Object Oriented Programming, Namespaces and Classes

- OOP Introduction
- Encapsulation
- Abstraction
- Inheritance
- Classes in Python
- Modules as Namespaces
- Classes as Namespaces

Acknowledgement

- Some of the slides in this presentation are taken from online available slides of:
  - Damian Gordon
    - https://www.slideshare.net/DamianGordon1/objectorientated-design?from_action=save
  - UMBC Python Course Lecture Notes
    - https://www.csee.umbc.edu/courses/691p/

Object Oriented Design

- Objects in software engineering are not typically *tangible somethings* that you can pick up, sense, or feel, but they are models of *somethings* that can do certain things and have certain things done to them
- Object-Oriented? Oriented simply means directed toward. So object-oriented simply means, "functionally directed toward modeling objects".
- It is one of many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior.

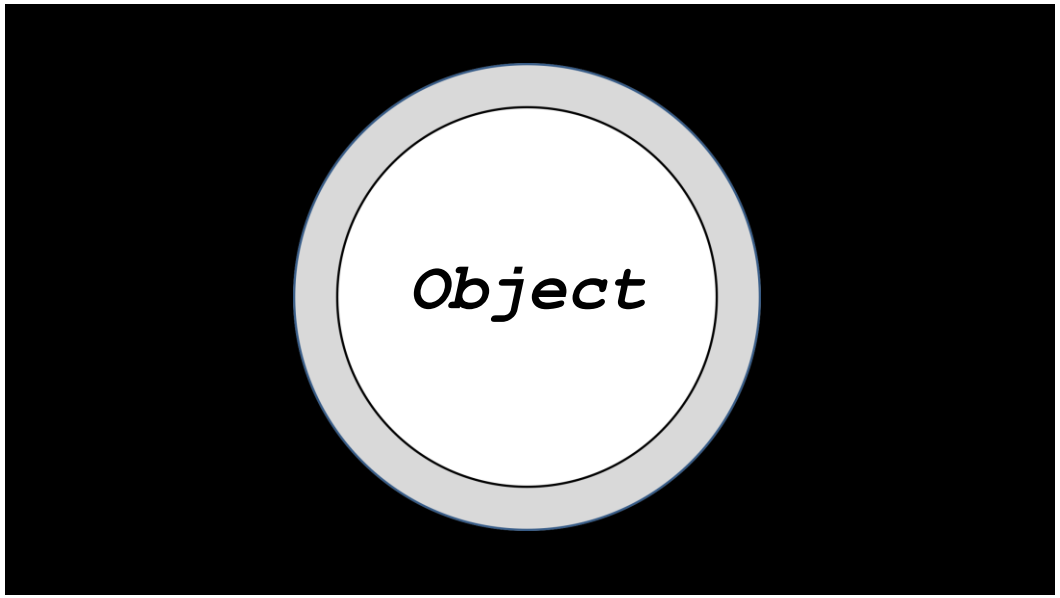
Object Oriented Jargons

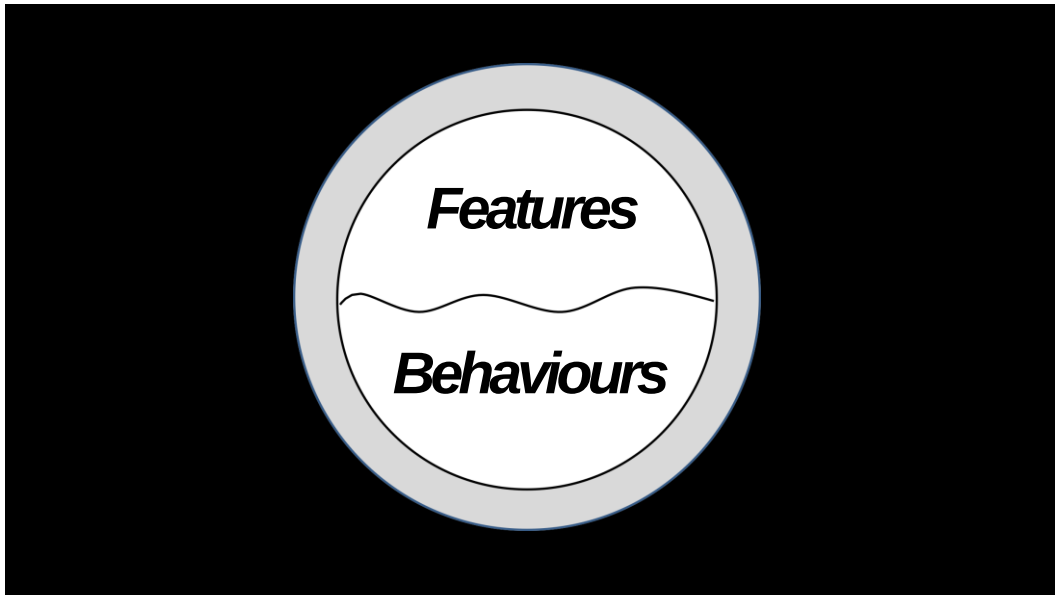
- **Object-oriented Analysis (OOA):** is the process of looking at a problem, system, or task that somebody wants to turn into an application and identifying the objects and interactions between those objects.
- **Object-oriented Design** (**OOD**): is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify what objects can activate specific behaviors on other objects.
- **Object-oriented Programming (OOP):** is the process of converting this perfectly defined design into a working program that does exactly what the CEO originally requested.

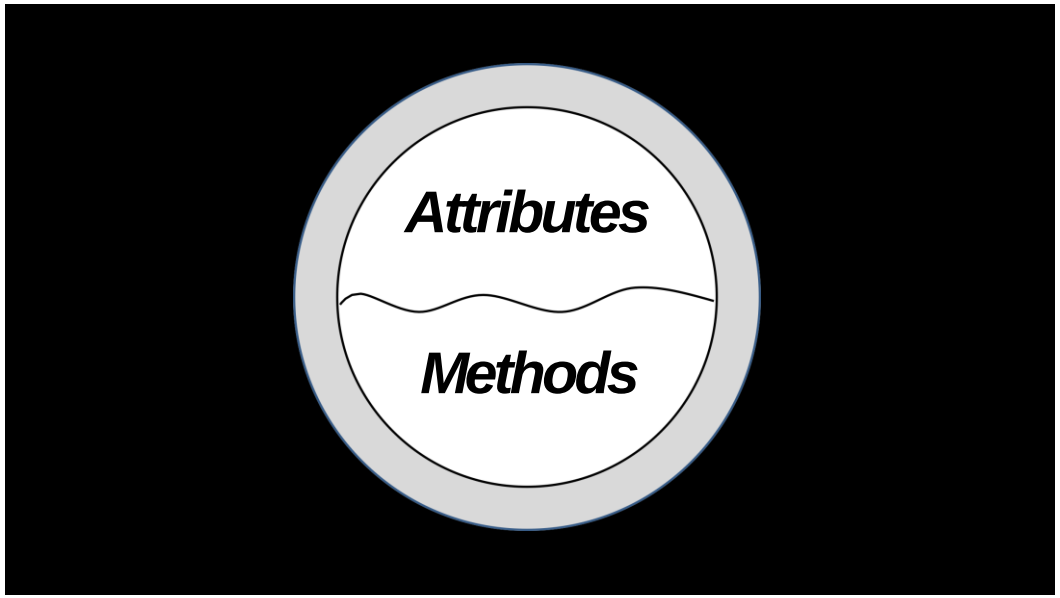
Object Orientation

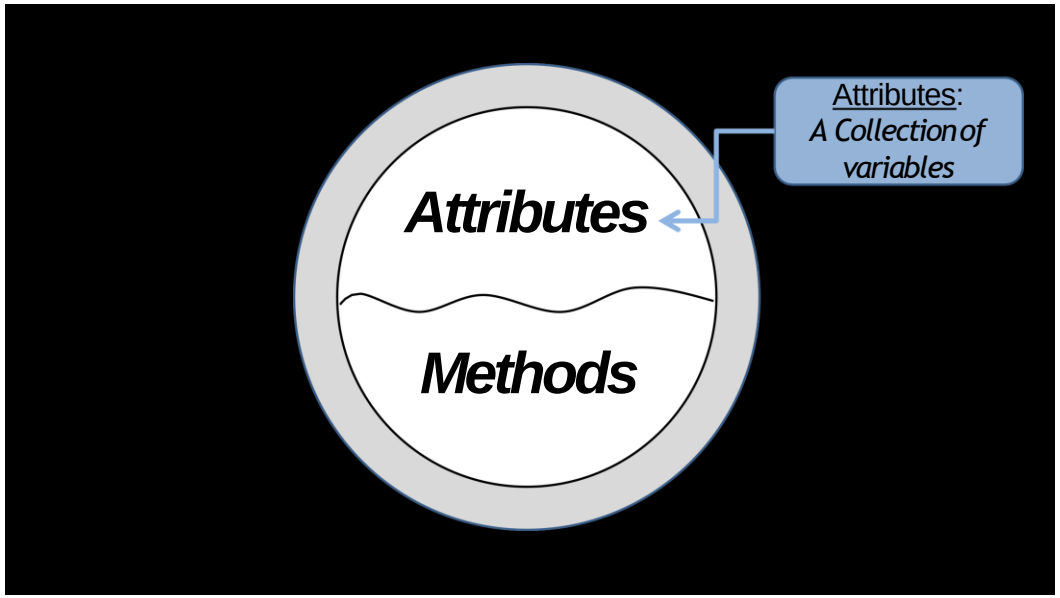
Object Orientation

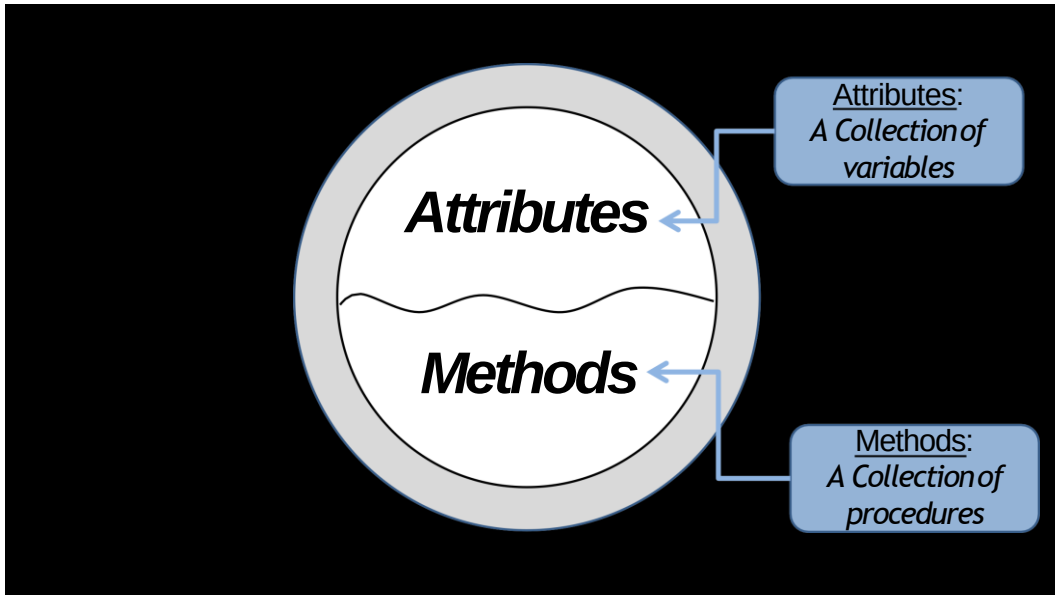
- Software objects are models are similar in the sense that they have certain features and can do certain things.

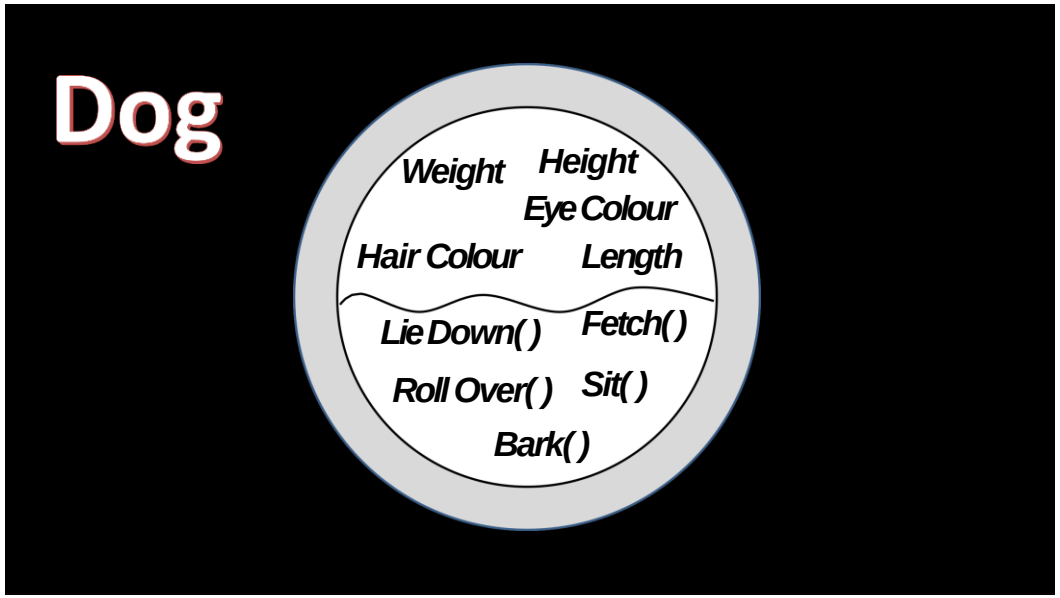

Object

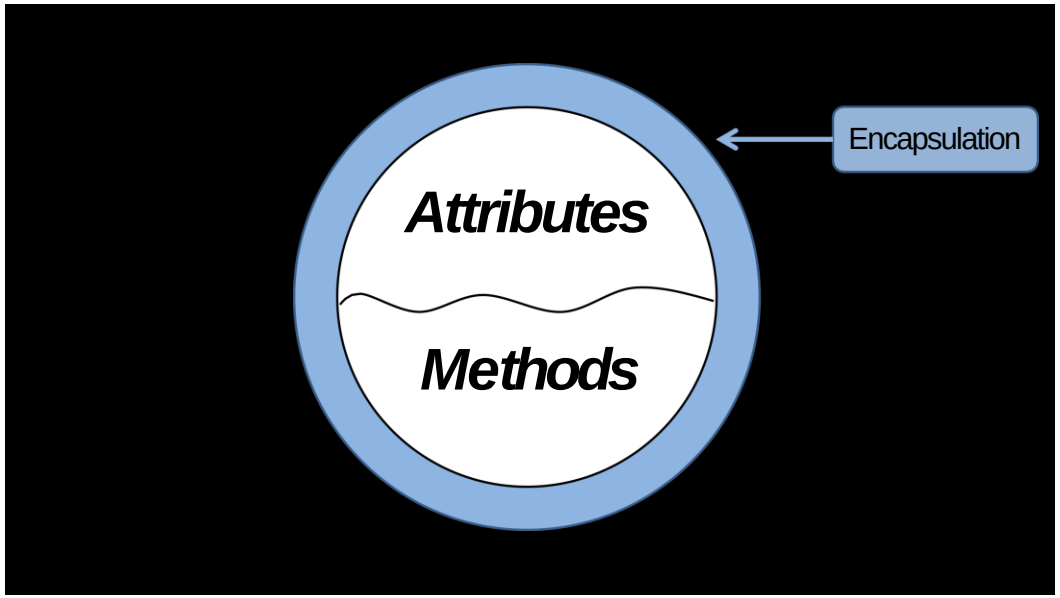

Features
Behaviours


Attributes
Methods


Attributes
Methods
Attributes:
A Collection of
variables

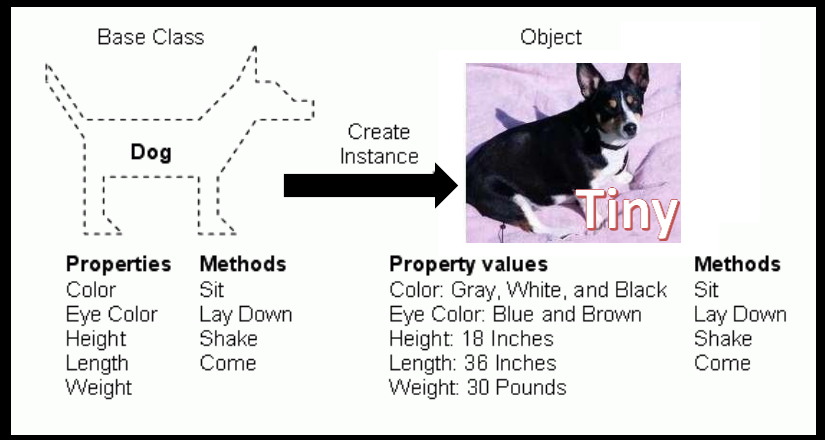

Attributes
Methods
Attributes:
A Collection of variables
Methods:
A Collection of procedures


Dog
Weight
Height
Eye Colour
Hair Colour
Length
Lie Down( )
Fetch( )
Roll Over( )
Sit( )
Bark( )



Object Orientation

- To create an object what we typically do is first create the general class that a specific object comes from, and then we create a specific object of that class.
- For example, if a want to model a specific dog, first create the general CLASS of DOG, and then create a specific instance of that class, an OBJECT, called TINY the dog.



Object Orientation

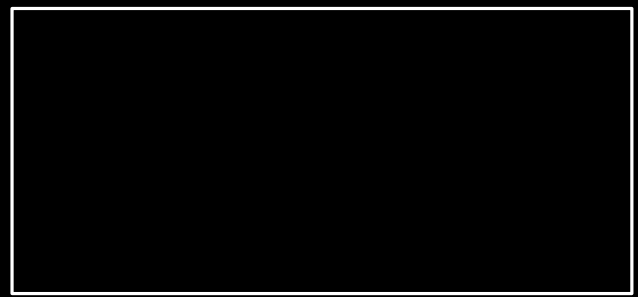
- Sometimes it's easier to model things as an OBJECT, but often it's easier to model a collection of things as a CLASS, and then create instances ('OBJECTS') of that CLASS.
- We can use UML (the Unified Modelling Language) to model Object-Orientated programs, and one modelling tool within UML is called a CLASS DIAGRAM.

Object Orientation

- Let's imagine we wanted to create a CLASSDIAGRAM for the following scenario:
  - We work in a greengrocer, we sell APPLES and ORANGES; APPLES are stored in BARRELS and ORANGES are stored in BASKETS.

Object Orientation

- APPLES are stored in BARRELS and ORANGES are stored in BASKETS.

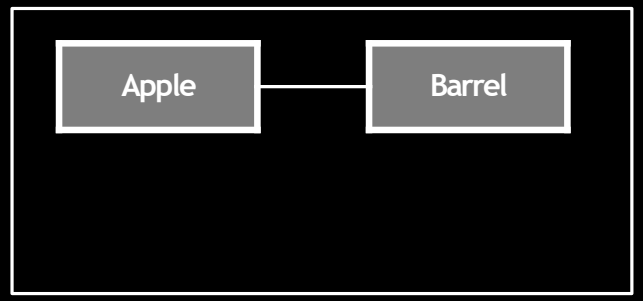


Object Orientation

- APPLES are stored in BARRELS and ORANGES are stored in BASKETS.

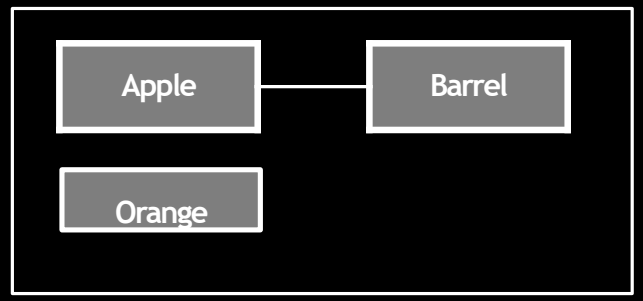
Object Orientation

- APPLES are stored in BARRELS and ORANGES are stored in BASKETS.



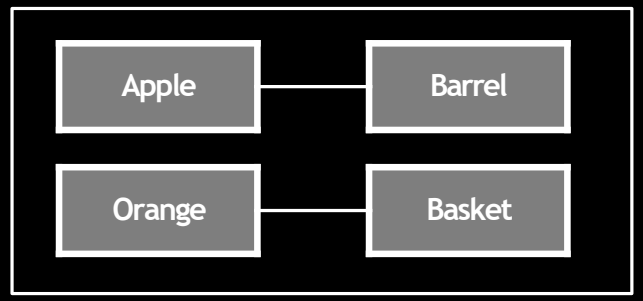
Object Orientation

- APPLESare stored in BARRELSand ORANGESare stored in BASKETS.



Object Orientation

- APPLESare stored in BARRELSand ORANGESare stored in BASKETS.

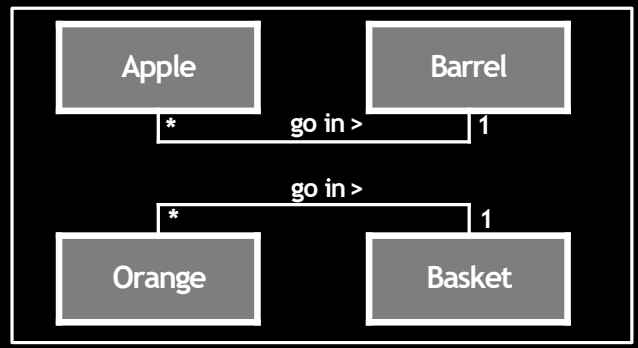


Object Orientation

- Let's add in more rules to the scenario:
  - Many APPLES are stored in one BARREL and many ORANGES are stored in one BASKET.

Object Orientation

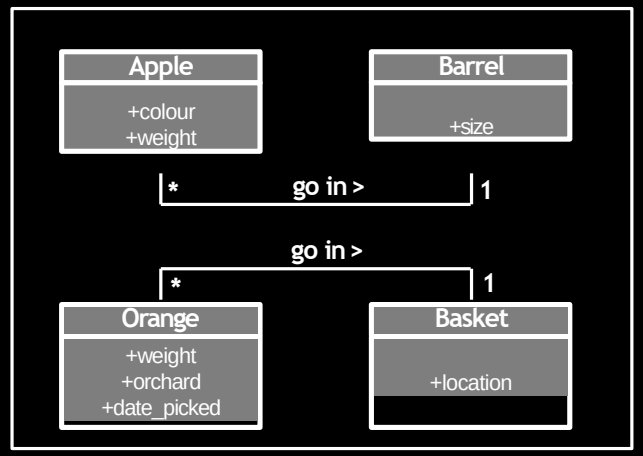
- Many APPLES are stored in one BARREL and many ORANGES are stored in one BASKET.



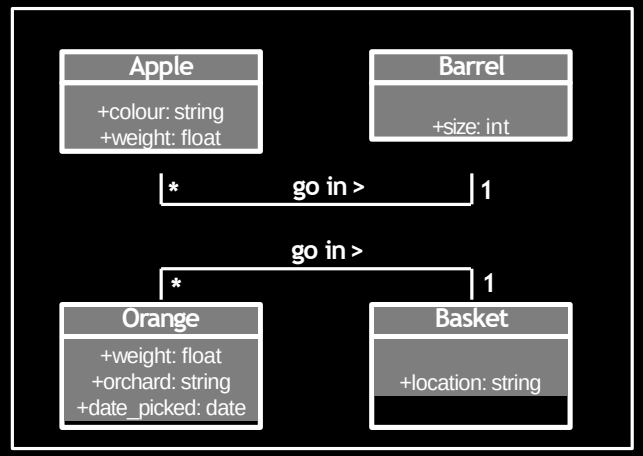
Object Orientation

- Let's add some attributes:

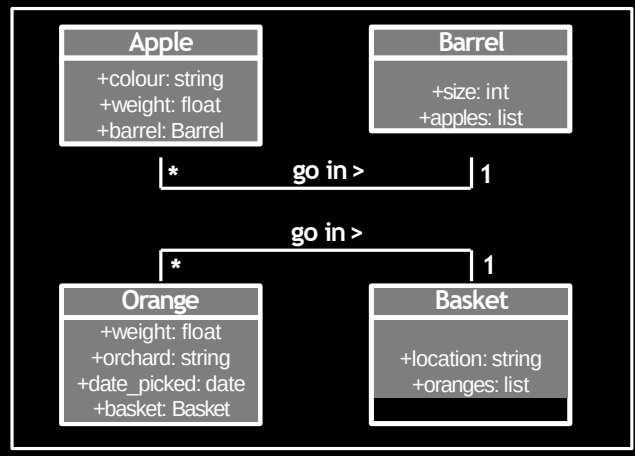
Object Orientation



Object Orientation



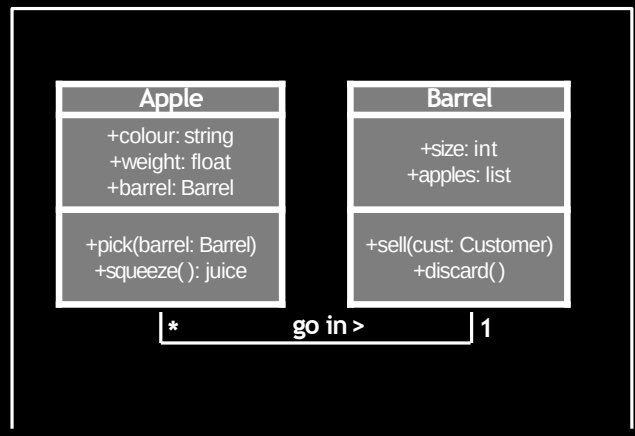
Object Orientation



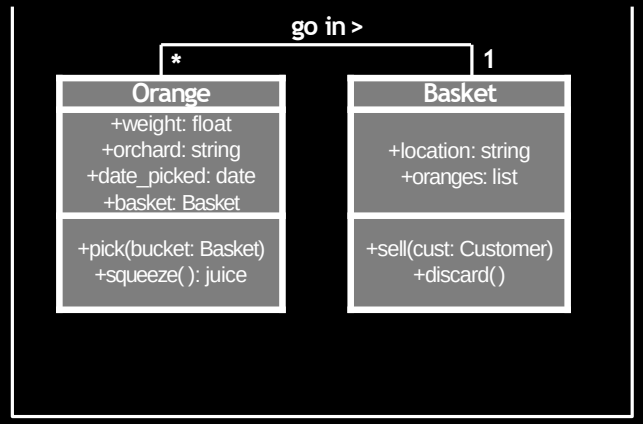
Object Orientation

- Now let's add some methods:

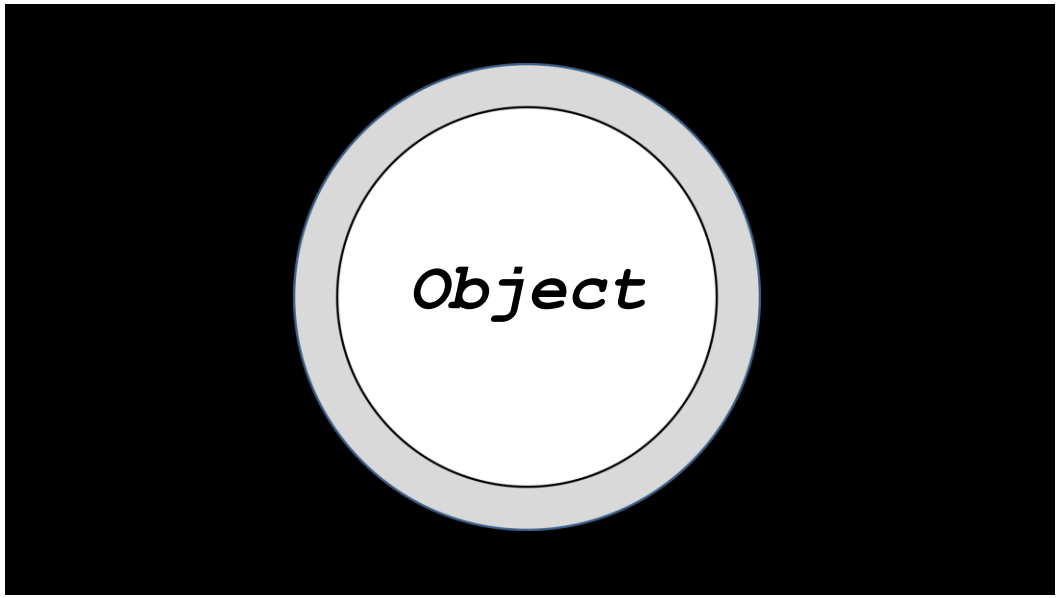
Object Orientation

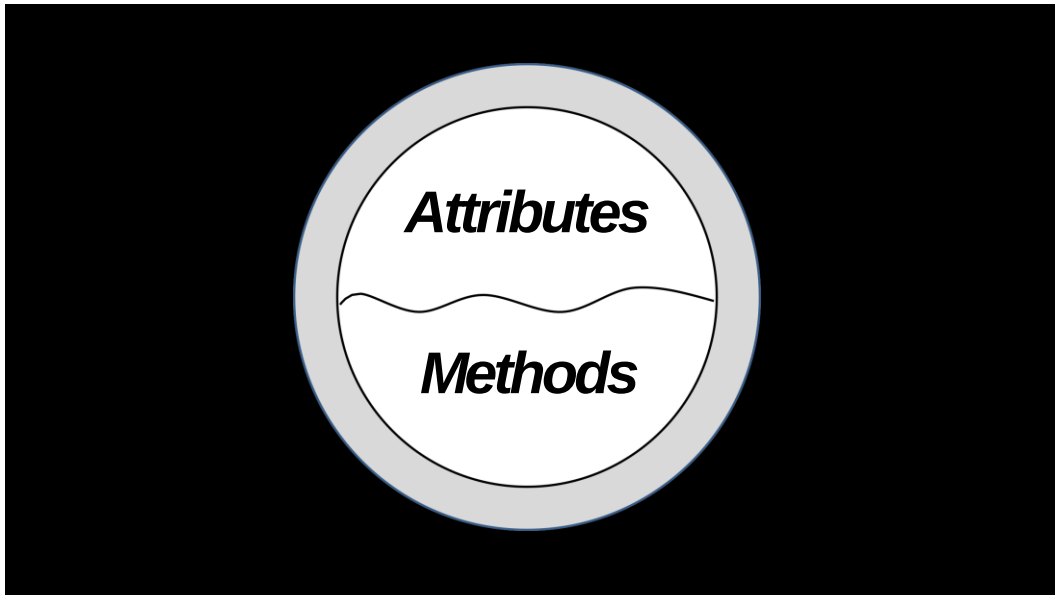


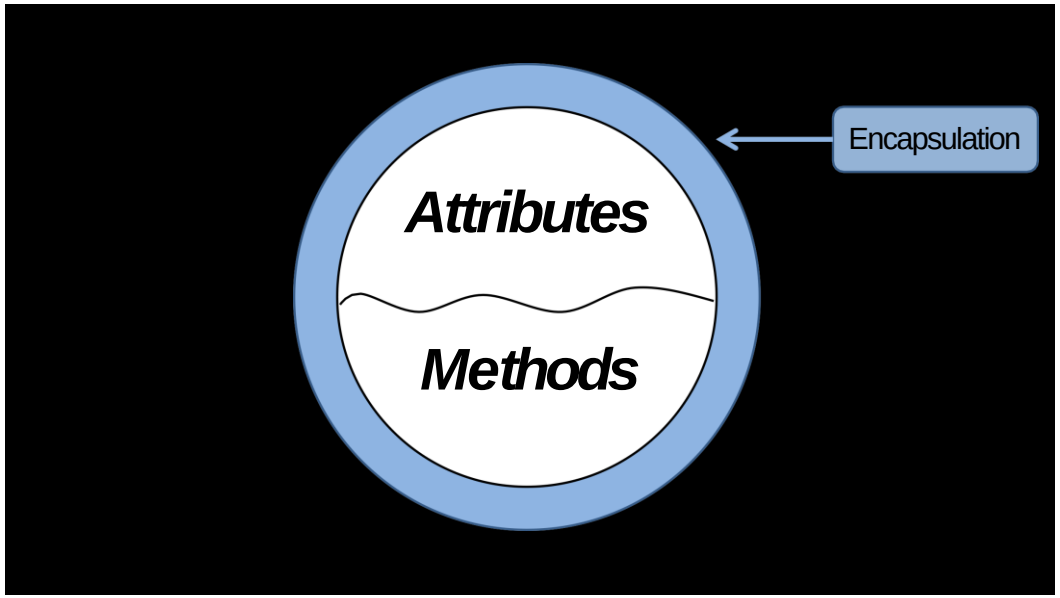
Object Orientation



Information Hiding

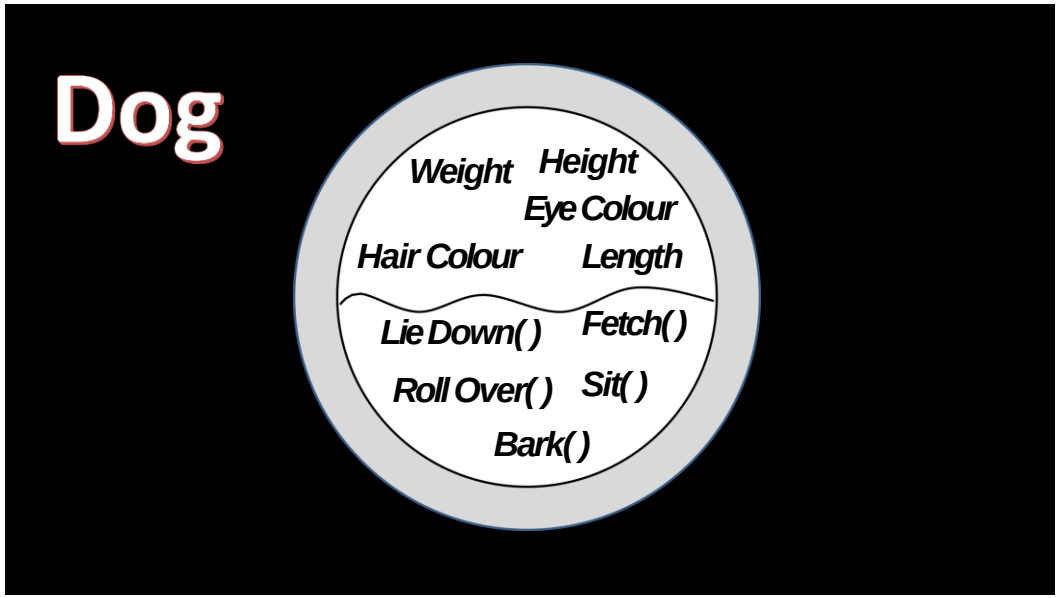

Object

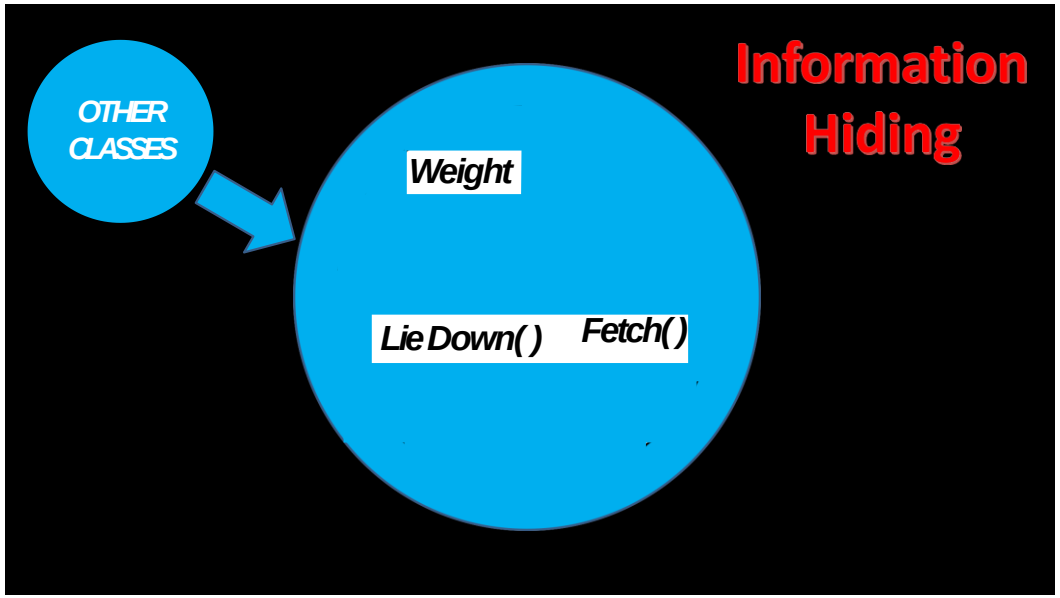

Attributes
Methods



Attribute or Method Visibility

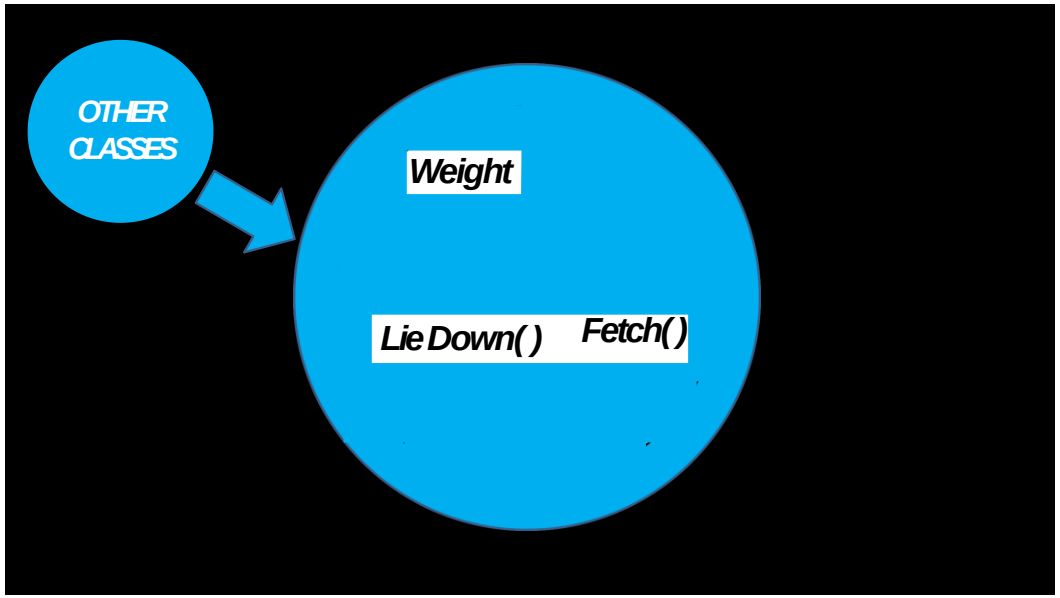
- Public +
- Private –
- Protected #

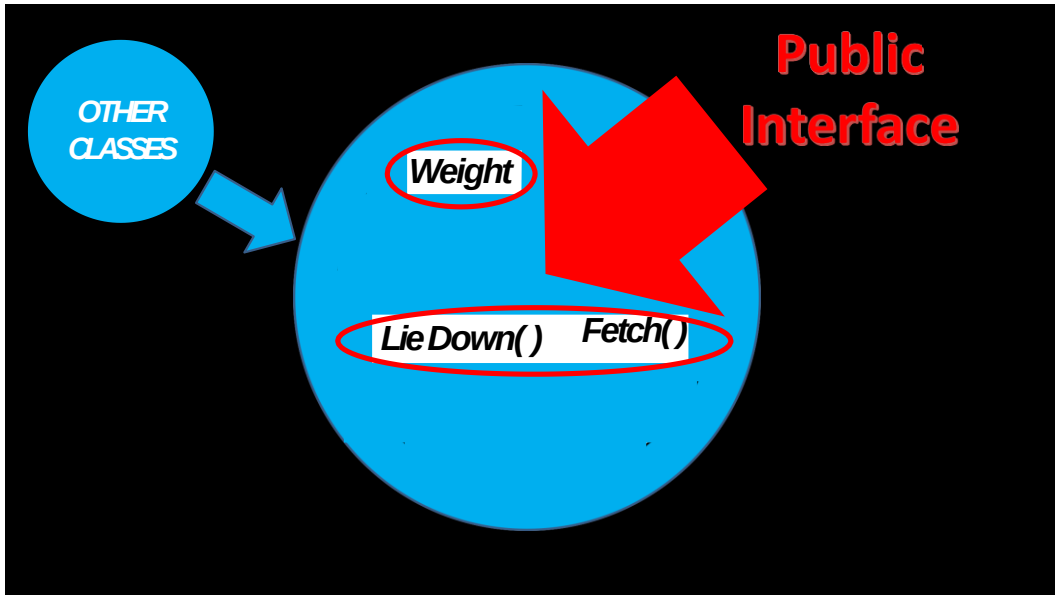

Dog
Weight
Height
Eye Colour
Hair Colour
Length
Lie Down( )
Fetch( )
Roll Over( )
Sit( )
Bark( )


Information Hiding
OTHER CLASSES
Weight
Lie Down( )
Fetch( )

Object Orientation

- An OBJECT is made up of ATTRIBUTES and METHODS. Some of these can be private to the CLASS, and some can be public and used by other classes.
- The public elements (ATTRIBUTES and METHODS) of the CLASS are referred to as the INTERFACE (or PUBLIC INTERFACE).





Object Orientation

- A common real-world example is the television. Our interface to the television is the remote control.
- Each button on the remote control represents a method that can be called on the television object.

Object Orientation

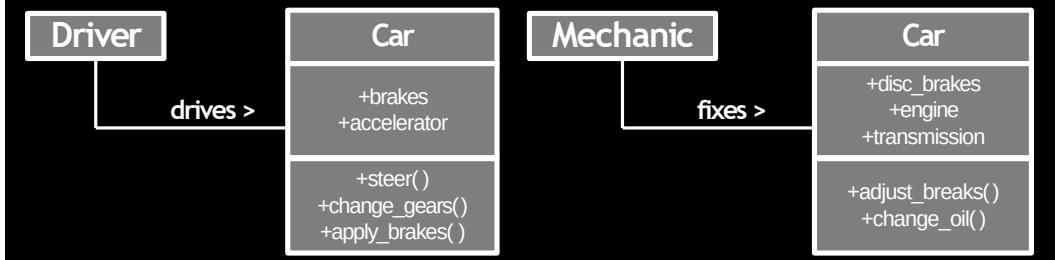
- The public interface is very important, we have to design it carefully, because it can be difficult to change in the future, and changing it will cause problems for any client objects that are calling it.
- We can change the internals as much as we like (e.g. to make it more efficient, or to access data over the network as well as locally) and the client objects will still be able to talk to it, unmodified, using the public interface.

Abstraction

Object Orientation

- ABSTRACTION means dealing with the level of detail that is most appropriate to a given task.
- For example, a DRIVER of a car needs to interact with steering, gas pedal, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A MECHANIC, on the other hand, works at a different level of abstraction, tuning the engine and bleeding the breaks.

Object Orientation

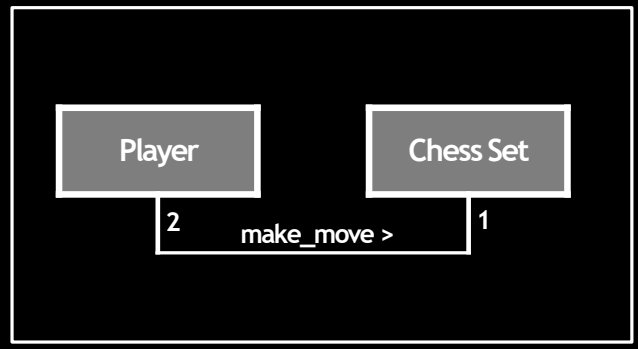


Object Orientation

- Let's try modelling computer chess as a CLASS DIAGRAM:

Object Orientation

- Let's try modelling computer chess as a CLASS DIAGRAM:

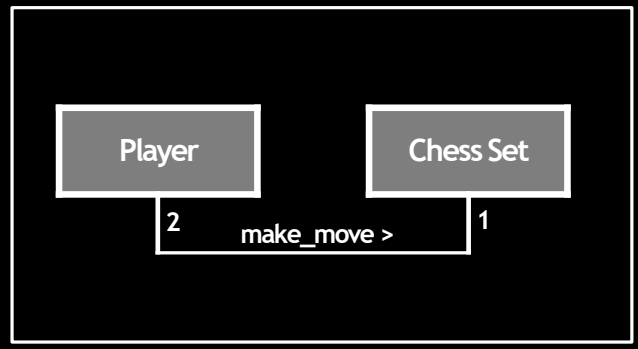


Object Orientation

- The diagram shows that exactly two players can interact with one chess set.
- This also indicates that any one player can be playing with only one chess set at a time.

Object Orientation

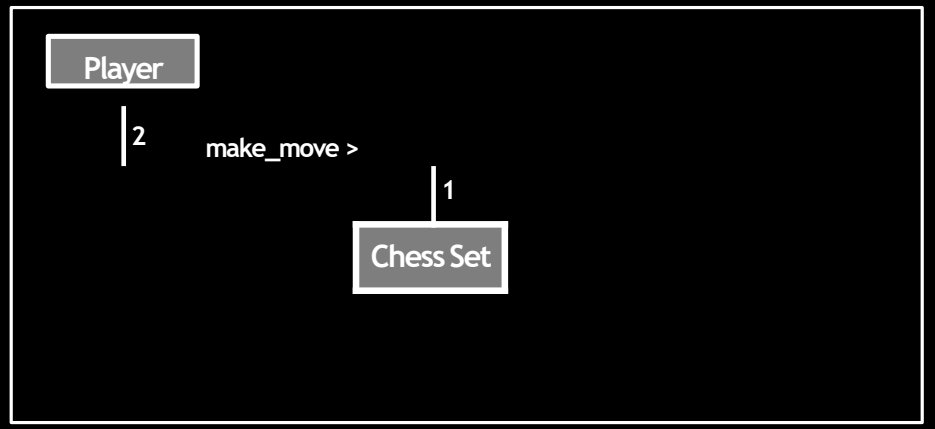
- Lets return to the classdiagram:

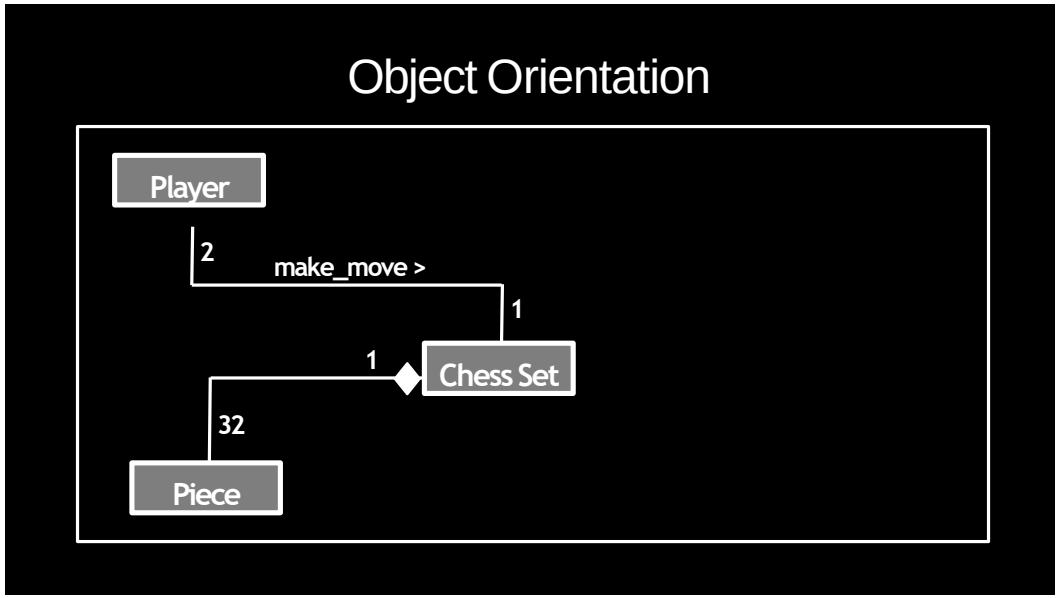


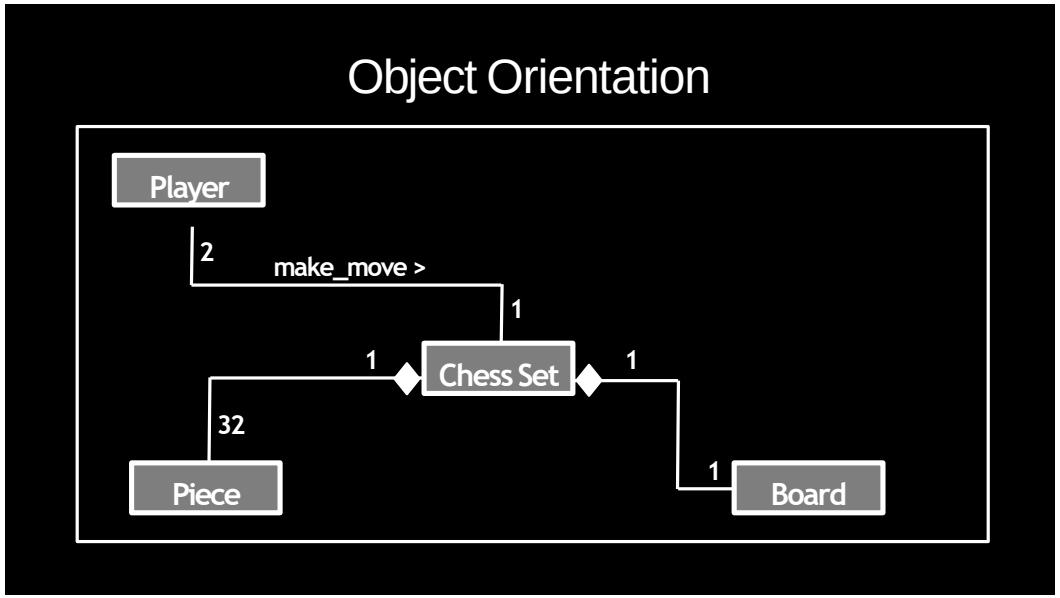
Object Orientation

- Thinking about the composition of a Chess Set we know it's made up of a board and 32 pieces.
- The board further comprises 64 positions.
- The composition relationship is represented in UML as a solid diamond.

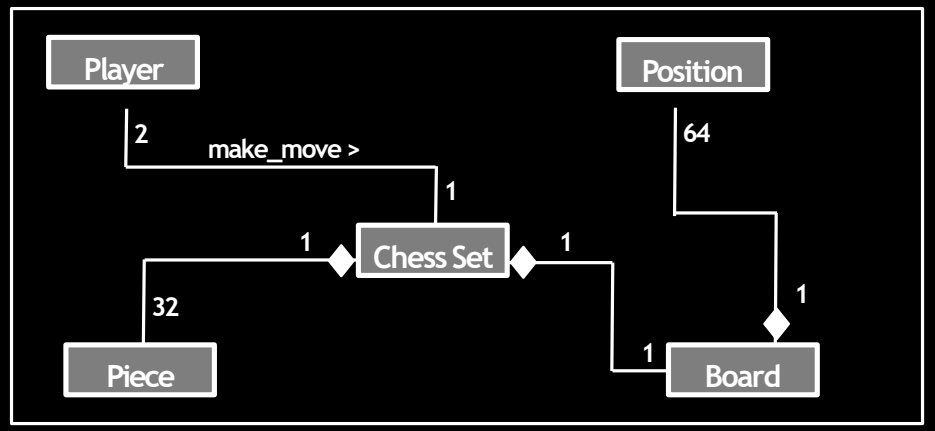
Object Orientation




Object Orientation
Player
2
make_move >
1
1
Chess Set
32
Piece

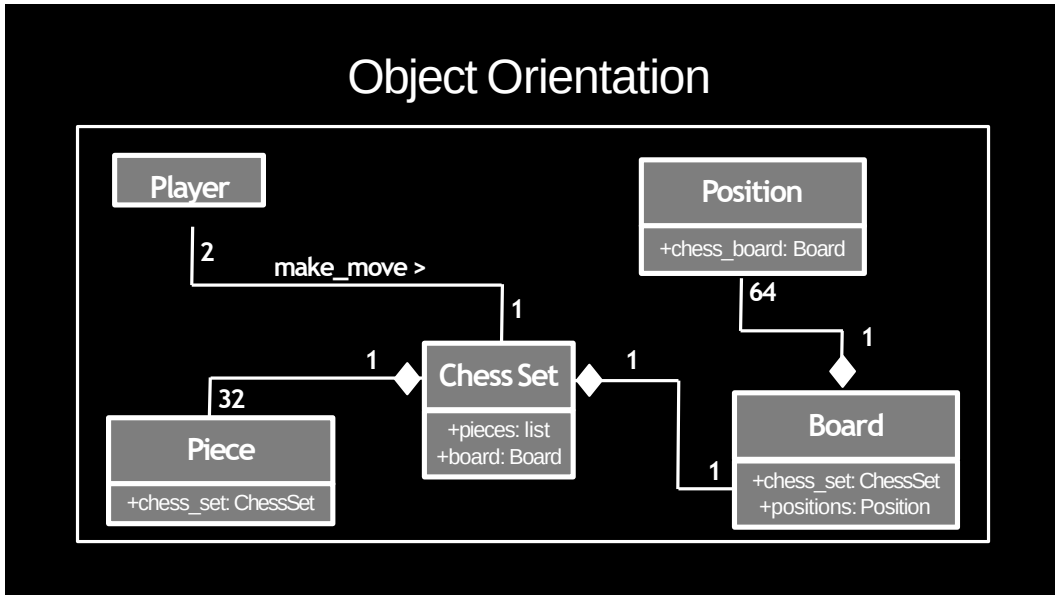

Object Orientation
Player
2
make_move >
1
1
Chess Set
1
32
Piece
1
Board

Object Orientation



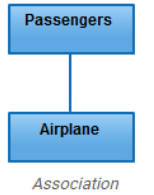
Object Orientation

- Finally let's add some attributes to the class diagram:

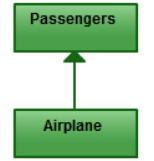


Relationship in Class Diagram

- Association:
  - is a broad term that encompasses just about any logica connection or relationship between classes. Fc example, passenger and airline may be linked as above
- Directed Association:
  - refers to a directional relationship represented by a line with an arrowhead. An airplane can have passengers.



Association



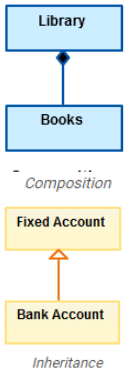
Directed Association

Relationship in Class Diagram

- Multiplicity:
  - is the active logical association when the cardinality of a class in relation to another is being depicted. For example, while one commercial airplane may contain zero to many passengers. The notation 0..* in the diagram means “zero to many”.
- Aggregation:
  - refers to the formation of a particular class as a result of one class being aggregated or built as a collection. For example, the class “library” is made up of one or more books, among other materials. In aggregation, the contained classes are not strongly dependent on the lifecycle of the container. In the same example, books will remain so even when the library is dissolved.

Relationship in Class Diagram

- Composition:
  - The composition relationship is very similar to the aggregation relationship. with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, the contained class will be obliterated when the container class is destroyed. For example, a shoulder bag’s side pocket will also cease to exist once the shoulder bag is destroyed.
- Inheritance:
  - refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To show inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.



INHERITANCE

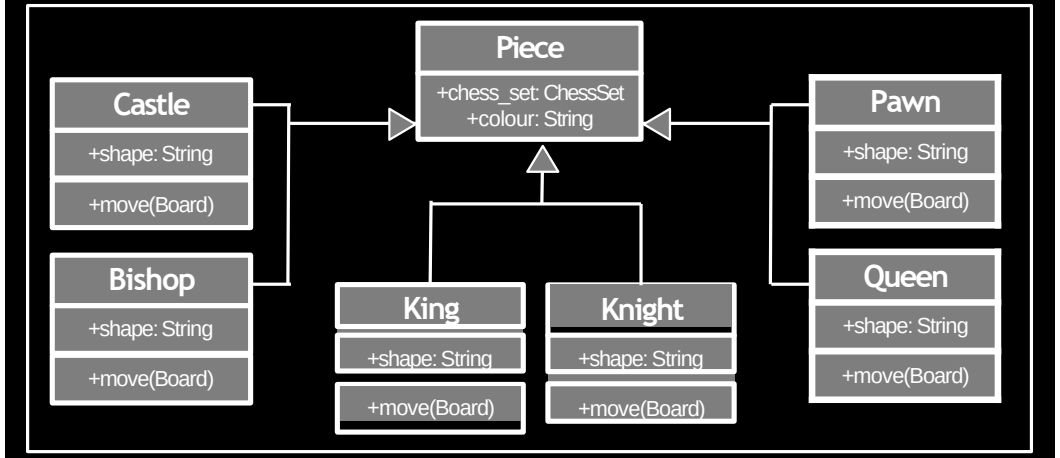
Object Orientation

- INHERITANCE is the last type of relationship we'll look at.
- Inheritance is simple, it means that one class can inherit attributes and methods from another class.
- So often we look for classes that have a lot in common, and create a single class with those commonalities, and use that class to give those features to all the other classes.

Object Orientation

- Let's look at an example of chess pieces:
- We know that all PIECES are part of a CHESS_SET and have a COLOUR (so they should be in our BASE CLASS).
- Each piece has a different SHAPE attribute and a different MOVE method to move the piece to a new position on the board at each turn.

Object Orientation



Object Orientation

- Inheritance also gives us POLYMORPHISM.
- Polymorphism is the ability to treat a class differently depending on which subclass is implemented.
- All the board has to do is call the `move()` method of a given piece, and the proper subclass will take care of moving it as a Knight or a Pawn.

OOP IN PYTHON

Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- *Instances* are objects that are created which follow the definition given inside of the class
- You just define the class and then use it

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block
- There must be a special first argument `self` in *all* of method definitions which gets bound to the calling instance
- There is usually a special method called `__init__` in most classes
- We'll talk about both later…

A simple class def: *student*

```
class student:
 """A class representing a student """
 def __init__(self,n,a):
     self.full_name = n
     self.age = a
 def get_age(self):
     return self.age
```

Creating Instances

Instantiating Objects

- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student(“Bob”, 21)
```

Constructor: __init__

- An `__init__` method can take any number of arguments.
- Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.
- However, the first argument `self` in the definition of __init__ is special…

A simple class def: *student*

```
class student:
 """A class representing a student """
 def __init__(self,n = 'None' ,a = 00):
     self.full_name = n
     self.age = a
 def get_age(self):
     return self.age
```

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***self***
- In `__init__`, *self* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called

Self

- Although you must specify `self` explicitly when ***defining*** the method, you don't include it when ***calling*** the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Access to Attributes and Methods

Definition of student

```
class student:
  """A class representing a student """
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student("Bob Smith", 23)

>>> f.full_name # Access attribute
"Bob Smith"

>>> f.get_age() # Access a method
23
```

Attributes

Two Kinds of Attributes

- The non-method data stored by objects are called attributes
- *Data* attributes
  - Variable owned by a *particular instance* of a class
  - Each instance has its own value for it
  - These are the most common kind of attribute

- *Class* attributes
  - Owned by the *class as a whole*
  - *All class instances share the same value for it*
  - Called “static” variables in some languages
  - Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
  - Simply assigning to a name creates the attribute
  - Inside the class, refer to data attributes using **`self`**
    - for example, **`self.full_name`**

```
class teacher:
  "A class representing teachers."
  def __init__(self,n):
      self.full_name = n
  def print_name(self):
      print self.full_name
```

Class Attributes

- Because all instances of a class share one copy of a class attribute, when *any* instance changes it, the value is changed for *all* instances
- Class attributes are defined *within* a class definition and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
  - Access class attributes using **`self.__class__.name`** notation -- This is just one way to do this & the safest in general.

```
class sample:
    x = 23
  def increment(self):
      self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
   overall_total = 0
        # class attribute
   def __init__(self):
       self.my_total = 0
         # data attribute
   def increment(self):
      counter.overall_total = \
      counter.overall_total + 1
      self.my_total = \
      self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Inheritance

Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.

- To define a subclass, put the name of the parentclass in parentheses after the subclass's name on the first line of the definition.

```
Class Cs_student(student):
```

Redefining Methods

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.

- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent's version of the method.

```
parentClass.methodName(self, a, b, c)
```

  - **The only time you ever explicitly pass 'self' as an argument is when calling a method of an ancestor.**

Definition of a class extending student

```
Class Student:
   "A class representing a student."

   def __init__(self,n,a):
       self.full_name = n
       self.age = a

   def get_age(self):
       return self.age
-----------------------------------------

Class Cs_student (student):
   "A class extending student."

   def __init__(self,n,a,s):
       student.__init__(self,n,a) #Call __init__ for student
       self.section_num = s

   def get_age():    #Redefines get_age method entirely
       print "Age: " + str(self.age)
```

Extending __init__

- Same as for redefining any other method…
  - Commonly, the ancestor's **`__init__`** method is executed in addition to new commands.
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

Modules, revisited

A module is a file containing Python code.

When the module is executed (imported), then the module is (also) a namespace.

- This namespace has a name, typically the name of the module.
- In this namespace live the names that are defined in the global scope of the module: the names of functions, values, and classes defined in the module.
- These names are the module's attributes.

Built-in function `dir()` returns the names defined in a namespace

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']
>>> math.sqrt
<built-in function sqrt>
>>> math.pi
3.14159265358979
```

To access the imported module's attributes, the name of the namespace must be specified

Importing a module

When the Python interpreter executes an `import` statement, it:

1. Looks for the file corresponding to the module to be imported.
2. Runs the module's code to create the objects defined in the module.
3. Creates a namespace where the names of these objects will live.

An import statement only lists a name, the name of the module

- without any directory information or .py suffix.

Python uses the Python search path to locate the module.

- The search path is a list of directories where Python looks for modules.
- The variable name `path` defined in the Standard Library module `sys` refers to this list.

current working directory

Standard Library folders

```
>>> import sys
>>> sys.path
['/Users/me', '/Library/Frameworks/Python.framework/Versions/3.2/lib/python32.zip',
. . .
'/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages']
>>>
```

The Python search path

Suppose we want to import module `example` stored in folder `/Users/me` that is not in list `sys.path`

names in the shell namespace; note that `example` is not in

```
>>> ================ RESTART ================
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

When called without an argument, function `dir()` returns the names in the top-level module

- the shell, in this case.

The Python search path

By just adding folder `/Users/me` to the search path, module `example` can be imported

no folder in the Python search path contains module `example`

```
>>> ================ RESTART ================
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> import example
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    import example
ImportError: No module named example
>>> import sys
>>> sys.path.append('/Users/me')
>>> import example
>>> example.f
<function f at 0x10278dc88>
>>> example.x
0
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__',
'example', 'sys']
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

When called without an argument, function `dir()` returns the names in the top-level module

- the shell, in this case.

Top-level module

A computer application is a program typically split across multiple modules.

One of the modules is special: It contains the "main program". This module is referred to as the top-level module.

- The remaining modules are "library" modules that are imported by other modules and that contain functions and classes used by it

When a module is imported, Python creates a few "bookkeeping" variables in the module namespace, including variable `__name__`:

- set to `'__main__'`, if the module is being run as a top-level module

A module is a top-level module if:

- it is run from the shell
- it is run at the command line

```
print('My name is {}'.format(__name__))
```

name.py

```
>>> === RESTART ===
>>>
My name is __main__
>>>
```

```
> python name.py
My name is __main__
```


Top-level module

A computer application is a program typically split across multiple modules.

One of the modules is special: It contains the "main program". This module is referred to as the top-level module.

- The remaining modules are "library" modules that are imported by the top-level module and that contain functions and classes used by it

When a module is imported, Python creates a few "bookkeeping" variables in the module namespace, including variable `__name__`:

- set to `'__main__'`, if the module is being run as a top-level module
- set to the module's name, if the file is being imported by another module

```
>>> import name
My name is name
```

```
>>> === RESTART ===
>>>
My name is name
```

```
print('My name is {}'.format(__name__))
```

name.txt

```
import name
```

import.txt

Three ways to import module attributes

1. Import the (name of the) module

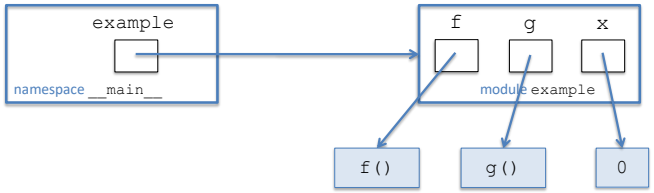
```
>>> import example
>>> example.x
0
>>> example.f
<function f at 0x10278dd98>
>>> example.f()
Executing f()
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



Three ways to import module attributes

2. Import specific module attributes

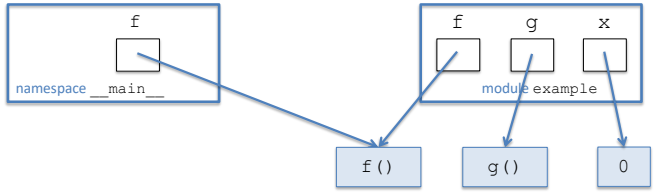
```
>>> from example import f
>>> f()
Executing f()
>>> x
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



Three ways to import module attributes

3. Import all module attributes

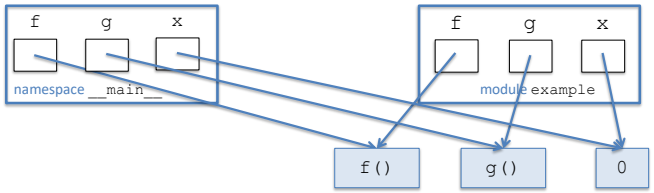
```
>>> from example import *
>>> f()
Executing f()
>>> g()
Executing g()
>>> x
0
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



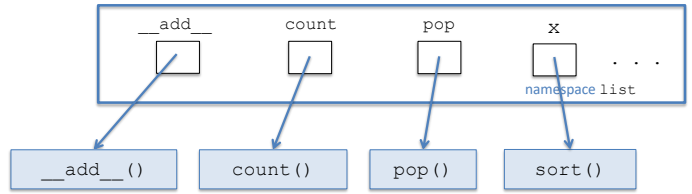
A class is a namespace

A class is really a namespace

- The name of this namespace is the name of the class
- The names defined in this namespace are the class attributes (e.g., class methods)
- The class attributes can be accessed using the standard namespace notation

```
>>> list.pop
<method 'pop' of 'list' objects>
>>> list.sort
<method 'sort' of 'list' objects>
>>> dir(list)
['__add__', '__class__',
...
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Function `dir()` can be used to list the class attributes



Class methods

A class method is really a function defined in the class namespace; when Python executes

```
instance.method(arg1, arg2, …)
```

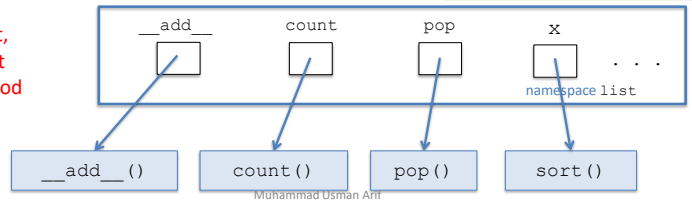
it first translates it to

```
class.method(instance, arg1, arg2, …)
```

and actually executes this last statement

The function has an extra argument, which is the object invoking the method

```
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> lst.sort()
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> list.sort(lst)
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst.append(6)
>>> lst
[1, 2, 3, 7, 8, 9, 6]
>>> list.append(lst, 5)
>>> lst
[1, 2, 3, 7, 8, 9, 6, 5]
```



Exercise

Rewrite the below Python statement so that instead of making the usual method invocations

```
instance.method(arg1, arg2, …)
```

you use the notation

```
class.method(instance, arg1, arg2, …)
```

```
>>> s = 'hello'
>>> s = 'ACM'
>>> s.lower()
'acm'
>>> s.find('C')
1
>>> s.replace('AC', 'IB')
'IBM'
```

```
>>> s = 'ACM'
>>> str.lower(s)
'acm'
>>> str.find(s, 'C')
1
>>> str.replace(s, 'AC', 'IB')
'IBM'
>>>
```