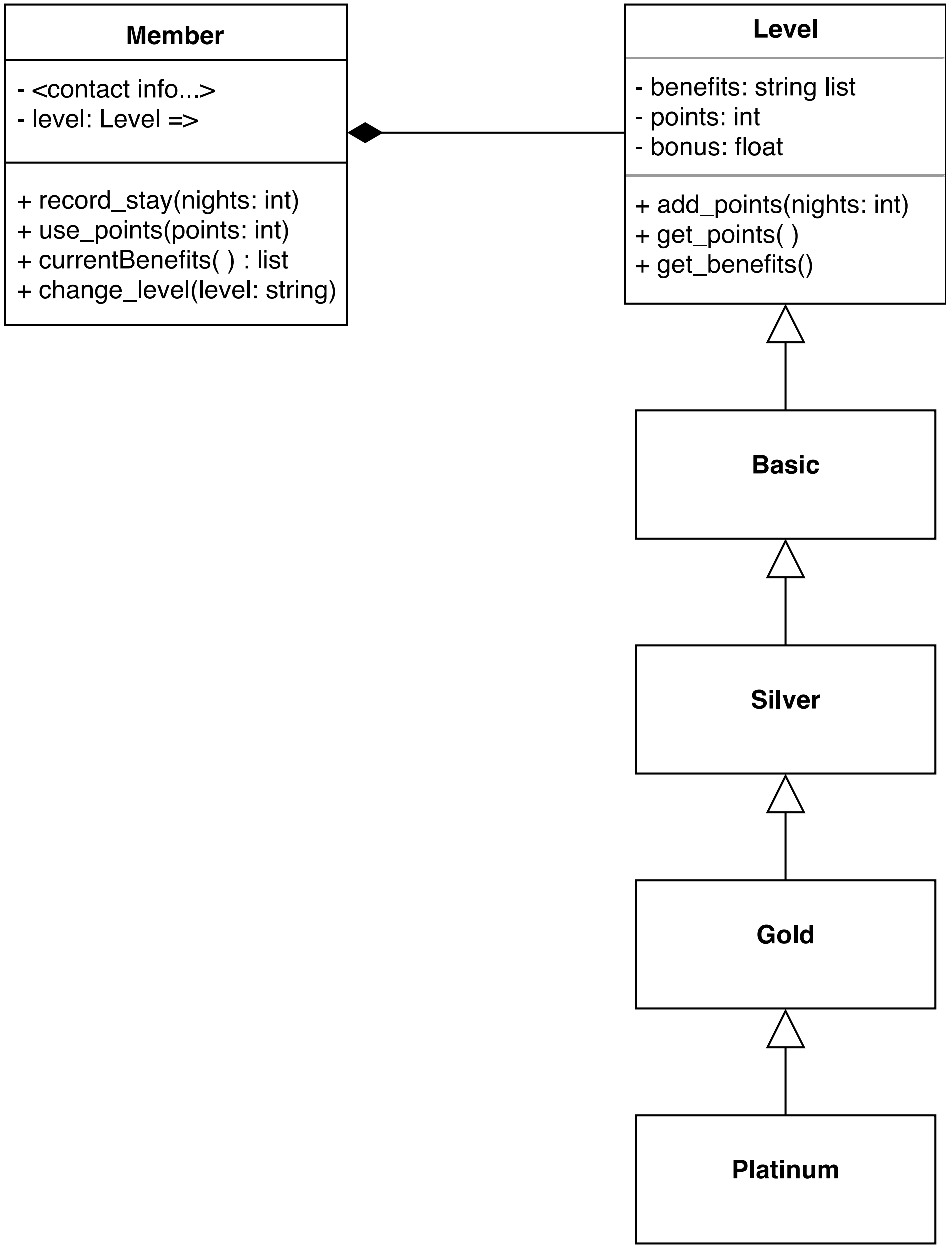
Design Steps

- What are the common **use cases** for this system?
  - Track awarding and redeeming of points
  - Track other benefits available
  - Change reward level
  - …?
- CRC Cards
  - Identify classes, responsibilities, collaborators, relationships
- UML Diagrams
  - **Class** diagram
  - **Sequence** diagrams for applicable use cases
    - (Not crucial for this example)

CRC Cards

Member	
● Knows contact info, tracks member rewards level ● Records rewards earnings and redemption	● Level

Abstract **Level** Basic, Silver, Gold, Platinum	
● Applies applicable level bonus points ● Knows extra benefits	



See *rewards.py*

Review Program 5 Spec

- **Payroll** System
- Different types of Employees
  - Salaried
  - Hourly
  - Commissioned (also receive a salary)
- Different types of Payment Methods
  - Mail a check
  - Direct transfer to bank
- Data files:
  - employees.csv
  - timecards.txt
  - receipts.txt

Employee	
- Manage Employee attributes - Change employee's classification - Change employee's payment method - Initiate payment to employee	- Classification - PaymentMethod

Abstract **Classification**	Hourly, Salaried, Commissioned
- Specifies the abstract method issue_payment - Know the employee	- Employee

Hourly	Classification
- Know the employee's hourly_rate - Add new time cards - Hold current time cards - Compute the employee's pay - Invoke the pay method	- Employee - PaymentMethod

Salaried	Classification
- Know the employee's salary - Invoke the pay method	- Employee - PaymentMethod

Classification

Commissioned

• Know the employee's salary, and commission rate • Add receipts • Hold the current sales receipts • Invoke the pay method	• Employee • PaymentMethod

Abstract

PaymentMethod

DirectMethod, MailMethod

• Specifies the abstract method issue, which posts the employee's payment • Knows the employee	• Employee

PaymentMethod

DirectMethod

• Know the employee's bank routing and account numbers • Transfers funds to the bank (in our case, just prints a line to the log file)	• Employee

PaymentMethod

MailMethod

• Know the employee's name and complete address • Print paycheck (in our case, just prints a line to the log file)	• Employee

Case Study 2

A Pet Store Chain

- **Pet Peddlers** has a chain of pet stores that offer different kinds of pets
    - **Mammals** (cats, dogs)
    - **Birds** (parakeets, toucans)
    - **Reptiles** (snakes, turtles)
    - **Amphibians** (frogs, newts)
    - **Fish** (koi, guppies)
- Each store has a unique, **numeric id number**
- Each store keeps track of their animal **inventory**, which *can change*
- All pets have a unique **name** and are **fed** regularly
- All concrete types of animal categories have the same diet (e.g., all cats eat mice)
- Individual pets take turns being the “featured pet”

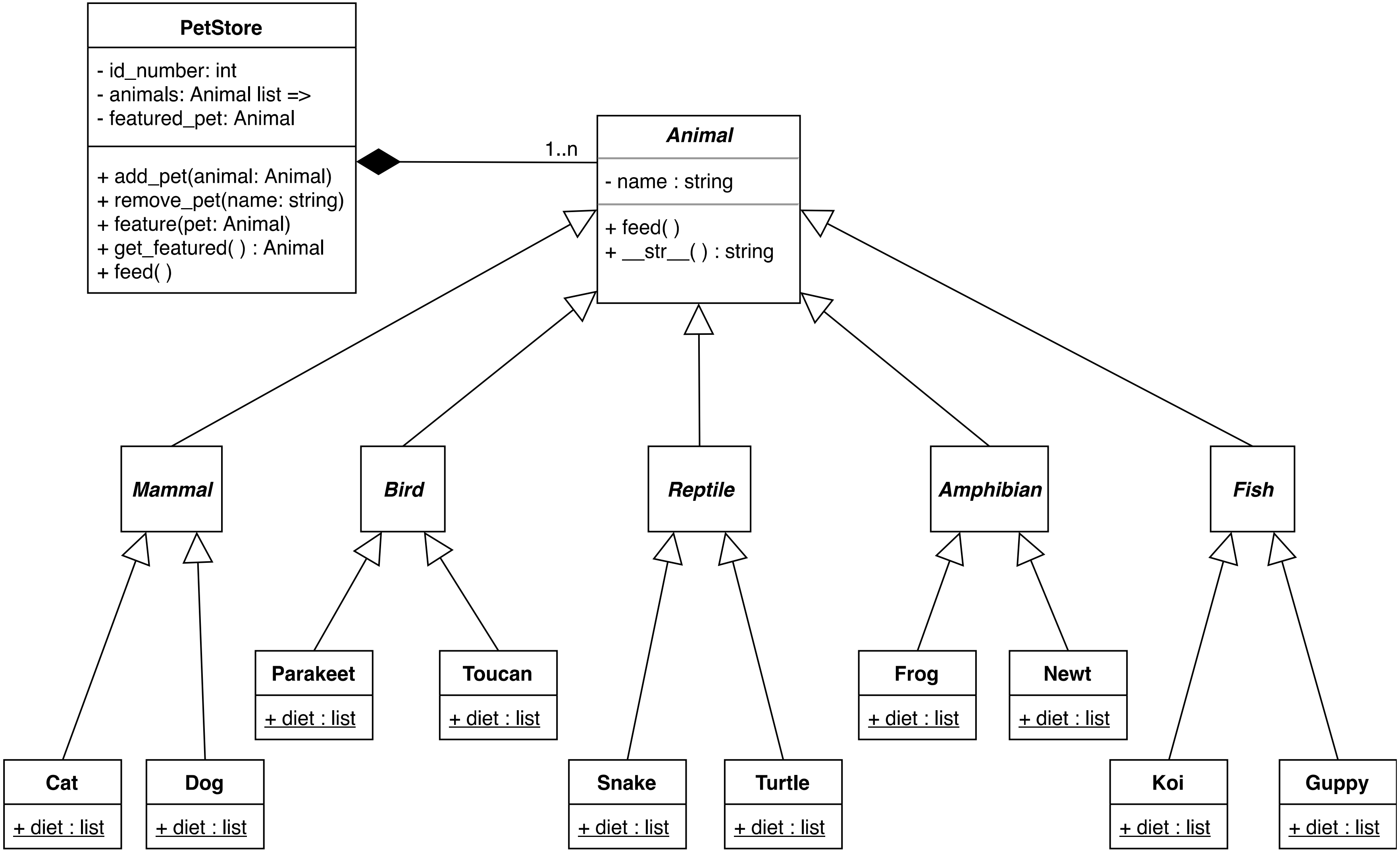
A Sample Run

```
store = PetStore(1)
store.add_pet(Guppy('Gus'))
store.add_pet(Newt('Tiny'))
store.add_pet(Toucan('Tad'))
store.add_pet(Cat('Kevin'))
store.add_pet(Turtle('Ted'))
store.add_pet(Snake('Slimey'))

store.feed()
print()
store.feature('Tiny')


print("\nReptiles:")
for pet in store.get_reptiles():
    print(pet)

print("\nFish:")
for pet in store.get_fish():
    print(pet)
```

```
Gus eating flakes
Tiny eating worms
Tad eating caterpillars
Kevin eating mice
Ted eating carrots
Slimey eating rodents

Featured pet.. Newt: Tiny

Reptiles:
Turtle: Ted
Snake: Slimey

Fish:
Guppy: Gus
```

What Are the Classes?

- Do CRC Cards as needed
  - See *petstoreCRC.pdf* in the Slides folder on Canvas
- Then do a UML diagram
- Then code it up!
  - See *petstore.py* in Code folder on Canvas



PetStore

- id_number: int
- animals: Animal list =>
- featured_pet: Animal

+ add_pet(animal: Animal)
+ remove_pet(name: string)
+ feature(pet: Animal)
+ get_featured( ) : Animal
+ feed( )

1..n

Animal

- name : string

+ feed( )
+ __str__( ) : string

Mammal

Bird

Reptile

Amphibian

Fish

Parakeet

+ diet : list

Toucan

+ diet : list

Frog

+ diet : list

Newt

+ diet : list

Cat

+ diet : list

Dog

+ diet : list

Snake

+ diet : list

Turtle

+ diet : list

Koi

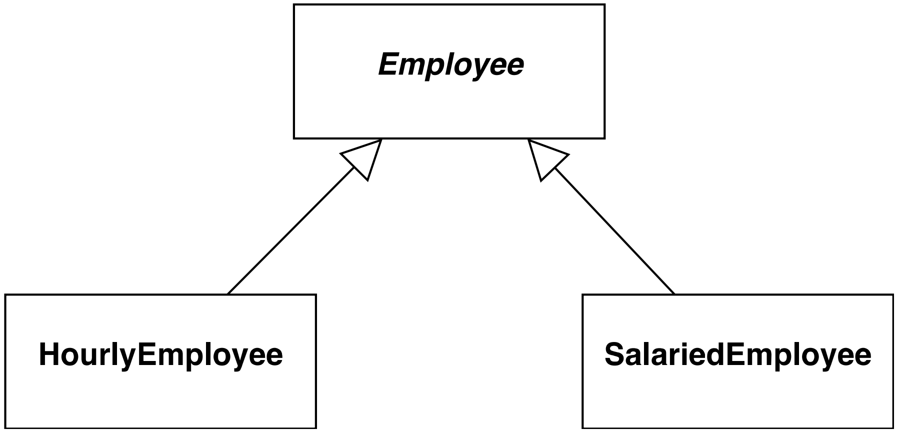
+ diet : list

Guppy

+ diet : list

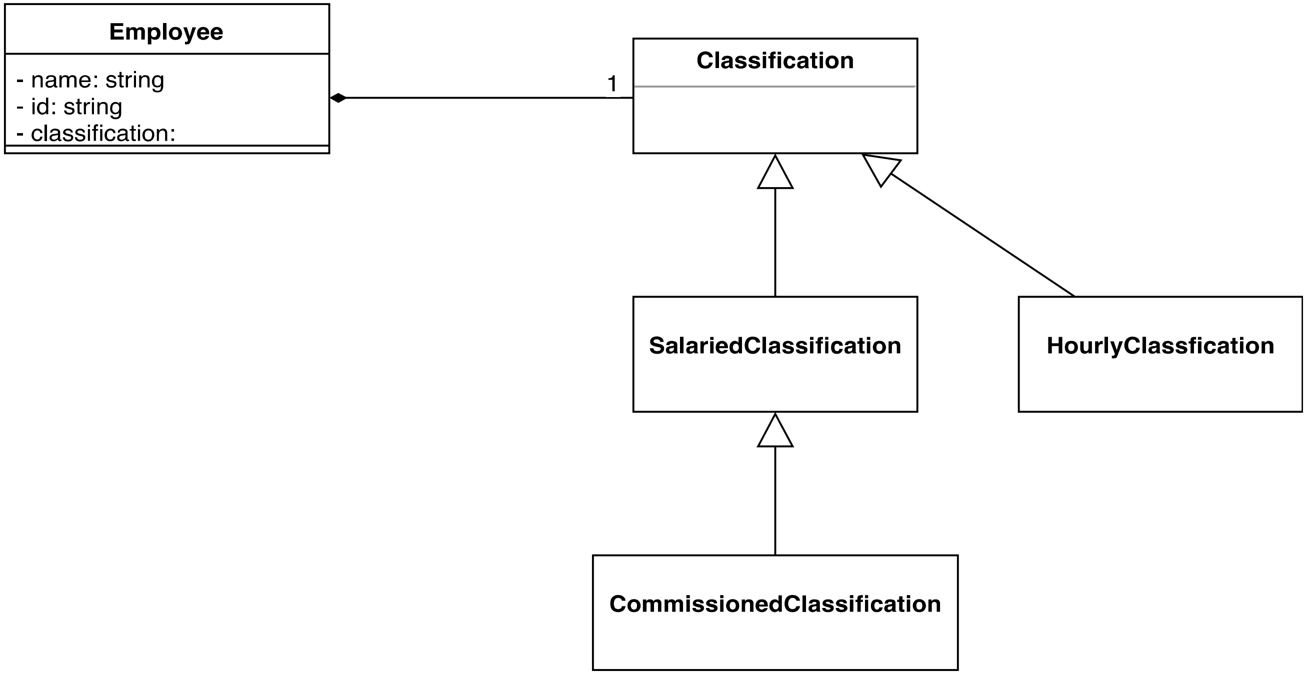
Don’t Do This!

- Employees’ **classification** can *change* in real life.
- But objects in programs can’t (shouldn’t) change their type!



Do This Instead

- “Separate what changes from what stays the same”
- We **separate** the **classification** from the Employee object



CENGAGE

Chapter Summary

- Most important features of OO programming:
  - **Encapsulation** restricts access to an object's data to users of the methods of its class
  - **Inheritance** allows one class to pick up the attributes and behavior of another class for free
  - **Polymorphism** allows methods in several different classes to have the same headers

- Class Hierarchies are often rooted in an **Abstract Class**
  - Establishes the **interface** for all types on the hierarchy
  - Often using **abstract methods**
  - Houses **shared code**, if any
  - Subclasses **override** methods as needed