Object Oriented Programming II

Dr. Burak Kaleci

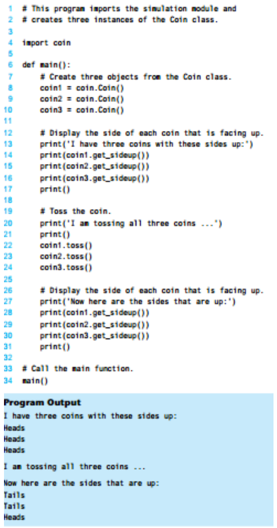
March 8, 2019

Content

Introduction

- **When a method uses the self parameter to create an attribute, the attribute belongs to the specific object that self references.**
- **We call these attributes instance attributes because they belong to a specific instance of the class.**
- It is possible to create many instances of the same class in a program.
- **Each instance will then have its own set of attributes.**

An Example



```
1  # This program imports the simulation module and
2  # creates three instances of the Coin class.
3
4  import coin
5
6  def main():
7      # Create three objects from the Coin class.
8      coin1 = coin.Coin()
9      coin2 = coin.Coin()
10     coin3 = coin.Coin()
11
12     # Display the side of each coin that is facing up.
13     print('I have three coins with these sides up:')
14     print(coin1.get_sideup())
15     print(coin2.get_sideup())
16     print(coin3.get_sideup())
17     print()
18
19     # Toss the coin.
20     print('I am tossing all three coins ...')
21     print()
22     coin1.toss()
23     coin2.toss()
24     coin3.toss()
25
26     # Display the side of each coin that is facing up.
27     print('Now here are the sides that are up:')
28     print(coin1.get_sideup())
29     print(coin2.get_sideup())
30     print(coin3.get_sideup())
31     print()
32
33 # Call the main function.
34 main()
```

Program Output

```
I have three coins with these sides up:
Heads
Heads
Heads

I am tossing all three coins ...

Now here are the sides that are up:
Tails
Tails
Heads
```

- In lines 8 through 10, the following statements create three objects, each an instance of the Coin class:

  coin1=coin.Coin()
  coin2=coin.Coin()
  coin3=coin.Coin()

- Notice each object has its own __sideup attribute.
- Lines 14 through 16 display the values returned from each object's get_sideup method.
- Then, the statements in lines 22 through 24 call each object's toss method:

  coin1.toss()
  coin2.toss()
  coin3.toss()

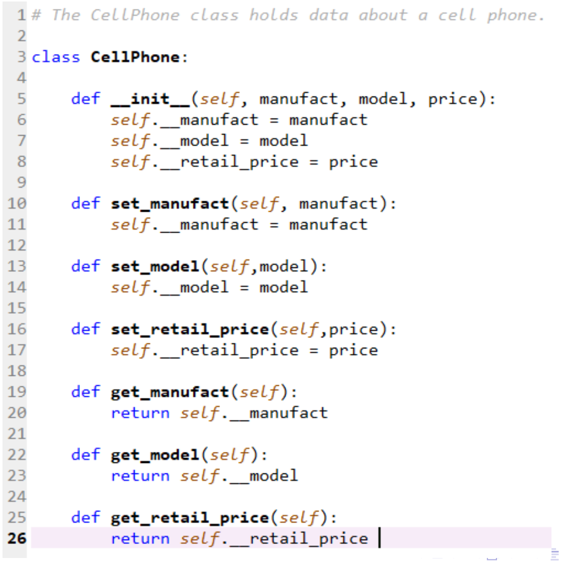
get and set functions

- You have been asked to design a class that represents a cell phone.
- The data that should be kept as attributes in the class are as follows:
  The name of the phone's manufacturer will be assigned to the __manufact attribute.
  The phone's model number will be assigned to the __model attribute.
  The phone's retail price will be assigned to the __retail_price attribute.

get and set functions

- The class will also have the following methods:
  An __**init**__ method that accepts arguments for the manufacturer, model number, and retail price.
  A **set_manufact** method that accepts an argument for the manufacturer. This method will allow us to change the value of the __manufact attribute after the object has been created, if necessary.
  A **set_model** method that accepts an argument for the model. This method will allow us to change the value of the __model attribute after the object has been created, if necessary.
  A **set_retail_price** method that accepts an argument for the retail price. This method will allow us to change the value of the __retail_price attribute after the object has been created, if necessary.
  A **get_manufact** method that returns the phone's manufacturer.
  A **get_model** method that returns the phone's model number.
  A **get_retail_price** method that returns the phone's retail price.

CellPhone Class



```python
# The CellPhone class holds data about a cell phone.

class CellPhone:

    def __init__(self, manufact, model, price):
        self.__manufact = manufact
        self.__model = model
        self.__retail_price = price

    def set_manufact(self, manufact):
        self.__manufact = manufact

    def set_model(self,model):
        self.__model = model

    def set_retail_price(self,price):
        self.__retail_price = price

    def get_manufact(self):
        return self.__manufact

    def get_model(self):
        return self.__model

    def get_retail_price(self):
        return self.__retail_price
```

Driver Program for CellPhone Class

```

# This program tests the CellPhone class

import cellphone

def main():

    man = input('Enter the manufacturer: ')
    mod = input('Enter the model number: ')
    retail = float(input('Enter the retail price: '))

    # Create an instance of the CellPhone class.
    my_phone = cellphone.CellPhone(man,mod,retail)

    # Display the data that was entered.
    print('Here is the data that you entered:')
    print('Manufacturer:', my_phone.get_manufact())
    print('Model Number:', my_phone.get_model())
    print('Retail Price: $', format(my_phone.get_retail_price(), ',.2f'), sep='')

    my_phone.set_manufact("AE000")
    my_phone.set_model("BE100")
    my_phone.set_retail_price(5000)

    # Display the data that after set functions.
    print('Here is the data that you entered:')
    print('Manufacturer:', my_phone.get_manufact())
    print('Model Number:', my_phone.get_model())
    print('Retail Price: $', format(my_phone.get_retail_price(), ',.2f'), sep='')
```

Accessor and Mutator Methods

- As mentioned earlier, it is a common practice to make all of **a class's data attributes private**, and to provide **public methods for accessing and changing** those attributes.
- This ensures that the object owning those attributes is in control of all the changes being made to them.
- A method that returns a value from a class's attribute but does not change it is known as an **accessor** method.
- Accessor methods provide a safe way for code outside the class to retrieve the values of attributes, without exposing the attributes in a way that they could be changed by the code outside the method.
- In the CellPhone class the **get_manufact**, **get_model**, and **get_retail_price** methods are **accessor** methods.

Accessor and Mutator Methods

- A method that stores a value in a data attribute or changes the value of a data attribute in some other way is known as a **mutator method**.
- Mutator methods can control the way that a class's data attributes are modified.
- When code outside the class needs to change the value of an object's data attribute, it typically calls a mutator and passes the new value as an argument.
- If necessary, the mutator can validate the value before it assigns it to the data attribute.
- In the CellPhone class the **set_manufact**, **set_model**, and **set_retail_price** methods are **mutator** methods.

Passing Objects as Arguments

- When you are developing applications that work with objects, you often need to write functions and methods that accept objects as arguments.
- For example, the following code shows a function named show_coin_status that accepts a Coin object as an argument:
```
def show_coin_status(coin_obj):
    print('This side of the coin is up:', coin_obj.get_sideup())
```
- The following code sample shows how we might create a Coin object, then pass it as an argument to the show_coin_status function:
```
my_coin = coin.Coin()
show_coin_status(my_coin)
```
- When you pass an object as an argument, the thing that is passed into the parameter variable is a reference to the object.
- **As a result, the function or method that receives the object as an argument has access to the actual object.**
- For example, look at the following flip method:
```
def flip(coin_obj):
    coin_obj.toss()
```
- This method accepts a Coin object as an argument, and it calls the object's toss method.

Passing Objects as Arguments

```
 1  # This program passes a Coin object as
 2  # an argument to a function.
 3  import coin
 4
 5  # main function
 6  def main():
 7      # Create a Coin object.
 8      my_coin = coin.Coin()
 9
10      # This will display 'Heads'.
11      print(my_coin.get_sideup())
12
13      # Pass the object to the flip function.
14      flip(my_coin)
15
16      # This might display 'Heads', or it might
17      # display 'Tails'.
18      print(my_coin.get_sideup())
19
20  # The flip function flips a coin.
21  def flip(coin_obj):
22      coin_obj.toss()
23
24  # Call the main function.
25  main()
```

Program Output

```
Heads
Tails
```

Program Output

```
Heads
Heads
```

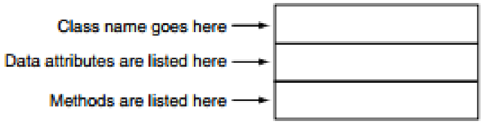
Program Output

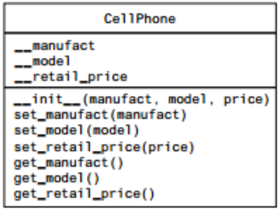
```
Heads
Tails
```

- The statement in line 8 creates a Coin object, referenced by the variable my_coin.
- Line 11 displays the value of the my_coin object's __sideup attribute. Because the object's __init__ method set the __sideup attribute to 'Heads', we know that line 11 will display the string 'Heads'.
- Line 14 calls the flip function, passing the my_coin object as an argument. Inside the flip function, the my_coin object's toss method is called.
- Then, line 18 displays the value of the my_coin object's __sideup attribute again. This time, we cannot predict whether 'Heads' or 'Tails' will be displayed because the my_coin object's toss method has been called.

The Unified Modeling Language (UML)

- When designing a class, it is often helpful to draw a UML diagram.
- UML stands for Unified Modeling Language. It provides a set of standard diagrams for graphically depicting object-oriented systems.
- Figure shows the general layout of a UML diagram for a class.
- Notice the diagram is a box that is divided into three sections.
  - The top section is where you write the **name of the class**.
  - The middle section holds a list of the class's **data attributes**.
  - The bottom section holds a list of the class's **methods**.





- **Notice we did not show the self parameter in any of the methods, since it is understood that the self parameter is required.**

An Example

- Write a class named Car that has the following data attributes:

  __year_model

  __make

  __speed
- The Car class should have an __init__ method that accepts the car's year, model and make as arguments.
- These values should be assigned to the object's __year_model and __make data attributes.
- It should also assign 0 to the __speed data attribute.

An Example

- The class should also have the following methods:
  The **accelerate** method should add 5 to the speed data attribute each time it is called.
  The **brake** method should subtract 5 from the speed data attribute each time it is called.
- The Car class should also have **accessor** and **mutator** methods for each attribute.
- Next, design a program that creates a Car object then calls the accelerate method five times. After each call to the accelerate method, get the current speed of the car and display it.
- Then call the brake method five times. After each call to the brake method, get the current speed of the car and display it.

Draw UML Class Diagram for Car Class.

Finding the Classes in a Problem

- When developing an object-oriented program, one of your first tasks is to identify the classes that you will need to create.
- Typically, your goal is to identify the different types of real-world objects that are present in the problem, then create classes for those types of objects within your application.
- Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps:
    - Get a written description of the problem domain.
    - Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
    - Refine the list to include only the classes that are relevant to the problem.

Writing a Description of the Problem Domain

- The problem domain is the set of real-world objects, parties, and major events related to the problem.
- If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself.
- If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.
- For example, suppose we are writing a program that the manager of Joe's Automotive Shop will use to print service quotes for customers.

Writing a Description of the Problem Domain

- Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

Writing a Description of the Problem Domain

- The problem domain description should include any of the following:
  - Physical objects such as vehicles, machines, or products
  - Any role played by a person, such as manager, employee, customer, teacher, student, etc.
  - The results of a business event, such as a customer order, or in this case a service quote
  - Recordkeeping items, such as customer histories and payroll records

Identify All of the Nouns

- The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.)
- Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.
- **Joe's Automotive Shop** services **foreign cars** and specializes in servicing **cars** made by **Mercedes**, **Porsche**, and **BMW**. When a **customer** brings a **car** to the **shop**, the **manager** gets the **customer's name**, **address**, and **telephone number**. The **manager** then determines the **make**, **model**, and **year** of the **car** and gives the **customer** a **service quote**. The **service quote** shows the **estimated parts charges**, **estimated labor charges**, **sales tax**, and **total estimated charges**.

Identify All of the Nouns

- Notice some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them:

- address
- BMW
- car
- cars
- customer
- estimated labor charges
- estimated parts charges
- foreign cars
- Joe's Automotive Shop
- make
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

Refining the List of Nouns

- The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all.
- The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand.
- We will look at the common reasons that a noun can be eliminated from the list of potential classes.
    - Some of the nouns really mean the same thing.
    - **car**, **cars**, and **foreign cars** refer to the general concept of a car.
    - **Joe's Automotive Shop** and **shop** refer to the company "Joe's Automotive Shop."
- Because **car**, **cars**, and **foreign cars** mean the same thing in this problem, we have eliminated **cars** and **foreign cars**.
- Also, because **Joe's Automotive Shop** and **shop** mean the same thing, we have eliminated **Joe's Automotive Shop**.

Refining the List of Nouns

- Some nouns might represent items that we do not need to be concerned with in order to solve the problem.
  - We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
  - We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

Refining the List of Nouns

- Some of the nouns might represent objects, not classes.
- We have eliminated **Mercedes**, **Porsche**,and **BMW** because they are all instances of a car class. That means that these nouns identify objects, not classes.
- Some of the nouns might represent simple values that can be assigned to a variable and do not require a class.
- If a noun represents a type of item that would not have any identifiable data attributes or methods, then it can probably be eliminated from the list.
- To help determine whether a noun represents an item that would have data attributes and methods, ask the following questions about it:
  - Would you use a group of related values to represent the item's state?
  - Are there any obvious actions to be performed by the item?

Refining the List of Nouns

- If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable.
- If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address**, **estimated labor charges**, **estimated parts charges**, **make**, **model**, **name**, **sales tax**, **telephone number**, **total estimated charges**, and **year**.
- These are all simple string or numeric values that can be stored in variables.
- We have eliminated everything except **car**, **customer**, and **service quote**.
- This means that in our application, we will need classes to represent cars, customers, and service quotes.

Identifying a Class's Responsibilities

- Once the classes have been identified, the next task is to identify each class's responsibilities.
- A class's responsibilities are:
  - the things that the class is responsible for knowing.
  - the actions that the class is responsible for doing.
- When you have identified the things that a class is responsible for knowing, then you have identified the class's data attributes.
- Likewise, when you have identified the actions that a class is responsible for doing, you have identified its methods.

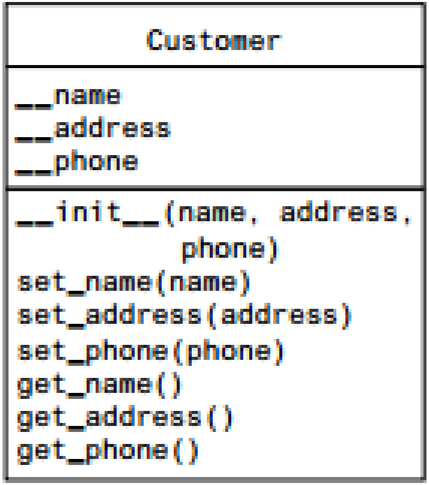
The Customer Class

- In the context of our problem domain, what must the Customer class know?
- The description directly mentions the following items, which are all data attributes of a customer:
  - the customer's name
  - the customer's address
  - the customer's telephone number
- These are all values that can be represented as strings and stored as data attributes.
- The Customer class can potentially know many other things.
- **One mistake that can be made at this point is to identify too many things that an object is responsible for knowing.**
- In some applications, a Customer class might know the customer's email address.
- This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

The Customer Class

- Now, let's identify the class's methods. In the context of our problem domain, what must the Customer class do? The only obvious actions are:
  initialize an object of the Customer class.
  set and return the customer's name.
  set and return the customer's address.
  set and return the customer's telephone number.
- From this list we can see that the Customer class will have an **__init__** method, as well as **accessors** and **mutators** for the data attributes.

The Customer Class

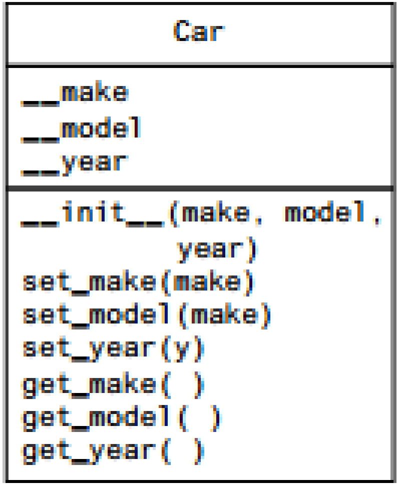


```
# Customer class
class Customer:
    def __init__(self, name, address, phone):
        self.__name = name
        self.__address = address
        self.__phone = phone

    def set_name(self, name):
        self.__name = name

    def set_address(self, address):
        self.__address = address

    def set_phone(self, phone):
        self.__phone = phone

    def get_name(self):
        return self.__name

    def get_address(self):
        return self.__address

    def get_phone(self):
        return self.__phone
```

The Car Class

- In the context of our problem domain, what must an object of the Car class know?
- The following items are all data attributes of a car and are mentioned in the problem domain:
  - the car's make
  - the car's model
  - the car's year
- Now let's identify the class's methods. In the context of our problem domain, what must the Car class do? Once again, the only obvious actions are the standard set of methods that we will find in most classes (an **__init__ method**, **accessors**, and **mutators**). Specifically, the actions are:
  - initialize an object of the Car class.
  - set and return the car's make.
  - set and return the car's model.
  - set and return the car's year.

The Car Class



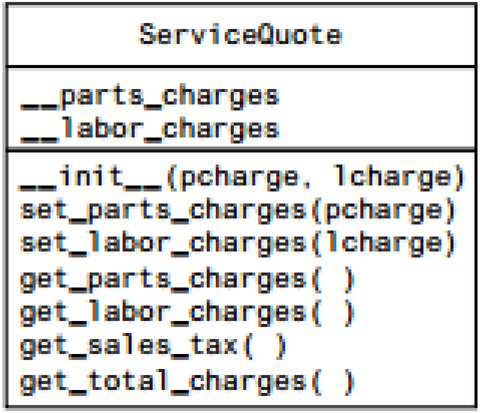
```
Car
__make
__model
__year
__init__(make, model,
         year)
set_make(make)
set_model(make)
set_year(y)
get_make( )
get_model( )
get_year( )
```

```python
# Car class
class Car:
    def __init__(self, make, model, year):
        self.__make = make
        self.__model = model
        self.__year = year

    def set_make(self, make):
        self.__make = make

    def set_model(self, model):
        self.__model = model

    def set_year(self, year):
        self.__year = year

    def get_make(self):
        return self.__make

    def get_model(self):
        return self.__model

    def get_year(self):
        return self.__year
```

The ServiceQuote Class

- In the context of our problem domain, what must an object of the ServiceQuote class know?
- The problem domain mentions the following items:
  the estimated parts charges
  the estimated labor charges
  the sales tax
  the total estimated charges
- The methods that we will need for this class are an __**init**__ method and the **accessors** and **mutators** for the estimated parts charges and estimated labor charges attributes.
- In addition, the class will need methods that **calculate and return the sales tax and the total estimated charges.**

The ServiceQuote Class



```
# Constant for the sales tax rate
TAX_RATE = 0.05

# ServiceQuote class
class ServiceQuote:
    def __init__(self, pcharge, lcharge):
        self.__parts_charges = pcharge
        self.__labor_charges = lcharge

    def set_parts_charges(self, pcharge):
        self.__parts_charges = pcharge

    def set_labor_charges(self, lcharge):
        self.__labor_charges = lcharge

    def get_parts_charges(self):
        return self.__parts_charges

    def get_labor_charges(self):
        return self.__labor_charges

    def get_sales_tax(self):
        return __parts_charges * TAX_RATE

    def get_total_charges(self):
        return __parts_charges + __labor_charges + \
               (__parts_charges * TAX_RATE)
```

An Example

- An instructor wants to determine and summarize automatically letter grades of a course. The instructor may have more than one course. Therefore, when the program start the course name should be ask. Then, the instructor enters letter grades from "A" to "F". When the instructor presses a special key, the program must stop input process and calculates total number of each letter grade. Lastly, the program must summarize the results.
- Design a class for the problem.
- Use UML Class Diagram to depict the class.