CS-119 Lab #8

Expected Learning Objectives

- Classes and objects
- Implementing class models
- Protected access
- Inheritance – superclasses and subclasses
- Class constructors
- Class get() and set() methods
- Abstract methods
- Overriding methods

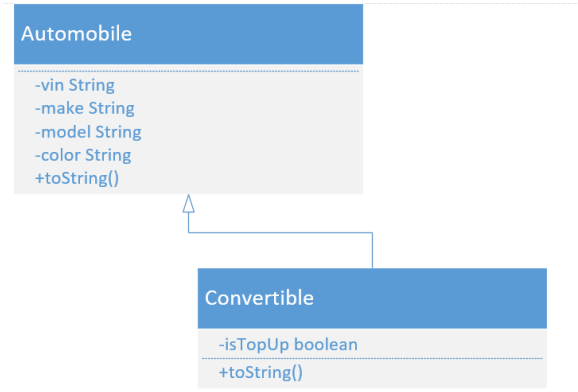
Overview

This lab will give you some hands-on experience creating user defined classes using inheritance in Python and creating instances in a "test fixture" application. Classes such as these are commonly implemented, tested and debugged in a small scale test environment and then later integrated into a larger scale project.

Exercise 1 Automobile

This exercise is based on exercise 1 on page 333 of the textbook.

Open a new UML drawing in Draw.io. Drag and drop 2 classes onto the page. Fill in the class name, attributes and properties. From the exercise, we are given the Automobile class properties are Vehicle Identification Number (VIN), make, model and color. We'll add a to_string() method to return a string of all the properties. The Convertible class has the added property to indicate whether the top is up and it will have a to_string() method we'll override. When showing an inheritance relationship in a class model, the key thing is to have the inheritance line with the arrow pointing to the parent or base class. The class model should look something like this:



Automobile Class Python Code

Create a new code file in Python called Automobile and a class called Automobile, code the properties, constructor, getters/setters, and to_string() method. This class will be coded like the ones we did in the last chapter with the exception that our attributes will be declared with *protected* access. Protected access is just like private except that it allows derived (inherited) classes direct access (see page 316) .

```python
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 1 Automobile class implementation


# The properties below are public by default
# a single underscore _var_name before a variable name makes it protected
# a double underscore __var_name before a variable name makes it private
#
class Automobile:
    # custom constructor. Constructors are named __init__()
    def __init__(self, vin, make, model, color):
        # set the attributes
        # Note the double underscore __ before the variable name makes them private
        self._vin = vin
        self._make = make
        self._model = model
        self._color = color


    # getter and setter methods for the various properties
    def get_vin(self):
        return self._vin

    def set_vin(self, vin):
        self._vin = vin

    def get_make(self):
        return self._make

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

    def get_model(self):
        return self._model

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

    def get_color(self):
        return self._color

    def set_color(self, color):
        self._color = color

    # instance method
    def to_string(self):
        return "VIN: " + self._vin + " Make: " \
               + self._make + " Model: " + self._model \
               + " Color: " + self._color
```

The Convertible Class

Now we're going to create a new code file called Convertible in Python. The added twist is in the class header where we have *class Convertible(Automobile),* and in the constructors where we first call the inherited class constructor by using the `super().__init__` statement and then add any new initialization behavior needed.

```
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 1 Automobile class implementation

# import the parent class
from Automobile import Automobile

# The properties below are public by default
# a single underscore _var_name before a variable name makes it protected
# a double underscore __var_name before a variable name makes it private
#

# Declare the Convertible class and extend (inherit from) automobile
class Convertible(Automobile):
    # custom constructor. Constructors are named __init__()
    def __init__(self, vin, make, model, color, is_top_up):
        # we are overriding the Automobile constructor so call the inherited constructor
        super().__init__(vin, make, model, color)
        # add any additional code needed for the convertible constructor
        self._is_top_up = is_top_up


    # getter and setter methods for the various properties
    def get_is_top_up(self):
        return self._is_top_up

    def set_is_top_up(self, is_top_up):
        self._is_top_up = is_top_up

    # public instance method to format the status of the top up/down
    def fmt_top_status(self):
        status = "No" # set no by default
        if self._is_top_up == True:
            status = "Yes"
        return status


    # instance method
    def to_string(self):
        # We are overriding this method. Call the inherited and then add new behavior
        return super().to_string() \
            + "\nIs top up? " + self.fmt_top_status()
```

Car Test Fixture

Create a new Python code file called CarTest.py. A screen shot of the code is provided below. There shouldn't be much in the way of surprises for the Python code.

```
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 1 Convertible class test fixture

# import the Convertible class
from Convertible import Convertible

make = input("Enter make: ")
model = input("Enter model: ")
vin = input("Enter VIN: ")
color = input("Enter color: ")
yn = input("Is the top up (yes/no): ")

if yn.upper() == "YES":
    is_top_up = True
else:
    is_top_up = False

# create an instance of a Convertible
my_car = Convertible(vin, make, model, color, is_top_up)

# invoke the to_string() method and display everything
print(my_car.to_string())
```

Exercise 2 Race Car class

Create a Python code file called Racecar.py and a class called RaceCar that inherits from the Automobile class you did in exercise 1. It will have one additional property called number of horse power. Add the class model to the Automobile class model in the last exercise and show an inheritance relationship.

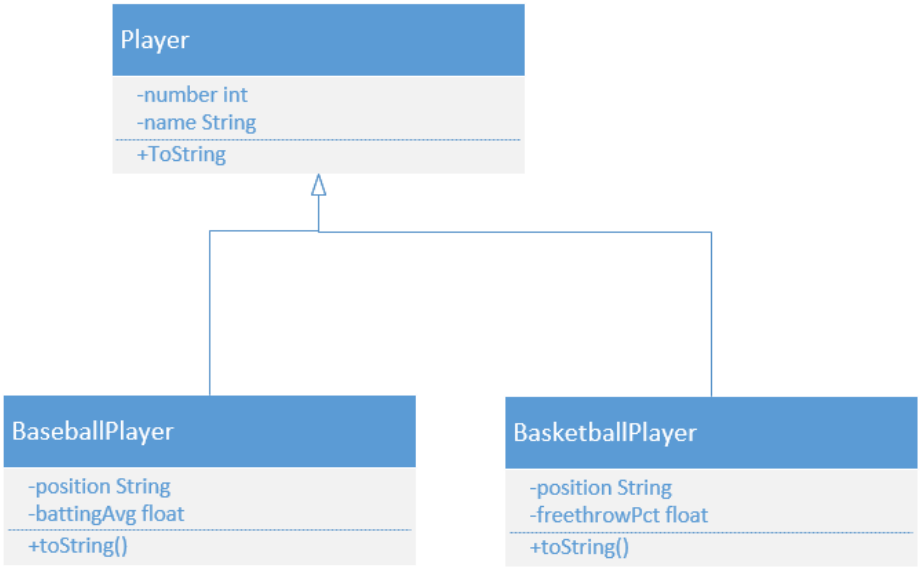
You may add the code needed to prompt the user for the number of horse power, create a race car object and call the to _string() method to your CarTest test fixture application.

Exercise 3 Player (Page 334, Exercise 4)

In this exercise, you'll create a general class called Player and then derive (inherit) 2 classes, BaseballPlayer and BasketballPlayer, from it.

Player Class Model

Create a new UML class model in Draw.io. Drag and drop a total of 3 classes onto the page, fill in the class names, attributes for each class, and a toString() method for each class. Show an inheritance relationship where the Player class is the parent class for BaseballPlayer and BasketballPlayer.



Player Python Code

Create a new Python code file called Player.py. Code a new class called Player. Implement the attributes, constructors, getter and setter methods, and the to_string() method.

One added twist to the code is that we will add some validation to setNumber to make sure a player's number is within some "reasonable" range. Notice the custom constructor will call setNumber(). The player class should look like the Python code screen shot below.

```python
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 2 player class implementation


# The properties below are public by default
# a single underscore _var_name before a variable name makes it protected
# a double underscore __var_name before a variable name makes it private
#
class Player:

    # custom constructor. Constructors are named __init__()
    def __init__(self, name, number):
        # set the attributes
        # Note the single underscore _ before the variable name makes them protected
        self._name = name
        self.set_number(number) # note we call the setter method here


    # getter and setter methods for the various properties
    def get_name(self):
        return self._name

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

    def get_number(self):
        return self._number

    def set_number(self, number):
        MIN_NUMBER = 1
        MAX_NUMBER = 9999

        # Add validation to make sure number is in the expected range
        if number >= MIN_NUMBER and number <= MAX_NUMBER:
            self._number = number
        else:
            self._number = MIN_NUMBER


    # instance method
    def to_string(self):
        return "Name: " + self._name + " Number: " + str(self._number)
```

BaseballPlayer Class

Create a Python code file called BaseballPlayer.py, create a new class called BaseballPlayer, implement the attributes, constructors, getters/setters and to_string() method. Python screen shots are provided below. In the class header, be sure it extends Player. Notice there's less code by inheriting from another class. Also note the user of the `super()` statement to call the inherited class constructor. In the to_string() method, notice how the statement `super().to_string()` calls the to_string() method in the parent class saving some work.

```
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 2 BaseballPlayer class implementation

# import the parent class
from Player import Player

# The properties below are public by default
# a single underscore _var_name before a variable name makes it protected
# a double underscore __var_name before a variable name makes it private
#

# Declare the BaseballPlayer class and extend (inherit from) Player
class BaseballPlayer(Player):
    # custom constructor. Constructors are named __init__()
    def __init__(self, name, number, position, batting_avg):
        # we are overriding the Player constructor so call the inherited constructor
        super().__init__(name, number)
        # add any additional code needed for the baseball player constructor
        self._position = position
        self._batting_avg = batting_avg


    # getter and setter methods for the various properties
    def get_position(self):
        return self._position

    def set_position(self, position):
        self._position = position

    def get_batting_avg(self):
        return self._batting_avg

    def set_batting_avg(self, batting_avg):
        self._batting_avg = batting_avg


    # instance method
    def to_string(self):
        # We are overriding this method. Call the inherited and then add new behavior
        return super().to_string() + " Position: " + self._position + " Batting Avg: " + str(self._batting_avg)
```

Player Test Fixture

```python
#!/user/bin/env python3

# Cuyamaca College CS-119
# Your name here!
# Lab 8 exercise 2 Player class test fixture

# import the baseball player class
from BaseballPlayer import BaseballPlayer

name = input("Enter name: ")
number = int(input("Enter number: "))
position = input("Enter position: ")
batting_avg = float(input("Enter batting average: "))


# create an instance of a baseball player
my_player = BaseballPlayer(name, number, position, batting_avg)

# invoke the to_string() method and display everything
print(my_player.to_string())
```

Exercise 4

Implement the BasketballPlayer class. This inherits from the Player class and has the added properties position and free throw percentage. You may add the code to prompt for free throw percentage, create the basketball player object and call the to_string() to your PlayerTest test fixture application.

Exercise 5

In this exercise, we will create a base class (superclass) and two inherited classes (subclasses). To keep things reasonably simple, we're going to implement just a few properties and a single method. In real world software development, classes like these would contain *numerous* properties and methods. Our goal is to learn the object oriented design/programming concepts, Python syntax for classes and inheritance relationships.

This exercise is very similar to what did in the previous exercises. This time, the instructions aren't going to be quite as detailed.

1. In Draw.io, create a new drawing using the UML template. Create a class model for a class called *Person* with the following properties: name, address, age. It will have a to_string() method that will display a formatted string of all the properties.

2. In your class model, add a class called employee. Show an inheritance (often called a *generalization*) relationship from the person class. With inheritance, the arrow points to the superclass or base class. In the employee class, add the following properties: job skills and years worked. The employee class will override the to_string() method.

3. Add a class called student to your model. Show an inheritance relationship from the person class. In the student class, add the following properties: major and units completed. The student class will also override the to_string() method..

4. Time for some Python code! Create a code file called Person.py and a class called Person. Implement the properties and methods required.

5. Create a Python code file called Employee.py. Implement the Employee class with the required properties and methods. Be sure to inherit from the Person class!

6. Create a Python code file called Student.py. Implement the Student class with the required properties and methods. Be sure to inherit from the Person class!

7. Create a Python code file called PersonTest.py. Prompt for the inputs needed to test the student and employee classes. Create an instance of each and display the values. Sample program run:

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)]
n32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: C:/Cuyamaca/2019FallCS119/LabsDev/Lab08/PersonTest.py =======
Enter name: John Smith
Enter address: 123 Main St
Enter age: 25
Enter job skills: Chef
Enter years worked: 5
Enter major: CS
Enter units completed: 12
Name: John Smith Address: 123 Main St Age: 25 Job Skills: Chef Years worked: 5.0
Name: John Smith Address: 123 Main St Age: 25 Major: CS Units completed: 12.0
```

Grading Criteria:

Deliverable	Points	Breakdown
Car exercise 1	10	Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships
Race car exercise 2	5	All properties and methods implemented per lab handout, integrated into the Car exercise. UML class model added to car exercise class model.
Player exercise 3	10	Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships
Basketball player class exercise 4	5	All properties and methods implemented per lab handout, integrated into the Player exercise. UML class model added to car exercise class model
Person, Employee, Student exercise 5	20	Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships
Lab Total	**50**	