9 Object Oriented Programming

Learning Outcome

Fundamentals of Object-Oriented Programming
Define and understand classes and objects
Understand encapsulation and how classes support information hiding and implementation independence
Understand inheritance and how it promotes software reuse
Understand polymorphism and how it enables code generalisation.
Exclude: method overloading and multiple inheritance

9.1 Procedural and Object-Oriented Programming

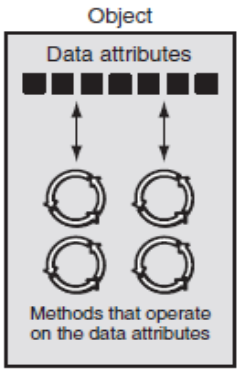
9.1.1 Procedural Programming: Writing programs made of functions that perform specific tasks

- Procedures typically operate on data items that are separate from the procedures
- Data items commonly passed from one procedure to another
- Focus: to create procedures that operate on the program's data

9.1.2 Object-Oriented Programming: focused on creating objects

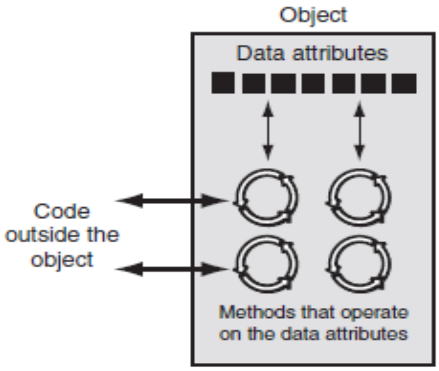
Object: entity that contains data and procedures

- Data is known as **data attributes** and procedures are known as **methods**
- Methods perform operations on the data attributes



Encapsulation: combining data and procedure into a single object

Data hiding: object's data attributes are hidden from code outside the object and access is restricted to the object's methods
- Protects from accidental corruption
- Outside code does not need to know internal structure of the object



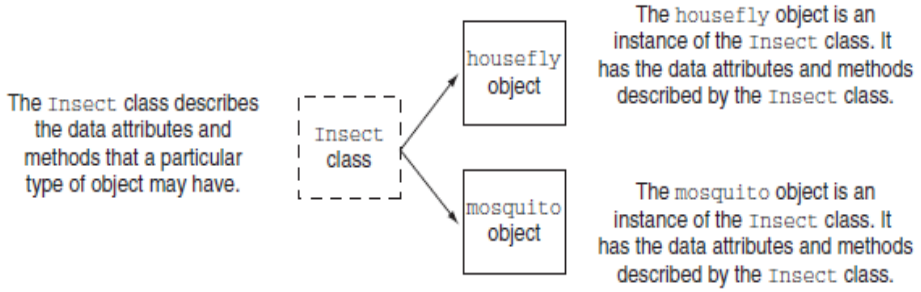
An Everyday Example of an Object
- Data attributes: define the state of an object
  - Example: clock object would have `second`, `minute`, and `hour` data attributes
- Public methods: allow external code to manipulate the object
  - Example: `set_time`, `set_alarm_time`
- Private methods: used for object's inner workings

9.2 Class vs Instance/Object

Class: code that specifies the data attributes and methods of a particular type of object
- Similar to a blueprint of a house or a cookie cutter

Instance: an object created from a class
- Similar to a specific house built according to the blueprint or a specific cookie
- There can be many instances of one class

The housefly and mosquito objects are instances of the Insect class



9.3 Class Definitions

set of statements that define a class's methods and data attributes

- Format: begin with `class Class_name:`
  - Class names often start with uppercase letter
- Method definition like any other python function definition
  - `self` parameter: required in every method in the class – references the specific object that the method is working on

9.3.1 Initializer Method

Also called class's constructor, this method automatically executed when an instance of the class is created

- Initializes object's data attributes and assigns `self` parameter to the object
- Format: `def __init__ (self):`
- Usually the first method in a class definition

(coin.py)

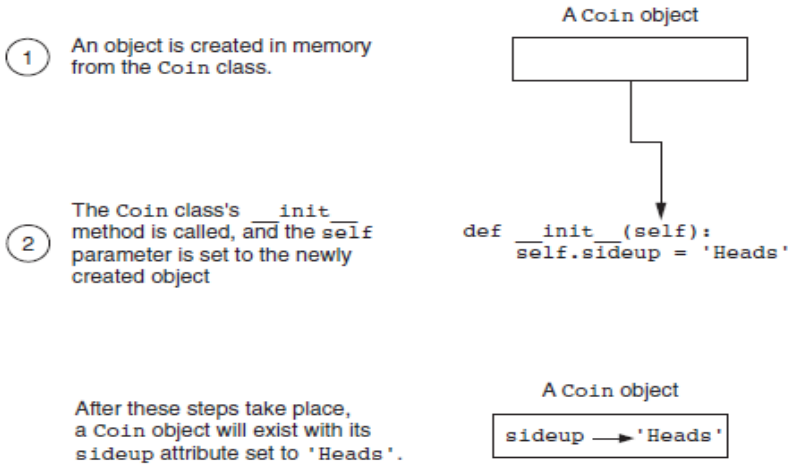
```
import random

# The Coin class simulates a coin that can be flipped.

class Coin:

     # initializes the sideup data attribute with 'Heads'.

     def __init__(self):
          self.sideup = 'Heads'

     # The toss method generates a random number in the range of 0
     # to 1. If the number is 0, then sideup is set to 'Heads'.
     # Otherwise, sideup is set to 'Tails'.

     def toss(self):
          if random.randint(0, 1) == 0:
               self.sideup = 'Heads'
          else:
               self.sideup = 'Tails'
```

9.3.2 Creating an Instance

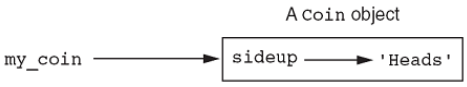
- To create a new instance of a class call the initializer method
  - Format: *`My_instance = Class_Name()`*
- To call any of the class methods using the created instance, use dot notation
  - Format: *`My_instance.method()`*
  - Because the `self` parameter references the specific instance of the object, the method will affect this instance, reference to `self` is passed automatically

```
# The main function.
def main():
     # Create an object from the Coin class.
     my_coin = Coin()

     # Display the side of the coin that is facing up
     print ('This side is up:', my_coin.sideup)

     # Toss the coin.
     print ('I am going to toss the coin ten times:')
     for count in range(10):
          my_coin.toss()
          print (my_coin.sideup)

# Call the main function.
main()
```

Actions caused by the Coin() expression



The my_coin variable references a Coin object



9.3.3 Data Hiding

Hiding Attributes

An object's data attributes should be private. In Python, a name prefixed with an underscore (e.g. _spam) is a commonly used convention to denote a private attribute or method.

Accessor and Mutator Methods

All of a class's data attributes are private and provide methods to access and change them.

- Accessor/Getter methods: return a value from a class's attribute without changing it
  - Safe way for code outside the class to retrieve the value of attributes
- Mutator/Setter methods: store or change the value of a data attribute

Storing Classes in Modules

Classes can be stored in modules

- Filename for module must end in .py
- Module can be imported to programs that use the class

The BankAccount Class

Class methods can have multiple parameters in addition to self

- For `__init__`, parameters needed to create an instance of the class
  - Example: a `BankAccount` object is created with a balance
  - When called, the initializer method receives a value to be assigned to the `_balance` attribute
- For other methods, parameters needed to perform required task
  - Example: `deposit` method amount to be deposited

(account.py)

```
# The BankAccount class simulates a bank account.

class BankAccount:

      # The __init__ method accepts an argument for the account's
      # balance. It is assigned to the _balance attribute.
      def __init__(self, bal):
            self._balance = bal

      # The deposit method makes a deposit into the account.
      def deposit(self, amount):
            self._balance += amount

      # The withdraw method withdraws an amount from the account.
      def withdraw(self, amount):
            if self._balance >= amount:
                  self._balance -= amount
            else:
                  print ('Error: Insufficient funds')

      # The get_balance method returns the account balance.
      def get_balance(self):
            return self._balance
```

(**account_test.py**)

```
# This program demonstrates the BankAccount class.

import account

def main():
      # Get the starting balance.
      start_bal = float(input('Enter your starting balance: '))

      # Create a BankAccount object.
      savings = account.BankAccount(start_bal)

      # Deposit the user's paycheck.
      pay = float(input('How much were you paid this week? '))
      print('I will deposit that into your account.')
      savings.deposit(pay)

      # Display the balance.
      print('Your account balance is ${savings.get_balance():.2f}.')

      # Get the amount to withdraw.
      cash = int(input('How much would you like to withdraw? '))
      print('I will withdraw that from your account.')
      savings.withdraw(cash)

      # Display the balance.
      print(f'Your account balance is${savings.get_balance():.2f}.')

# Call the main function.
main()
```

The __str__ method

- Object's state: the values of the object's attribute at a given moment
- `__str__` method: displays the object's state
    - Automatically called when you pass the object's name to the `print` statement
    - Automatically called when the object is passed as an argument to the `str` function

```
# The __str__ method returns a string indicating the object's state.

def __str__(self):
    state_string = f'Your account balance is ${self._balance:.2f}.'
    return  state_string
```

```
# Display the balance.

print(savings)
```

9.4 Working With Instances

- Instance attribute: belongs to a specific instance of a class
  - Created when a method uses the `self` parameter to create an attribute
- If many instances of a class are created, each would have its own set of attributes

(**simulation.py**)

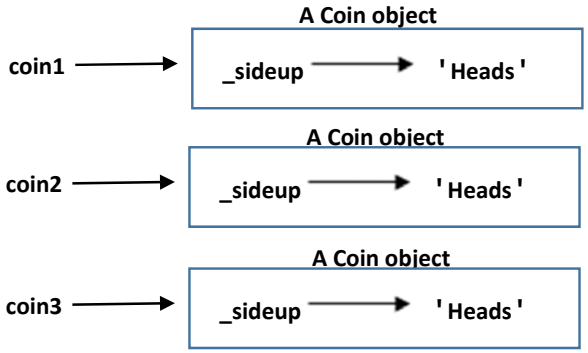
```
import random

# The Coin class simulates a coin that can be flipped.

class Coin:

     # The __init__ method initializes the _sideup data attribute
     # with 'Heads'.

     def __init__(self):
        self._sideup = 'Heads'

     # The toss method generates a random number in the range of 0
     # to 1. If the number is 0, then sideup is set to 'Heads'.
     # Otherwise, sideup is set to 'Tails'.

     def toss(self):
        if random.randint(0, 1) == 0:
           self._sideup = 'Heads'
        else:
           self._sideup = 'Tails'

     # The get_sideup method returns the value referenced by sideup

     def get_sideup(self):
        return  self._sideup
```

(**coin_demo.py**)

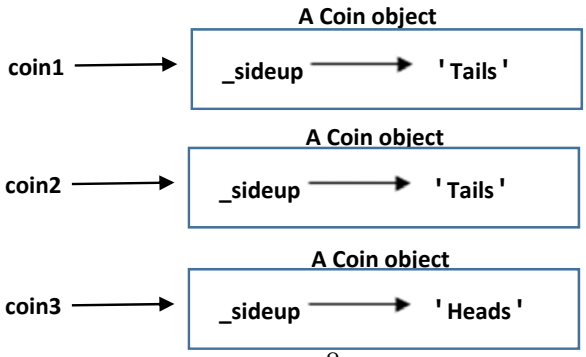
```
# This program imports the simulation module and creates three
# instance of the Coin class.

import simulation

def main():
     # Create three objects from the Coin class.
     coin1 = simulation.Coin()
     coin2 = simulation.Coin()
     coin3 = simulation.Coin()
```

```
    # Display the side of each coin that is facing up.
    print ('I have three coins with these sides up:')
    print (coin1.get_sideup())
    print (coin2.get_sideup())
    print (coin3.get_sideup())
    print ()

    # Toss the coin.
    print ('I am tossing all three coins...')
    print ()
    coin1.toss()
    coin2.toss()
    coin3.toss()

    # Display the side of each coin that is facing up.
    print ('Now here are the sides that are up:')
    print (coin1.get_sideup())
    print (coin2.get_sideup())
    print (coin3.get_sideup())
    print ()

# Call the main function.
main()
```

The coin1, coin2, and coin3 variables reference three Coin objects



The objects after the toss method



Passing Objects as Arguments

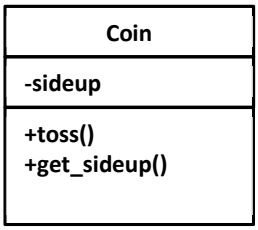
- Methods and functions often need to accept objects as arguments
- When you pass an object as an argument, you are actually passing a reference to the object. The receiving method or function has access to the actual object
- Methods of the object can be called within the receiving function or method, and data attributes may be changed using mutator methods

9.5 Techniques for Designing Classes

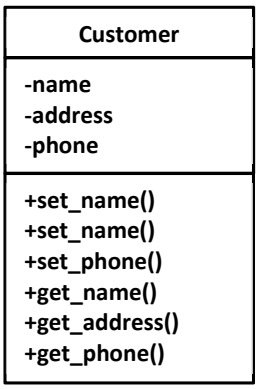
Class diagram is a standard diagram for graphically depicting object-oriented systems.

General layout: box divided into three sections:
- Top section: name of the class
- Middle section: list of data attributes
- Bottom section: list of class methods
- Access Modifier :
  - \- (minus sign) private
  - \+ (plus sign) public

Class diagram for the Coin class



Class diagram for the Customer Class



9.6 Rules of Thumb for Defining a Simple Class

- Before writing a line of code, think about the behavior and attributes of the objects of new class
- Choose an appropriate class name and develop a short list of the methods available to users
- Write a short script that appears to use the new class in an appropriate way
- Choose appropriate data structures for attributes
- Fill in class template with `__init__` and `__str__`
- Complete and test remaining methods incrementally
- Document your code

Tutorial 9A

1. What is encapsulation?

2. What is the difference between a class and an instance of a class?

3. Why should an object's data attributes be hidden from code outside the class? How do we hide an attribute in a python class?

4. The following statement calls an object's method. What is the name of the method? What is the name of the variable that references the object?

```
wallet.get_dollar()
```

5. When the `__init__` method executes, what does the `self` parameter reference?

6. How do you call the __str__ method?

7. Suppose `my_car` is the name of a variable that references an object, and `go` is the name of a method. Write a statement that uses the `my_car` variable to call the `go` method. (You do not have to pass any arguments to the go method.)

8. Draw a class diagram for `Book`. The `Book` class has data attributes for title, author's name, publisher's name and their corresponding accessor and mutator methods for each attribute.

Assignment 9A

1. Write a class definition named `Book`. The Book class should have data attributes for a book's title, the author's name, and the publisher's name. The class should also have the following:

   a) An `__init__` method for the class. The method should accept an argument for each of the data attributes.

   b) Accessor and mutator methods for each data attribute.

   c) An `__str__` method that returns a string indicating the state of the object.

2. Write a class named `Pet`, which should have the following data attributes:

   • `_name` (for the name of a pet)
   • `_animal_type` (for the type of animal that a pet is. Example values are 'Dog', 'Cat' and 'Bird')
   • `_age` (for the pet's age)

   The Pet class should have an `__init__` method that creates these attributes. It should also have the following methods:
   • `set_name`: assigns a value to the `_name` field
   • `set_animal_type:` assigns a value to the `_animal_type` field
   • `set_age:` assigns a value to the `_age` field.
   • `get_name:` returns the value of the `_name` field
   • `get_type:` returns the value of the `_animal_type` field
   • `get_age`: returns the value of the `_age` field

   Once you have written the class, write a program that creates an object of the class and prompts the user to enter the name, type and age of his or her pet. This data should be stored as the object's attributes. Use the object's accessor methods to retrieve the pet's name, type and age and display the data on the screen.

3. Write a class named `Car` that has the following data attributes:
   • `_year_model` (for the car's year model)
   • `_make` (for the make of the car)
   • `_speed` (for the car's current 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.

   The class should also have the following methods:

•`accelerate`: add 5 to the speed data attribute each time it is called.
•`brake`: subtract 5 from the speed data attribute each time it is called.
•`get_speed`: return the current speed.

Next, design a program that creates a `Car` object, and 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.

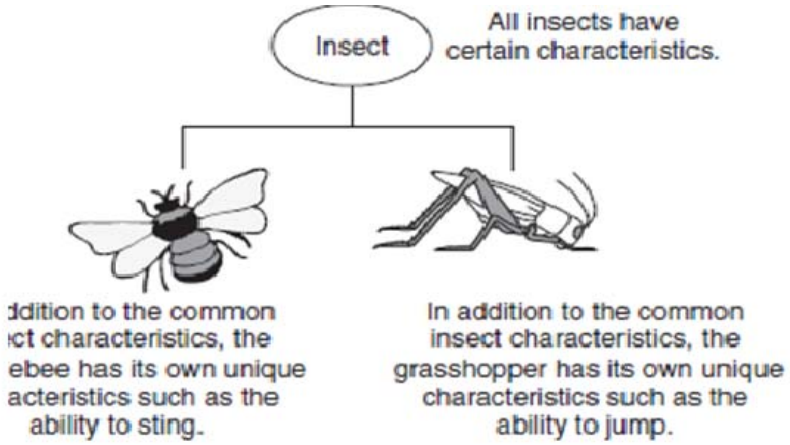
9.7 Inheritance

In the real world, many objects are a specialized version of more general objects.

Example: grasshoppers and bees are specialized types of insect
In addition to the general insect characteristics, they have unique characteristics:

- Grasshoppers can jump
- Bees can sting, make honey, and build hives

Bumblebees and grasshoppers are specialized versions of an insect.



9.7.1 Inheritance and the "Is a" relationship

- "Is a" relationship: exists when one object is a specialized version of another object
  - specialized object has all the characteristics of the general object plus unique characteristics
  - Example: Rectangle is a shape, Car is a vehicle

- Inheritance: used to create an "is a" relationship between classes
- Superclass (base class): a general class

- Subclass (derived class): a specialized class, an extended version of the superclass
  - Inherits attributes and methods of the superclass
  - New attributes and methods can be added

For example, need to create classes for cars, pickup trucks, and SUVs

- All are automobiles
  - Have a make, year model, mileage, and price
  - This can be the attributes for the base class
- In addition:
  - Car has a number of doors
  - Pickup truck has a drive type
  - SUV has a passenger capacity

In a class definition for a subclass:

- To indicate inheritance, the superclass name is placed in parentheses after subclass name
  - Example: `class Car(Automobile):`
- The initializer method of a subclass calls the initializer method of the superclass and then initializes the unique data attributes
- Add method definitions for unique methods

(vehicles.py)

```
# The Automobile class holds general data about an automobile.

class Automobile:

      # The __init__ method accepts arguments for the make, model,
      # mileage, and  price. It initializes the data attributes with
      # these values.

      def __init__(self, make, model, mileage, price):
            self._make = make
            self._model = model
            self._mileage = mileage
            self._price = price

      # The following methods are mutators for the class's data
      # attributes.

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

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

      def set_mileage(self, mileage):
            self._mileage = mileage

      def set_price(self, price):
            self._price = price
```

```
      # The following methods are the accessors for the class's data
      # attriburtes.

      def get_make(self):
            return self._make

      def get_model(self):
            return self._model

      def get_mileage(self):
            return self._mileage

      def get_price(self):
            return self._price

# The Car class represents a car. It is a subclass of the Automobile
# class.

class Car(Automobile):
      # The __init__ method accepts arguments for the car's make,
      # model, mileage, price, and doors.

      def __init__(self, make, model, mileage, price, doors):
            # Call the superclass's __init__ method and pass the
            # required arguments.
            super().__init__(make, model, mileage, price)

            # Initialize the __doors attribute.
            self._doors = doors

      # The set_doors method is the mutator for the __doors
      # attribute.

      def set_doors(self, doors):
            self._doors = doors

      # The get_doors method is the accessor for the __doors
      # attribute.

      def get_doors(self):
            return self._doors

# The Truck class represents a pickup truck. It is a subclass of the
# Automobile class.

class Truck(Automobile):
      # The __init__ method accepts arguments for the car's make,
      # model, mileage, price, and drive type.

      def __init__(self, make, model, mileage, price, drive_type):
            # Call the superclass's __init__ method and pass the
            # required arguments.
            super().__init__(make, model, mileage, price)
```

```
            # Initialize the _drive_type attribute.
            self._drive_type = drive_type

      # The set_drive_type method is the mutator for the
      # _drive_type attribute.

      def set_drive_type(self, drive_type):
            self._drive_type = drive_type

      # The get_drive_type method is the accessor for the
      # _drive_type attribute.

      def get_drive_type(self):
            return self._drive_type
# The SUV class represents a sport utility vehicle. It is a subclass
# of the Automobile class.

class SUV(Automobile):
      # The __init__ method caccepts arguments for the car's make,
      # model, mileage, #price, and passenger capacity.

      def __init__(self, make, model, mileage, price, pass_cap):
            # Call the superclass's __init__ method and pass the
            # required arguments.
            super().__init__(make, model, mileage, price)

            # Initialize the _pass_cap attribute.
            self._pass_cap = pass_cap

      # The set_pass_cap method is the mutator for the _pass_cap
      # attribute.

      def set_pass_cap(self, pass_cap):
            self._pass_cap = pass_cap

      # The get_pass_cap method is the accessor for the _pass_cap
      # attribute.

      def get_pass_cap(self):
            return self._pass_cap
```

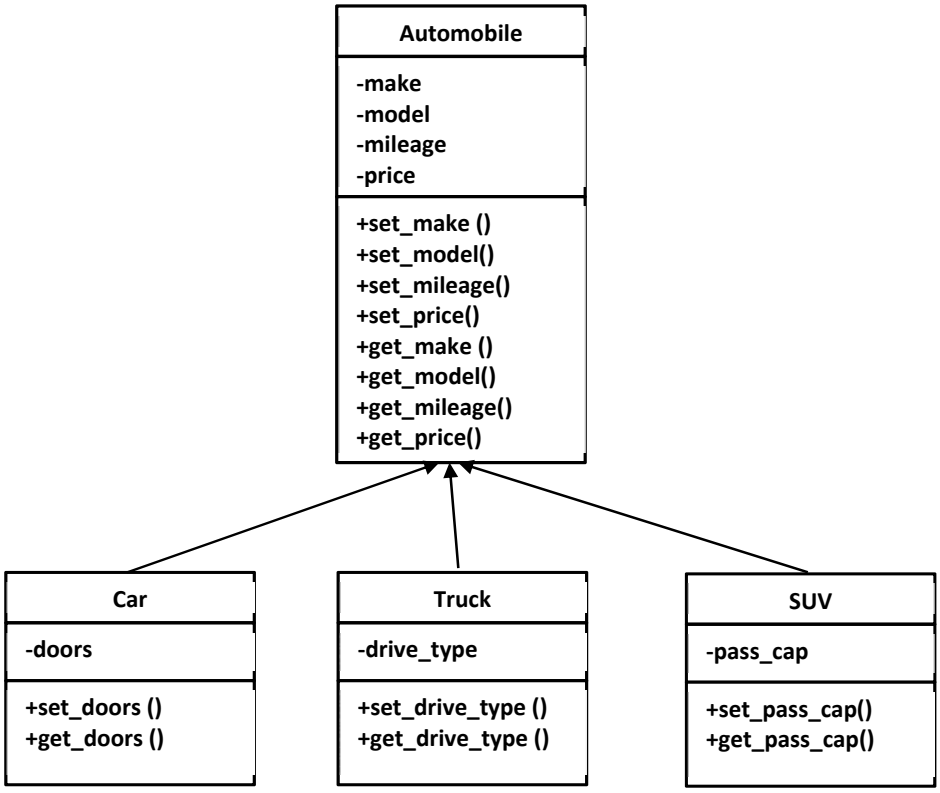
(car_truck_suv_demo.py)

```
# This program creates a Car object, a Truck object, and an SUV
# object.

import vehicles

def main():
     # Create a Car object for a used 2001 BMW with 70,000 miles,
     # priced at $15,000, with 4 doors.
     car = vehicles.Car('BMW', 2001, 70000, 15000.0, 4)

     # Create a Truck object for a used 2002 Toyota pickup with
     # 40,000 miles, priced at $12,000, with 4-wheel drive.
     truck = vehicles.Truck('Toyota', 2002, 40000, 12000.0, '4WD')

     # Create an SUV object for a used 2000 Volvo with 30,000
     # miles, priced at $18,500, with 5 passenger capacity.
     suv = vehicles.SUV('Volvo', 2000, 30000, 18500.0, 5)

     print ('USED CAR INVENTORY')
     print ('==================')

     # Display the car's data.
     print ('The following car is in inventory:')
     print ('Make:', car.get_make())
     print ('Model:', car.get_model())
     print ('Mileage:', car.get_mileage())
     print ('Price:', car.get_price())
     print ('Number of doors:', car.get_doors())
     print ()

     # Display the truck's data.
     print ('The following pickup truck is in inventory.')
     print ('Make:', truck.get_make())
     print ('Model:', truck.get_model())
     print ('Mileage:', truck.get_mileage())
     print ('Price:', truck.get_price())
     print ('Drive type:', truck.get_drive_type())
     print ()

     # Display the SUV's data.
     print ('The following SUV is in inventory.')
     print ('Make:', suv.get_make())
     print ('Model:', suv.get_model())
     print ('Mileage:', suv.get_mileage())
     print ('Price:', suv.get_price())
     print ('Passenger Capacity:', suv.get_pass_cap())

# Call the main function.
main()
```

9.7.2 Inheritance in Class Diagrams

In Class diagram, show inheritance by drawing a line with an open arrowhead from subclass to superclass

Class diagram showing inheritance



9.8 Polymorphism

- Polymorphismanan object's ability to take different forms
- Essential ingredients of polymorphic behavior:
  - Ability to define a method in a superclass and override it in a subclass
    - Subclass defines method with the same name
  - Ability to call the correct version of overridden method depending on the type of object that is used to call it

- In previous inheritance examples showed how to override the `__init__` method
  - Called superclass `__init__` method and then added onto that
- The same can be done for any other method. The method can call the superclass equivalent and add to it, or do something completely different

(animals.py)

```
# The Mammal class represents a generic mammal.

class Mammal:

      # The __init__ method accepts an argument for the mammal's
      # species.
      def __init__(self, species):
            self._species = species

      # The show_species method displays a message indicating the
      # mammal's species.
      def show_species(self):
            print ('I am a', self._species)

      # The make_sound method is the mammal's way of making a
      # generic sound.
      def make_sound(self):
            print ('Grrrrr')


# The Dog class is a subclass of the Mammal class.

class Dog(Mammal):

      # The __init__ method calls the superclass's  __init__ method
      # passing 'Dog' as the species.

      def __init__(self):
            super().__init__('Dog')

      # The make_sound method overrides the superclass's make_sound
      # method.

      def make_sound(self):
            print ('Woof! Woof!')


# The Cat class is a subclass of the Mammal class.

class Cat(Mammal):

      # The __init__ method calls the superclass's  __init__ method
      # passing 'Cat' as the species.

      def __init__(self):
            super().__init__('Cat')

      # The make_sound method overrides the superclass's make_sound
      # method.

      def make_sound(self):
            print ('Meow')
```

Here is the example of code that creates a Mammal object, a Dog object, and calls the methods:

```
>>> import animals

>>> mammal = animals.Mammal('regular mammal')
>>> mammal.show_species()
I am a regular mammal
>>> mammal.make_sound()
Grrrrr

>>> dog = animals.Dog()
>>> dog.show_species()
I am a Dog
>>> dog.make_sound()
Woof! Woof!
```

Tutorial 9B

1. What does a subclass inherit from its superclass?

2. What are the benefits of having class `B` extend or inherit from class `A`?

3. Look at the following class definition. What is the name of the superclass? What is the name of the subclass?

```
class Tiger(Felis):
```

4. Describe what the `__init__` method should do in a class that extends another class.

5. Look at the following class definition:

```
class Beverage:
          def __init__(self,  bev_name):
               self._bev_name = bev_name
```

   Write the code for a class named `Cola` that is a subclass of the `Beverage` class. The `Cola` class's `__init__` method should call the `Beverage` class's `__init__` method, passing 'cola' as an argument.

6. What is an overridden method?

7. Look at the following class definitions:

```
class Plant:
     def __init__(self,  plant_type):
          self._plant_type = plant_type

     def  message(self):
          print ("I'm a plant")
```

```
class Tree(Plant):
      def __init__(self):
            super().__init__('tree')

      def message(self):
            print ("I'm a tree")
```

Given these class definitions, what will the following statements display?

```
p = Plant('sapling')
t = Tree()
p.message()
t.message()
```

8. Class `B` extends class `A`. Class `A` defines an `__str__` method that returns the string representation of its instance variables. Class `B` defines a single variable named `_age`. Write the code to define the `__str__` method for class `B`. This method should return the combined string information from both classes. Label the data for `_age` with the string "Age: ".

Assignment 9B

1. `Employee` and `ProductionWorker` Classes

   Write an `Employee` class that keeps data attributes for the following pieces of information:

   - Employee name
   - Employee number

   Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data attributes for the following information:

   - Shift number (an integer, such as 1,2, or 3)
   - Hourly pay rate

   The workday is divided into two shifts: day and night. The shift attribute will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate accessor and mutator methods for each class.

   Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data attributes. Store the data in the object and then use the object's accessor methods to retrieve it and display it on the screen.

2 `ShiftSupervisor` Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in question 1. The `ShiftSupervisor` class should keep a data attribute for the annual salary and a data attribute for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3 `Person` and `Customer` Classes

Write a class named `Person` with data attributes for a person's name, address and telephone number. Next, write a class name `Customer` that is a subclass of the `Person` class. The `Customer` class should have a data attribute for a customer number and a Boolean data attribute indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.