Inheritance

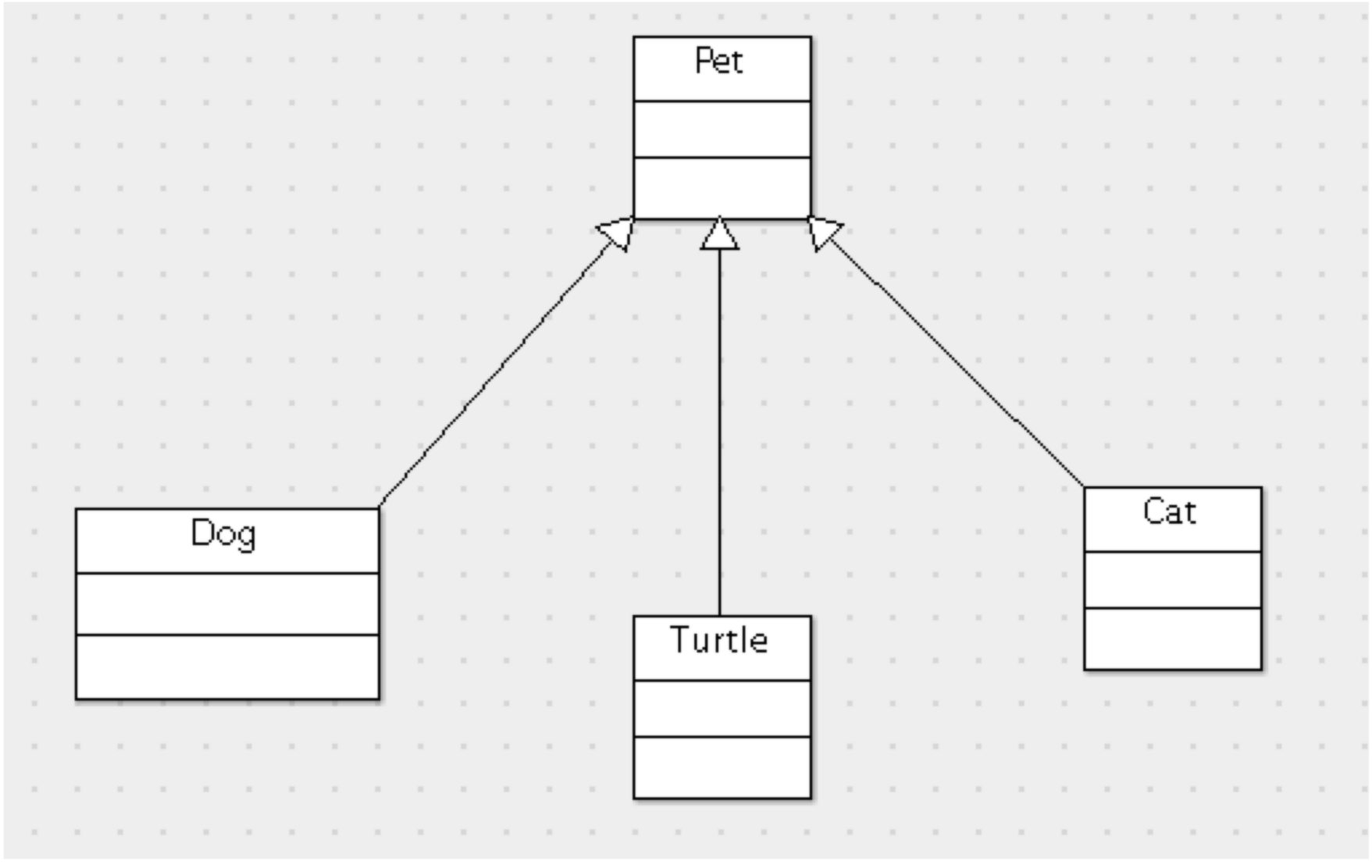
Pet Store

```
class Dog:
    def __init__(self, name):
        self.name = name
    def describe(self):
        return "the dog says: Woof!"
class Bird:
    def __init__(self, name):
        self.name = name
    def describe(self):
        return "the bird says: Chirp!"
class Cat:
    def __init__(self, name):
        self.name = name
    def describe(self):
        return "the cat says: Meow!"
```

```
>>> fido = Dog("Fido")
>>> fido.describe()
'the dog says: Woof!'
```

Repetitive! How can we do better?

Common Base



Inheritance

`Dog`, `Cat`, and `Bird` are all specific versions of a more general concept: a **pet**.

When you have several classes related in this way, you can create a **base class** called `Pet`.

This base class holds common code that `Cat`, `Dog`, `Bird`, etc. all build on, and use.

This is called **inheritance**.

Inheriting

Use parentheses on the `class` line.

```
class Pet:
    def __init__(self, name):
        self.name = name

class Dog(Pet):
    def describe(self):
        return "the dog says: Woof!"

class Cat(Pet):
    def describe(self):
        return "the cat says: Meow!"
```

```
>>> fido = Dog("Fido")
>>> fido.describe()
'the dog says: Woof!'
>>> fluffy = Cat("Fluffy")
>>> fluffy.describe()
'the cat says: Meow!'
```

Terminology

We say that `Dog` and `Cat` **subclass** `Pet`. And, `Dog` and `Cat` are **subclasses** of `Pet`.

Conversely, `Pet` is the **superclass** of both `Dog` and `Cat`.

Pet is also the **base class** because it doesn't inherit from anything, other than `object`.

Inheritance Chain

```
class Pet:
    def __init__(self, name):
        self.name = name

class Dog(Pet):
    def describe(self):
        return "the dog says: Woof!"

class LapDog(Dog):
    def describe(self):
        return "the lap dog says: Yip!"

class LoudLapDog(LapDog):
    def describe(self):
        return "the loud lap dog says: YIP!"
```

```
>>> buck = Dog("Buck")
>>> buck.describe()
'the dog says: Woof!'

>>> shorty = LapDog("Shorty")
>>> shorty.describe()
'the lap dog says: Yip!'

>>> pip = LoudLapDog("Pip")
>>> pip.describe()
'the loud lap dog says: YIP!'
```

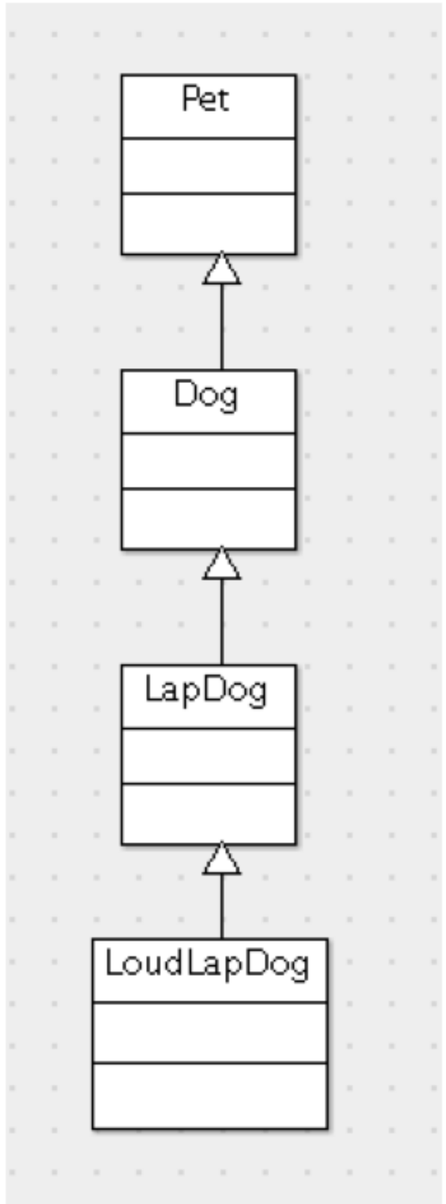
Terminology

```
class Pet:
    def __init__(self, name):
        self.name = name

class Dog(Pet):
    def describe(self):
        return "the dog says: Woof!"

class LapDog(Dog):
    def describe(self):
        return "the lap dog says: Yip!"

class LoudLapDog(LapDog):
    def describe(self):
        return "the loud lap dog says: YIP!"
```

- `LoudLapDog` is a subclass of `LapDog`. `LapDog` subclasses `Dog`. `Dog` subclasses `Pet`.
- `Pet` is the superclass of `Dog`, which is the superclass of `LapDog`, which superclasses `LoudLapDog`.
- There is only one base class: `Pet`.

Visualizing

```
class Pet:
    def __init__(self, name):
        self.name = name

class Dog(Pet):
    def describe(self):
        return "the dog says: Woof!"

class LapDog(Dog):
    def describe(self):
        return "the lap dog says: Yip!"

class LoudLapDog(LapDog):
    def describe(self):
        return "the loud lap dog says: YIP!"
```



General to Specific

`isinstance()` is a built-in function. It tells you whether an object is an instance of a class or not.

An object is an instance of its class, AND also an instance of *all* its superclasses.

```
>>> biff = LapDog("Biff")
>>> isinstance(biff, LapDog)
True
>>> isinstance(biff, LoudLapDog)
False
>>> isinstance(biff, Dog)
True
>>> isinstance(biff, Pet)
True
```

This spectrum of generic to specific turns out to be *very* useful.

Practice

Create a file named "pets.py". Type in the following:

```
class Pet:
    def __init__(self, name):
        self.name = name
class Dog(Pet):
    def describe(self):
        return "the dog says: Woof!"
class Cat(Pet):
    def describe(self):
        return "the cat says: Meow!"

fred = Dog("Fred")
misha = Cat("Misha")
print(fred.name + "  " + fred.describe())
print(misha.name + "  " + misha.describe())
```

Run as a Python program. This ought to be the output:

```
Fred the dog says: Woof!
Misha the cat says: Meow!
```

EXTRA CREDIT: Can you move `describe()` into the `Pet` class?

"Is-A" Relationships.

Inheritance models "X is a Y" relationships. "Dog" is a "Pet", "LapDog" is a "Dog", etc.

It's possible to build an inheritance chain that violates this. But it tends to create problems, and you'll usually end up regretting it.

Defining Member Vars, Again

Let's look at this choice with the `Quarter` class again.

```
class Quarter:
    value = 25

class Quarter:
    def __init__(self):
        self.value = 25
```

For `Quarter`, it doesn't really matter. But in general, these two choices give you different benefits and options.

Overriding Values

Subclasses can override the superclass' value. This sometimes lets you define one method, in the base, instead of redefining it in every subclass.

```
class Pet:
    sound = ""
    def __init__(self, name):
        self.name = name
    def describe(self):
        return "the pet says: {}!".format(
               self.sound)

class Dog(Pet):
    sound = "Woof"
class Cat(Pet):
    sound = "Meow"
class Bird(Pet):
    sound = "Chirp"
```

```
>>> rover = Dog("Rover")
>>> rover.describe()
'the pet says: Woof!'

>>> misty = Cat("Misty")
>>> misty.describe()
'the pet says: Meow!'

>>> angel = Bird("Angel")
>>> angel.describe()
'the pet says: Chirp!'
```

The __class__ attribute

Python instances all have an attribute called `__class__`.

This is the *same class object* they were instantiated from.

```
>>> class Penny:
...     value = 1

>>> coin = Penny()
>>> coin.__class__
<class '__main__.Penny'>

>>> # Can even instantiate from it!
... new_coin = coin.__class__()
>>> type(new_coin)
<class '__main__.Penny'>
```

__class__.__name__

In Python, all class objects have a `__name__` attribute - a string:

```
>>> Penny.__name__
'Penny'
>>> coin.__class__.__name__
'Penny'
```

Even the built-in types have this! They're classes too.

```
>>> int.__name__
'int'
>>> dict.__name__
'dict'

>>> x = 4.5
>>> y = ["a", "b", "c"]
>>> x.__class__.__name__
'float'
>>> y.__class__.__name__
'list'
```

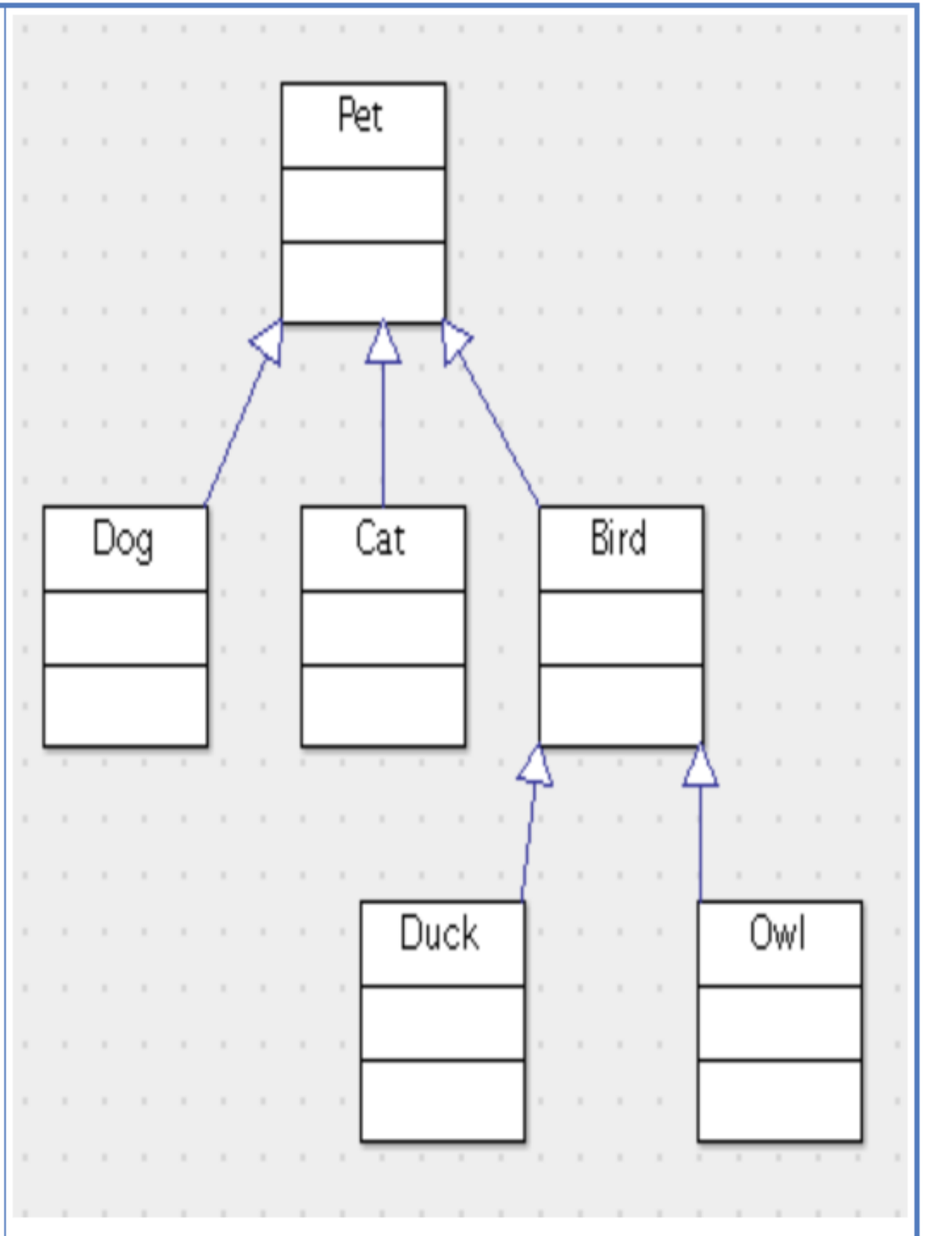
Pet.describe()

```
class Pet:
    sound = ""
    def __init__(self, name):
        self.name = name
    def describe(self):
        kind_of_pet = self.__class__.__name__.lower()
        return "the {} says: {}!".format(kind_of_pet, self.sound)
class Dog(Pet):
    sound = "Woof"
# Etc. for Cat, Bird, Giraffe...
```

```
>>> rover = Dog("Rover")
>>> rover.describe()
'the dog says: Woof!'
>>> misty = Cat("Misty")
>>> misty.describe()
'the cat says: Meow!'
>>> angel = Bird("Angel")
>>> angel.describe()
'the bird says: Chirp!'
```

Sub-sub-types

```
class Bird(Pet):
    def describe(self):
        return "the bird says: Chirp!"

class Duck(Bird):
    def describe(self):
        return "the duck says: Quack!"

class Owl(Bird):
    def describe(self):
        return "the owl says: Hoooo!"
```



Lab: Simple Inheritance

Lab file: `inheritance.py`

- In `labs` folder
- When you are done, give a thumbs up...
- ... and then do `inheritance_extra.py`

Stocks Again

Let's revisit the StockModel and StockView classes:

```
>>> model = StockModel('AAPL', 176.18, 177.09, 154718, 2505047)
>>> view = StockView()
>>> view.render(model)
'AAPL: $177.09 (Bearish)'
```

Let's subclass to create views for other output formats.

JSON View

We may want to serve this data through an API endpoint.

Let's make a view that will render a JSON response body:

```
import json
class StockJSONView(StockView):
    def render(self, model):
        params = self.params(model)
        return json.dumps(params)
```

```
>>> model = StockModel('AAPL', 159.29, 163.05, 44035531, 22509937)
>>> view = StockJSONView()
>>> view.render(model)
'{"name": "AAPL", "price": 163.05, "sentiment": "Bullish"}'
```

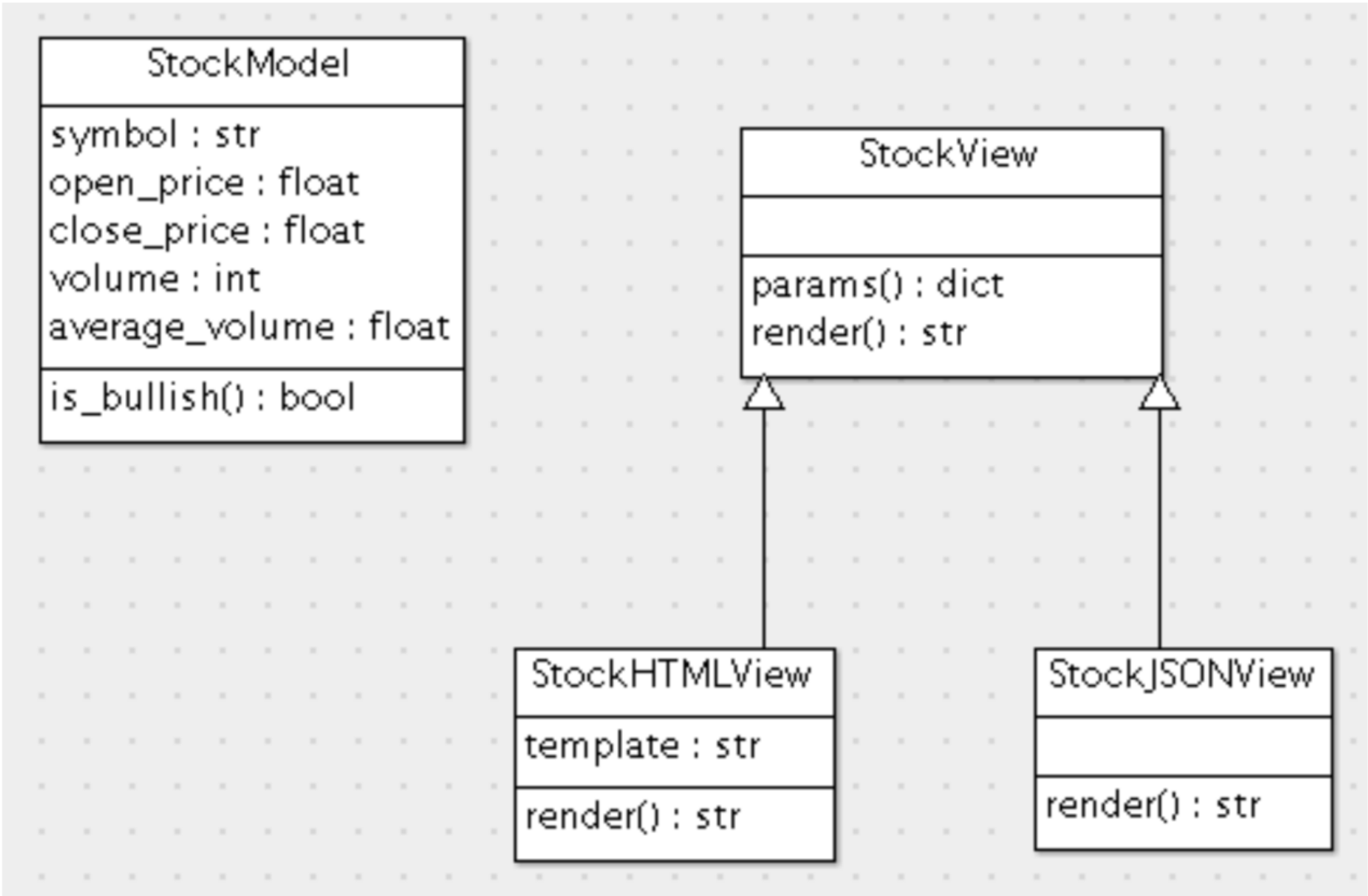
HTML View

```
STOCK_HTML_TEMPLATE = '''
<html>
  <title>Stock Report for {name}</title>
  <body>
     <dl><dt>Name:</dt><dd>{name}</dd>
          <dt>Closing price:</dt><dd>{price}</dd>
          <dt>Assessment:</dt><dd>{sentiment}</dd>
     </dl></body></html>
'''.strip()

class StockHTMLView(StockView):
    def __init__(self, template):
        self.template = template

    def render(self, model):
        params = self.params(model)
        return self.template.format_map(params)
```

HTML View

```
>>> model = StockModel('FB', 172.06, 183.37, 76_670_183, 25219450)
>>> view = StockHTMLView(STOCK_HTML_TEMPLATE)
>>> print(view.render(model))
<html>
  <title>Stock Report for FB</title>
  <body>
     <dl><dt>Name:</dt><dd>FB</dd>
         <dt>Closing price:</dt><dd>183.37</dd>
     <dt>Assessment:</dt><dd>Bullish<dd></dd>
</dl></body></html>
```

Class Diagram



Polymorphism

The `render()` method produces different output, depending on whether you invoke it on an instance of `StockView`, `StockHTMLView` or `StockJSONView`.

But the signature and kind of result is the same.

This is called **polymorphism**. It means the same operation (the `render()` method) is available in each, customized to the type.