ITP 115 – Programming in Python

Objects
part 2

Review

Object-Oriented Programming (OOP)

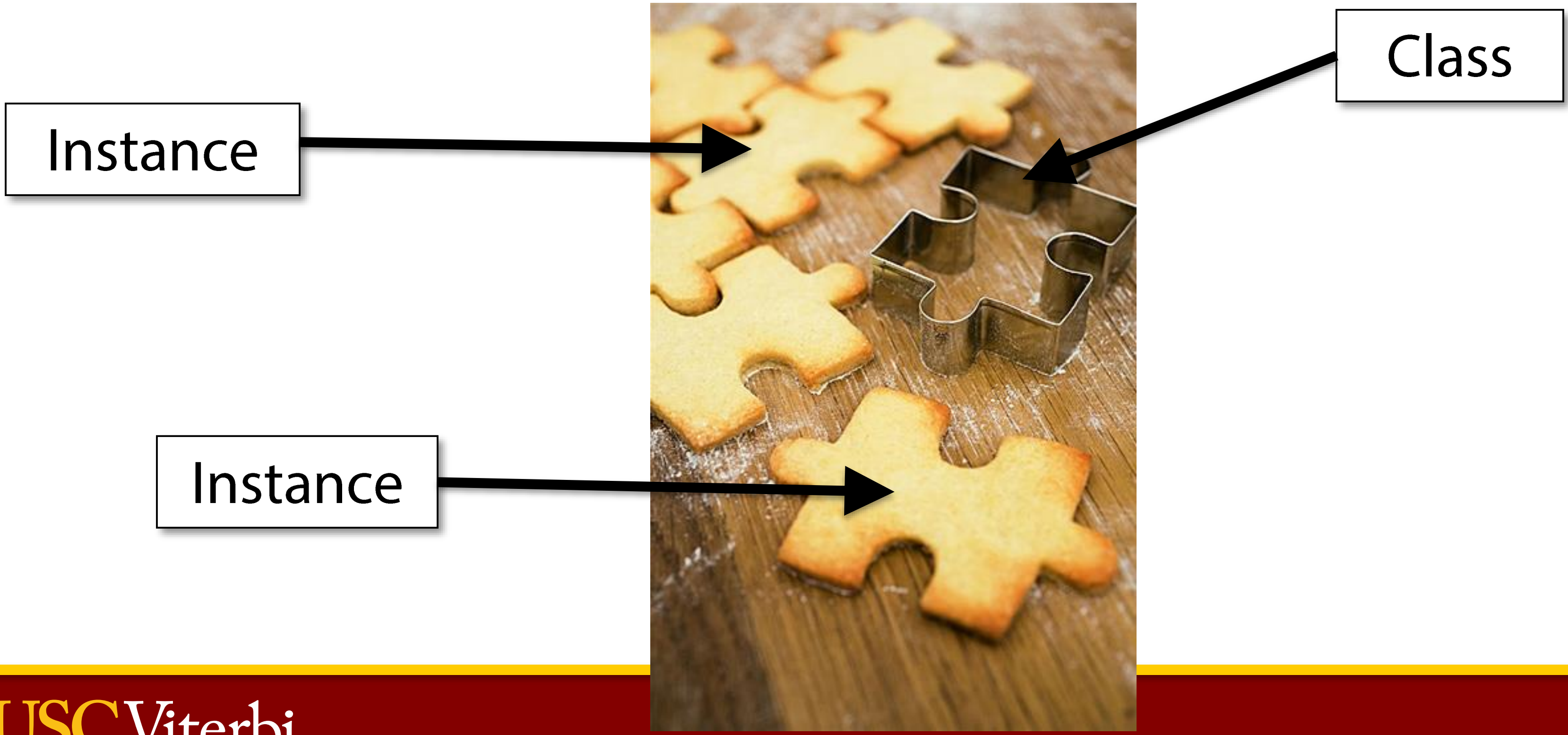
- A different way of thinking about programming
- A modern methodology used in the creation of the majority of new, commercial software
- The basic building block is the **software object**
  - just called an **object**

Classes and Objects

- Classes are like blueprints and defined by `class`
  - A class isn't an object, it's a design for one
- Objects are created (*instantiated*) from a class definition
- Classes contain
  - Attributes: set of object variables given to every object
  - Methods: functions that are part of each object

Classes and Instances

- Think of a **class** as a cookie cutter
- **Instances** (or **objects**) are the cookies



Instance

Class

Instance

Creating an Instance of a Class

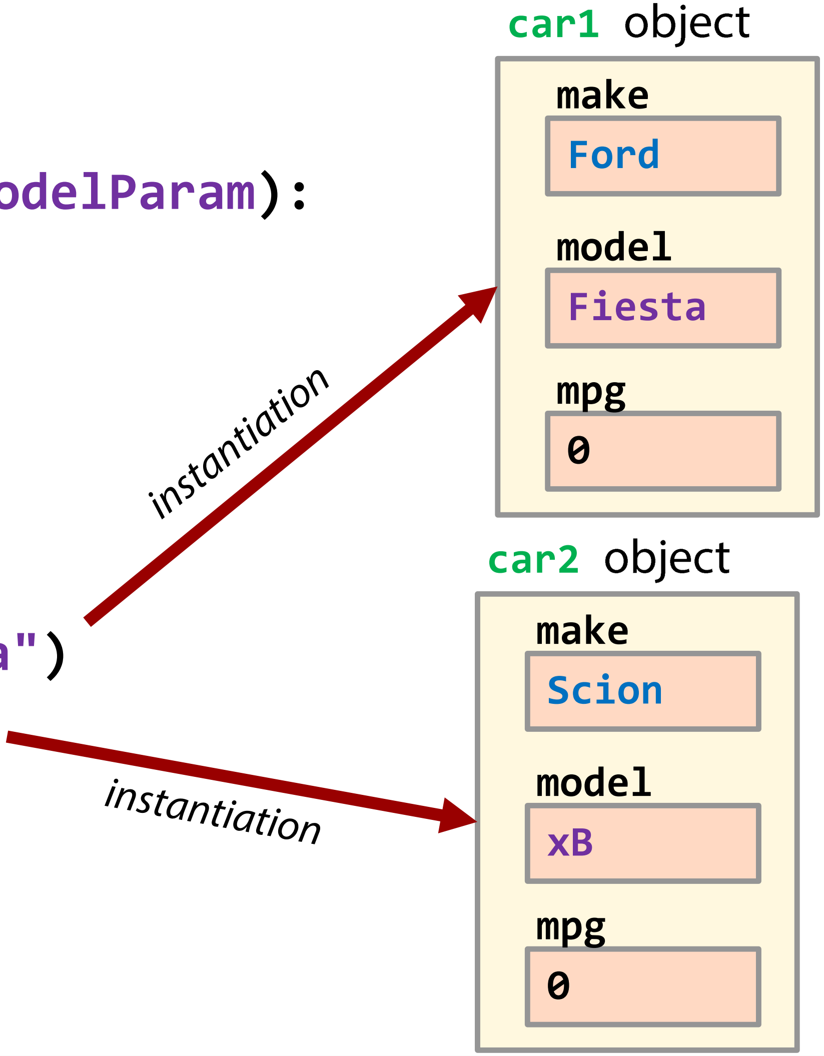
```
class Vehicle(object):
  ...

def main():
  v1 = Vehicle()

main()
```

`__init__(self):`

- A constructor is **method** that is used to create an instance of an object
- A constructor define what attributes will exist inside a object
- Constructors are called **automatically** when you create an object

Attributes and Constructors

```
class Vehicle(object):
  def __init__(self, makeParam, modelParam):
    self.make = makeParam
    self.model = modelParam
    self.mpg = 0

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car2 = Vehicle("Scion", "xB")
```



Methods

- Classes can have methods (or behaviors)

- Methods are part of the object just like attributes
  - Think: *functions associated with an object*

- Methods can access the attributes defined in the constructor using **`self`**

Method Input and Output

```
class Vehicle(object):
  def calcTripCost(self, miles):
    ... #perform some calculations
    return totalCost #new variable

def main():
  v1 = Vehicle()
  cost = v1.calcTripCost(100)
```

ENCAPSULATION AND ABSTRACTION

Consider Driving a Car

- We use brake pedal, accelerator pedal, steering wheel – know **what** they do

- We do not see mechanical details of **how** they do their jobs

- The complexity of how a car works has been **abstracted** away
  - **What** a car does (drive) is separate from **how** it works (engine, etc).

What is the point of all this?

- On a large software project, there might be dozens of programmers, hundreds of classes, and millions of lines of code

- OOP means organizing our code differently to solve these issues

Design Approach BEFORE OOP: Two Roles to Consider

- User
  - Interacts with the program (through keyboard, mouse, etc.)
  - Doesn't need to know anything about the code
- Programmer, class user (you)
  - Writes overall program logic, main()

Design Approach After OOP: Now There are Three!

- User
  - Interacts with the program (through keyboard, mouse, etc.)
  - Doesn't need to know anything about the code
- Programmer, class user (you)
  - Writes overall program logic, main()
  - **Uses classes**
- Programmer, class designer (you, or another programmer)
  - **Creates class definition** to be used by other programmers
  - Structures classes to be updated with little impact on users

Encapsulation

- Encapsulation means knowing **what** a class does without needing to know **how** it does it

- Ex: How does a dictionary actually work?

  crickets

  – To us, it isn't important

  – We just need to know a dictionary can do

Information Hiding

- Class design defines a method so it can be used without knowing details

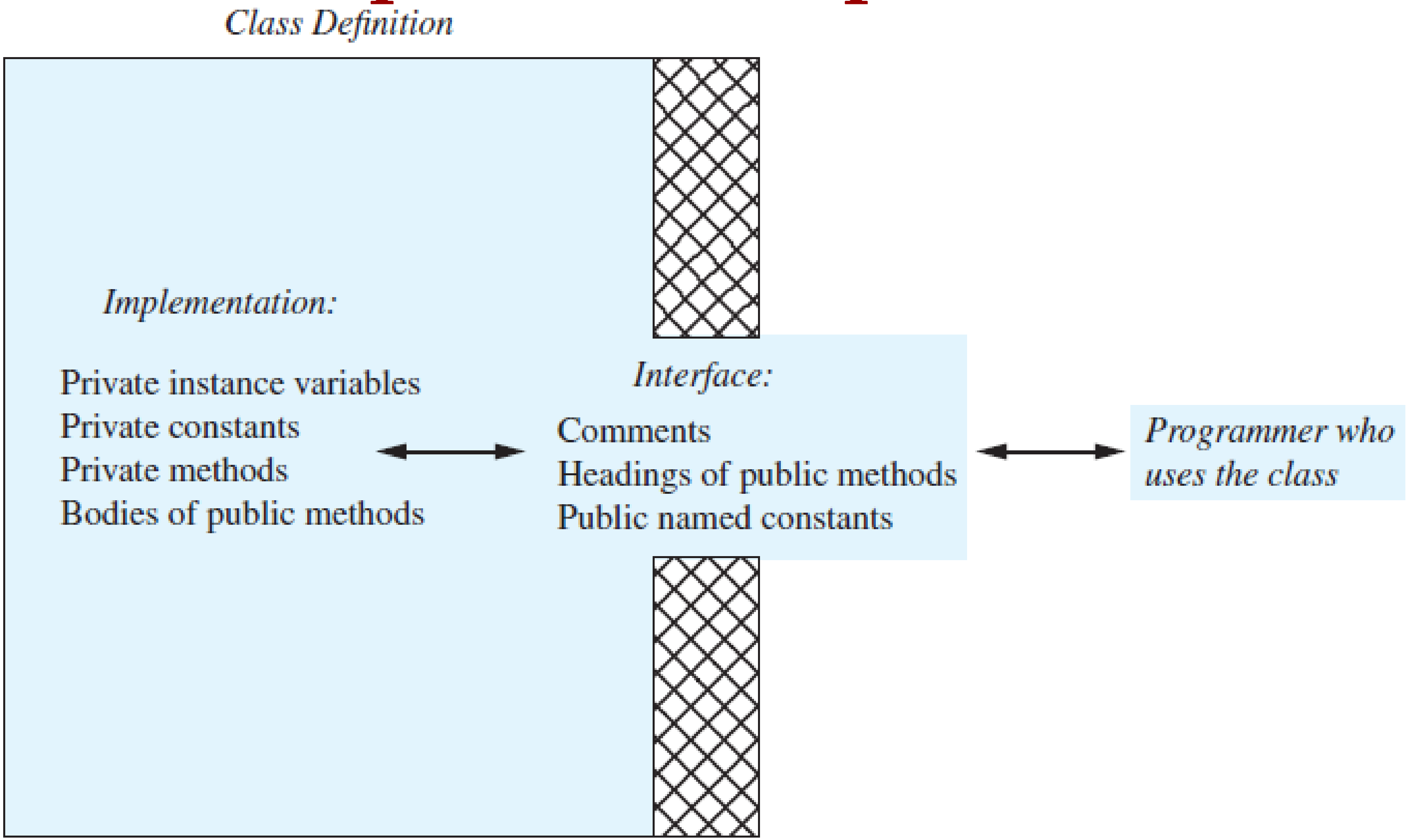
- Programmer using a class / method need <u>not</u> know details of implementation
  - Only needs to know *what* the method does

- Method design should separate ***what*** from ***how***

Encapsulation Separates Classes into Two Parts

- A class interface
  - Tells **what** the class does (not how)
  - Gives **headings** from public methods (the ones we can use) and comments about them
- A class implementation
  - Contains private attributes (the ones we can't see)
  - Includes **definitions** (details) of public and private methods

Encapsulation in pictures



Class Definition

Implementation:

Private instance variables
Private constants
Private methods
Bodies of public methods

Interface:

Comments
Headings of public methods
Public named constants

Programmer who uses the class

Advantages of Encapsulation

- Reduces errors
  - Prevents other programmers from directly changing attributes of objects
- Makes it easier to collaborate / work on large projects
  - Simplifies uses classes through public interface
- Code is easier to maintain / read

ENCAPSULATION IN PYTHON

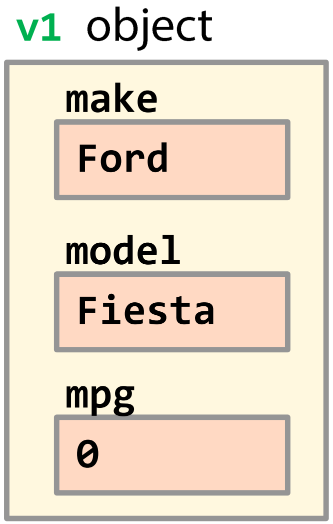
Public Attributes

- By default, all of an object's attributes and methods are **public**

- They can be directly accessed or invoked by a class user (e.g. in **main()**)

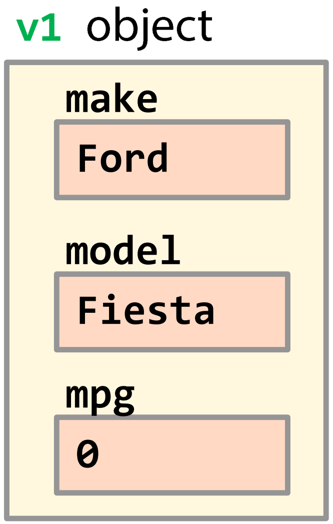
Example

```
class Vehicle(object):
   def __init__(self, make, model):
     self.make = make
     self.model = model
     self.mpg = 0

def main():
  v1 = Vehicle("Ford", "Fiesta")
```



Example

```
class Vehicle(object):
   def __init__(self, make, model):
     self.make = make
     self.model = model
     self.mpg = 0

def main():
   v1 = Vehicle("Ford", "Fiesta")
   print("The MPG is", v1.mpg)
```

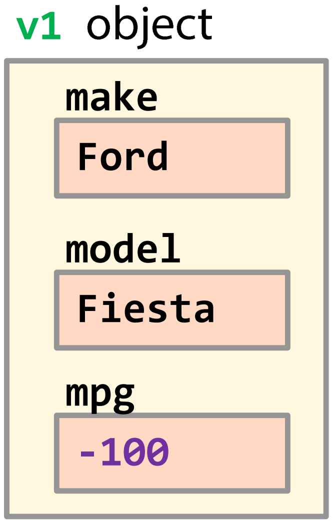


Output

```
The MPG is 0
```

Example

```
class Vehicle(object):
   def __init__(self, make, model):
     self.make = make
     self.model = model
     self.mpg = 0

def main():
  v1 = Vehicle("Ford", "Fiesta")
  print("The MPG is", v1.mpg)
  v1.mpg = -100
```



Should this be allowed?

Private Attributes

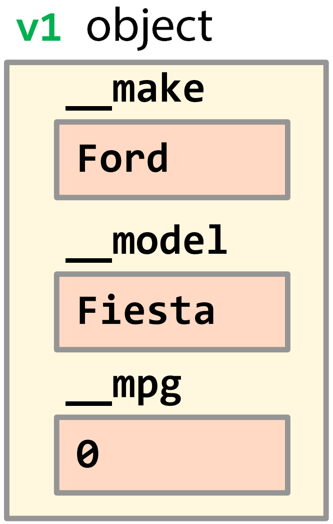
- To create a private attribute, begin the attribute name with **two underscores**

```
class Vehicle(object):
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
        self.__mpg = 0
```

- **Private attributes** can only be directly accessed from methods inside the class definition

Example

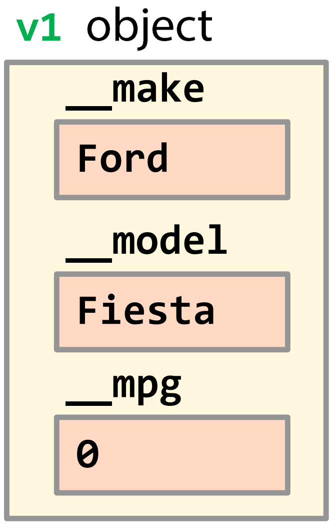
```
class Vehicle(object):
   def __init__(self, make, model):
     self.__make = make
     self.__model = model
     self.__mpg = 0

def main():
  v1 = Vehicle("Ford", "Fiesta")
```



Same as before

Example

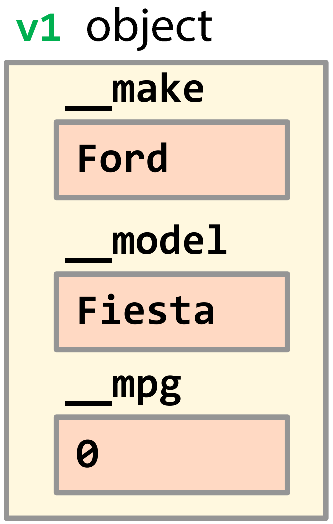
```
class Vehicle(object):
   def __init__(self, make, model):
     self.__make = make
     self.__model = model
     self.__mpg = 0

def main():
  v1 = Vehicle("Ford", "Fiesta")
  print(v1.__mpg)
```



Error! main() can't directly access **mpg**

Example

```
class Vehicle(object):
   def __init__(self, make, model):
     self.__make = make
     self.__model = model
     self.__mpg = 0

def main():
  v1 = Vehicle("Ford", "Fiesta")
  print(v1.__mpg)
  v1.__mpg = -100
```



Error! main() can't directly change **mpg**

Private Attributes

- Data is now `private`…
  - But we can't access it or change it at all
- We would like a way to **control** access and modification
- Allow indirect access to attributes
  - Also impose some sort of restrictions on that access (like error checking)

Using Get Methods

- One type of access method is a **get method**
  - Provides read access to a private attribute
  - Referred to as an **accessor** method

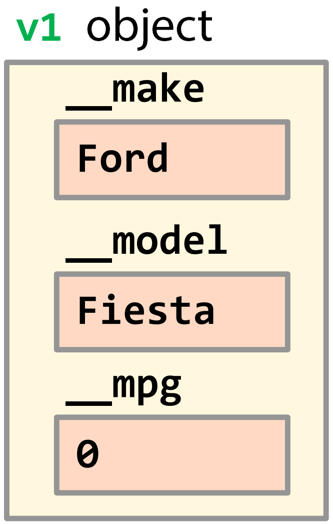
- Syntax

```
getAttribute(self)
```

  - Always **returns** the value of the attribute

Using Get Methods

```
class Vehicle(object):
   def __init__(self, make, model):
      self.__make = make
      self.__model = model
      self.__mpg = 0

  def getMPG(self):
      return self.__mpg
```



v1 object

__make

Ford

__model

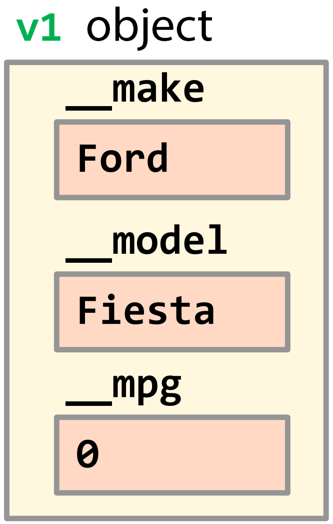
Fiesta

__mpg

0

Using Get Methods

```
class Vehicle(object):
   def __init__(self, make, model):
      self.__make = make
      self.__model = model
      self.__mpg = 0

   def getMPG(self):
      return self.__mpg

def main():
   v1 = Vehicle("Ford", "Fiesta")
   print("The MPG is", v1.getMPG())
```



Output

```
The MPG is 0
```

Using Set Methods

- Attributes can be changes with **set** method
  - Modifies the value of a private attribute
  - Referred to as a **mutator** method

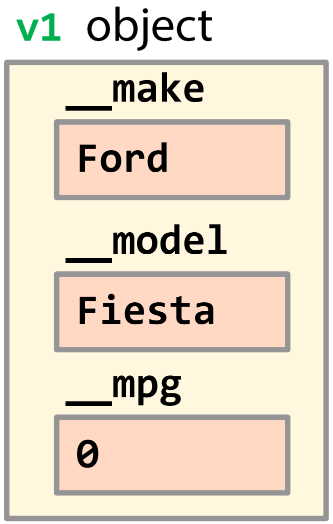
- Syntax

```
setAttribute(self, newAttribute)
```

  - Assigns the parameter value to the attribute
  - May perform error checking
  - Doesn't return anything

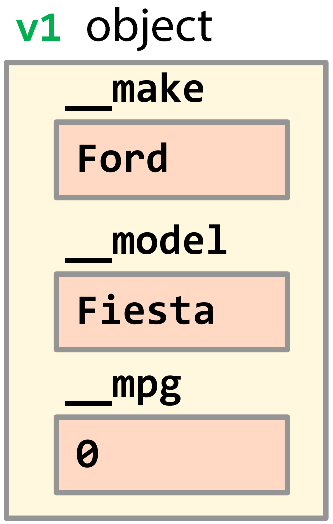
Using Set Methods

```
class Vehicle(object):
   def __init__(self, make, model):
      self.__make = make
      self.__model = model
      self.__mpg = 0

   def setMPG(self, newMPG):
      if newMPG >= 0:
         self.__mpg = newMPG
      else:
         print("Invalid MPG")

def main():
   v1 = Vehicle("Ford", "Fiesta")
```



Using Set Methods

```
class Vehicle(object):
   def __init__(self, make, model):
      self.__make = make
      self.__model = model
      self.__mpg = 0

   def setMPG(self, newMPG):
      if newMPG >= 0:
         self.__mpg = newMPG
      else:
         print("Invalid MPG")

def main():
   v1 = Vehicle("Ford", "Fiesta")
```



v1 object

__make

Ford

__model

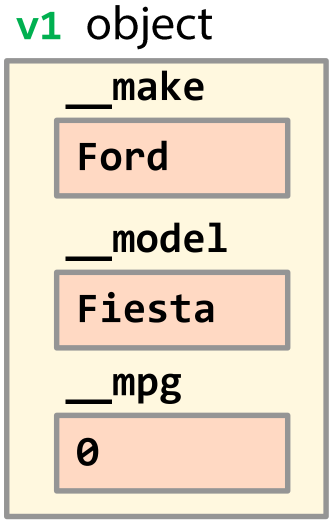
Fiesta

__mpg

0

Using Set Methods

```
class Vehicle(object):
    def __init__(self, make, model):
        self.__make = make
        self.__model = model
        self.__mpg = 0

    def setMPG(self, newMPG):
        if newMPG >= 0:
            self.__mpg = newMPG
        else:
            print("Invalid MPG")

def main():
    v1 = Vehicle("Ford", "Fiesta")
    v1.setMPG(-18)
```



v1 object

__make

Ford

__model

Fiesta

__mpg

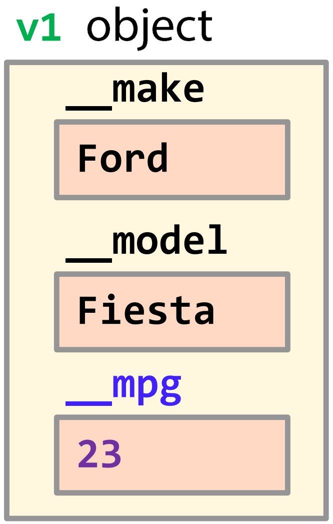
0

Output

```
Invalid MPG
```

Using Set Methods

```
class Vehicle(object):
   def __init__(self, make, model):
      self.__make = make
      self.__model = model
      self.__mpg = 0

   def setMPG(self, newMPG):
      if newMPG >= 0:
         self.__mpg = newMPG
      else:
         print("Invalid MPG")

def main():
   v1 = Vehicle("Ford", "Fiesta")
   v1.setMPG(-18)
   v1.setMPG(23)
```



v1 object

__make

Ford

__model

Fiesta

__mpg

23

Rough Guidelines for Implementing Privacy in a Class*

- `public:`
  - **get** and **set** methods for each instance variable
  - methods the user needs to use in your class

- `private:`
  - attributes / instance variables
  - any methods that the user shouldn't access *(all methods in our course will be public*)

* On a project there can be good reasons to follow these guidelines

For reference only

Private Methods

- To create a private method, add two leading underscores to its name

```
class Vehicle(object):
   def __init__(self, make, model):
      print("A new vehicle is born!")
      self.make = make            # public attribute
      self.__model = model        # private attribute

   def __privateMethod(self):
      print("This is inside a private method." )

   def publicMethod(self):
      self.__privateMethod()      # OK to call private method
```

For reference only

Accessing Private Methods

- If you try to access the private method outside of Vehicle

```
v1 = Vehicle("Toyota", "Corolla")
v1.publicMethod()
                    This is inside a private method
v1.privateMethod()
                                             Error!
v1.__privateMethod()
                                             Error!
```

Using Decorators

- Decorators allow you to harness the power of access methods while hiding the implementation from the client

- They essentially wraps access methods around the consistent and familiar dot notation

- These are optional methods to use *instead* of **get** and **set methods**

For reference only

Using Decorators – *Get* Methods

```
class Foo(object):
  def __init__(self):
    self.__privateX = 40

  @property
  def x(self):
    return self.__privateX
```

```
def main():
  f = Foo()
  print(f.x)
```

For reference only

Using Decorators – *Set* Methods

```
class Foo(object):
  def __init__(self):
    self.__privateX = 40

  @x.setter
  def x(self, value):
    self.__privateX = value
```

```
def main():
  f = Foo()
  f.x = 30
```

For reference only

Decorator Example

```
class Foo(object):
    def __init__(self):
        self._x = 40

    @property
    def x(self):
        print("get")
        return self._x

    @x.setter
    def x(self, value):
        print("set")
        self._x = value

def main():
    f = Foo()
    print(f.x)
    f.x = 50
    print(f.x)
```

- End lecture

SEPARATING CLASSES INTO MULTIPLE FILES

Using Separate Files

- Common practice with object programming
- Use separate files for each class
- Use one (or multiple) files to “drive” your program (this file has **main** method)

Using Separate Files

- Class file – `Vehicle.py`
  - Define class, methods, variables as before

```
class Vehicle(object):
    def __init__(self):
        ...
```

Using Separate Files

- “Driver” file – `Program.py`
  - This file contains the `main()` function
  - `main()` contains the logic that runs the entire program
  - In `main()` you will create `Vehicle` objects
  - To create `Vehicle` objects, you need to tell Python what / where `Vehicle` is defined

Using Separate Files

- “Driver” file – **Program.py**
  - General Syntax

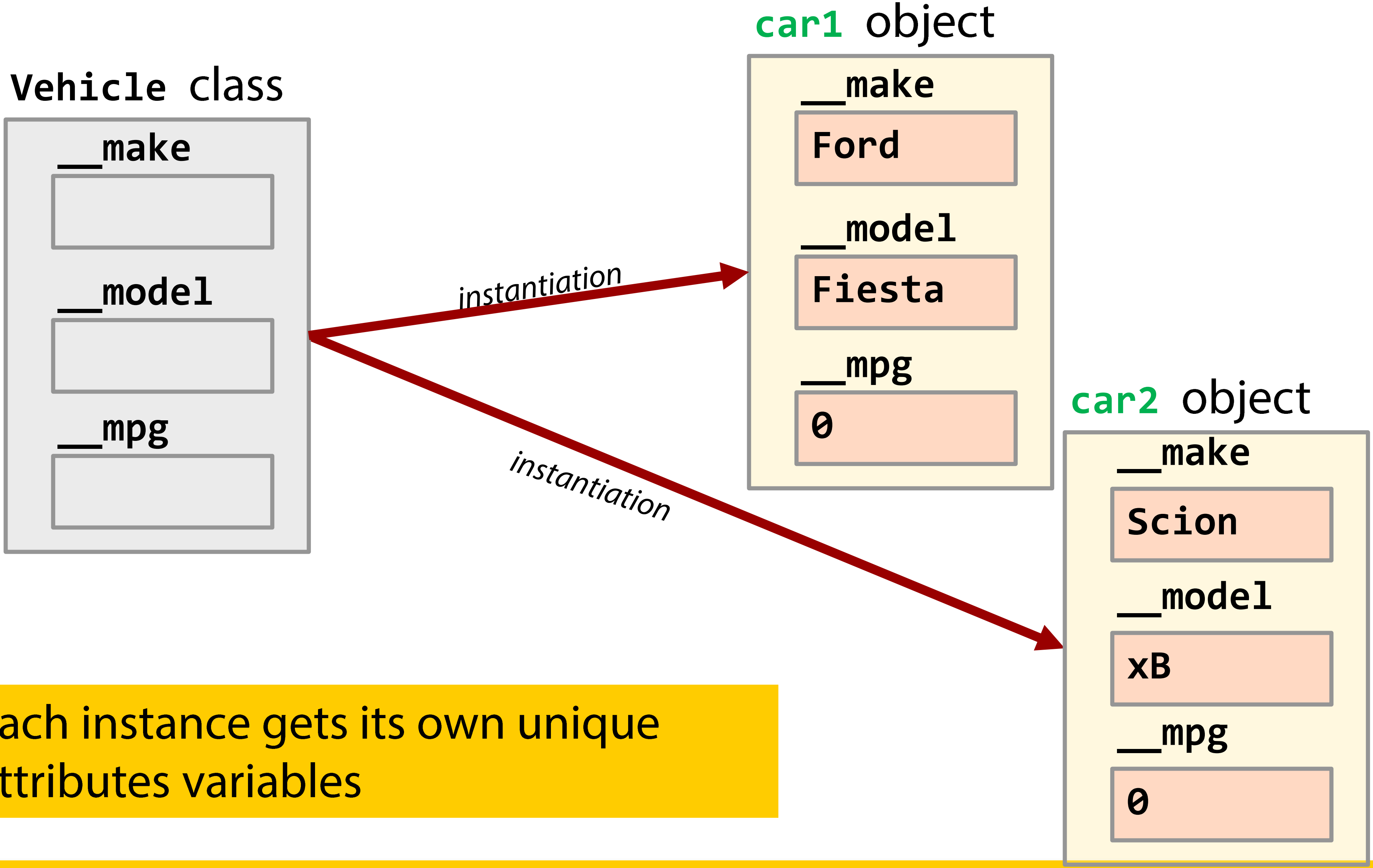
```
from fileName import className
```

```
from Vehicle import Vehicle
```

Instance Attributes

```
class Vehicle(object):
  def __init__(self, make, model):
    self.__make = make
    self.__model = model
    self.__mpg = 0

def main():
    car1 = Vehicle("Ford", "Fiesta")
    car2 = Vehicle("Scion", "xB")
```

Instance Attributes



Each instance gets its own unique attributes variables

Shared Attributes

- What is we want similar object to be able to share some data?

- Example
  - Constants used by all objects of a class
  - Count of number of objects created

Class Variables

- Attributes are shared by <u>all instances</u> of a class
- Can be accessed by all objects of that class type
- Only 1 version of a class variable exists
  - Even if many objects exist
- These are sometimes called ***static variables***

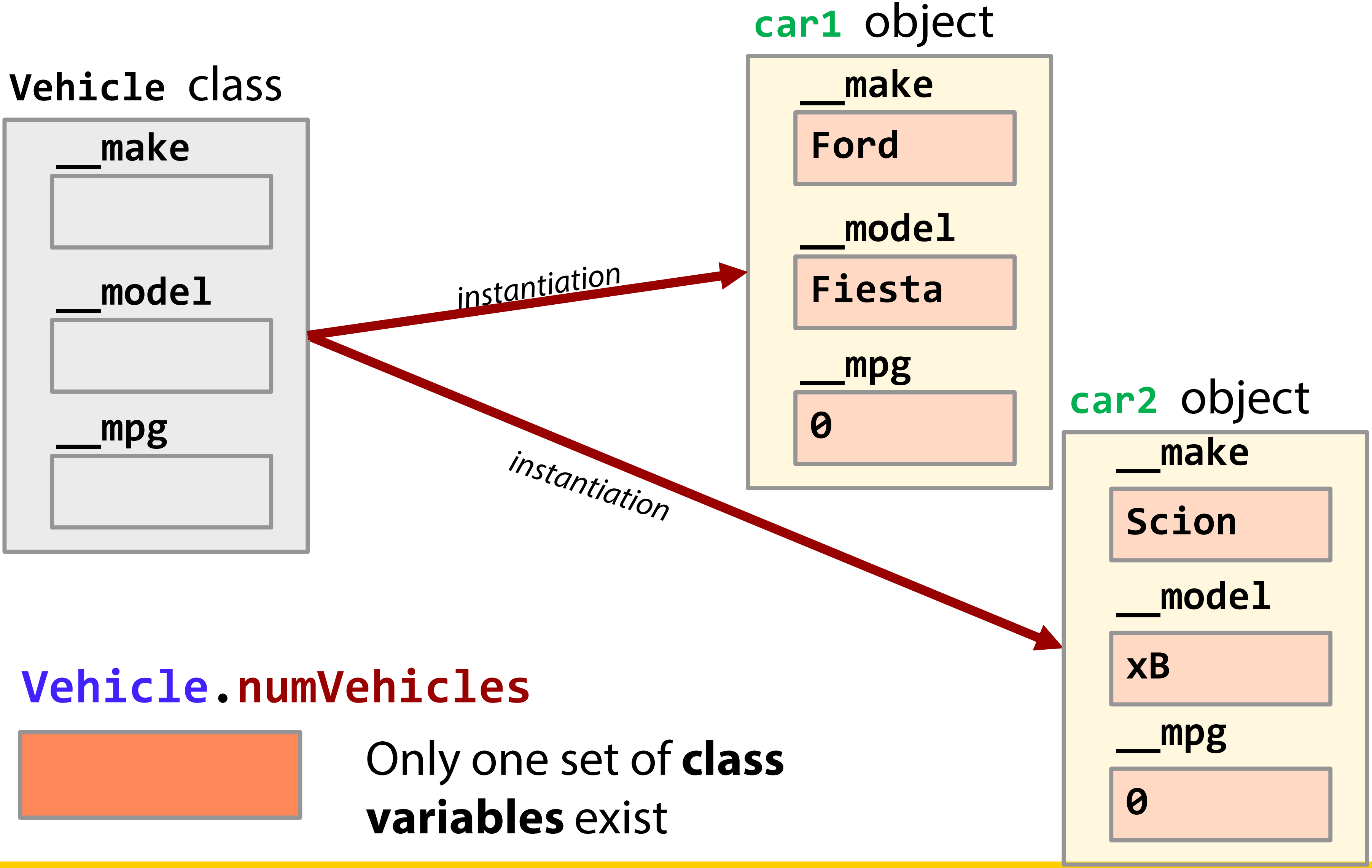
Class Variables

```
class Vehicle(object):
  numVehicles = 0
  def __init__(self, make, model):
    self.__make = make
    self.__model = model
    self.__mpg = 0
    Vehicle.numVehicles += 1

def main():
  v1 = Vehicle("Ford", "Fiesta")
  v2 = Vehicle("Scion", "xB")
```

class variables are declared *outside* of **`__init__`**

class variables are accessed by `ClassName.variable`

Instance Attributes



Only one set of **class variables** exist

Class Variables

```
def main():
  print("Total num is", Vehicle.numVehicles)
  v1 = Vehicle("Ford", "Fiesta")
  print("Total num is", Vehicle.numVehicles)
  v2 = Vehicle("Scion", "xB")
  print("Total num is", Vehicle.numVehicles)
```

class variables can be accessed before objects have been created

Output

```
Total vehicles is 0
Total vehicles is 1
Total vehicles is 2
```

Summary: 3 Types of Variables

- Local variables
- Instance variables
- Class variables

Local Variables

```
def main():
    msg = "hello world"
```

- Declared in a function (or method)
- These variable exist **only** during the function's execution
- Use them for temporary operations
- Remember **scope**

Instance (or Object) Variables

```
class Vehicle(object)
   def __init__(self, make, model):
      self.__make = make
```

- Declared in a class
- Exist as long as the object exists
- Every object of the class has a **unique set** of variables
- Can be **public** or **private**

Class (or Static) variables

```
class Vehicle(object):
    numVehicles = 0
```

- Declared in a class
- Exist as long as the program is running
- Every object of the class **shares only one** copy of the variable

For reference only

Static methods

- Static methods are declared in a class…
- But are invoked without using a specific object
- Instead use the class name

```
    Vehicle.showCount()
```

For reference only

Static Methods

```
class Vehicle(object):
  numVehicles = 0

  def __init__(self, make, model):
    self.__make = make
    self.__model = model
    Vehicle.total += 1

  @staticmethod
  def status():
    print("Total number of Vehicles ", Vehicle.numVehicles)

def main():
  v1 = Car("Ford", "Fiesta")
  Vehicle.status()
```

For reference only

Summary

- Attributes
  - Instance variables
    - Each instance of the class has its own values for the attributes
  - Class (or *static*) variables
    - If a class is like a blueprint, then a class attribute is like a Post-it note stuck to the blueprint
- Methods
  - Instance methods
    - Special ones – constructor (`__init__`) and print (`__str__`)
  - Static methods *(reference only)*
    - Use `@staticmethod` decorator