

Python Object-Oriented Program with Libraries

Unit 1: Object-Oriented Programming

CHAPTER 2: CLASS RELATIONSHIP AND CLASS DESIGN

DR. ERIC CHOU

IEEE SENIOR MEMBER



Topic of this Chapter

1. Encapsulation of Python Classes (Data Class and Functional Closure Class)
2. Scope of Python Module and Classes
3. Python Pointers (alias)
4. Python Class as Record (struct in C/C++)
5. Method as object and Calling Instance Method using Class Specifier
6. Class/Module as Namespace
7. Class to Class Relationship (Is_A and has_A Relationship)



Topic of this Chapter

- 8. Class Design Principles (Quality of Modules, Coupling and Coherence)
- 9. Python Object Model
- 10. Meta-class

Encapsulation

LECTURE 1



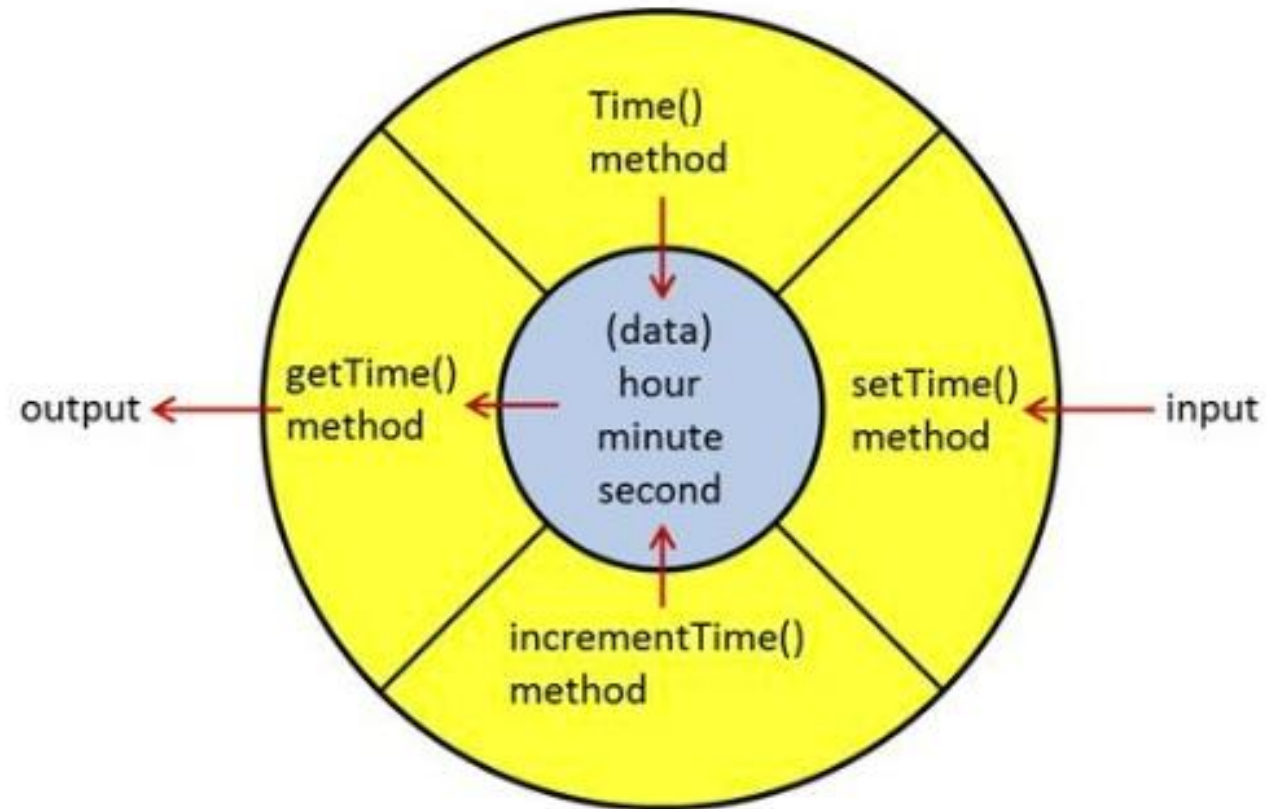
Why Encapsulate?

By defining a specific interface you can keep other modules from doing anything incorrect to your data

By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users

- Write to the Interface, not the Implementation
- Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

Encapsulation





Public and Private Data

Currently everything in atom/molecule is public, thus we could do something **really stupid** like

- `>>> at = atom(6,0.,0.,0.)`
- `>>> at.position = 'Grape Jelly'`

that would break any function that used `at.position`

We therefore need to protect the `at.position` and provide accessors to this data

- Encapsulation or Data Hiding
- accessors are "getters" and "setters"

Encapsulation is particularly important when other people use your class



Public and Private Data, Cont.

In Python anything with two leading underscores is private

- `__a`, `__my_variable`

Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.

- `_b`
- Sometimes useful as an intermediate step to making data private



Encapsulated Atom Using Private Data Field and Public Methods

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.__position = (x,y,z) #position is private
    def getposition(self):
        return self.__position
    def setposition(self,x,y,z):
        self.__position = (x,y,z) #typecheck first!
    def translate(self,x,y,z):
        x0,y0,z0 = self.__position
        self.__position = (x0+x,y0+y,z0+z)
```



Making Class as Container or Function Closure

__getattr__(self): use the object as RHS variable.

__setattr__(self, value): use the object as LHS = value.

__getitem__(self, index) : this function make the object instance act likes a container, array in this case. object[index] as an element in an array. (RHS)

__setitem__(self, index, value) : this function make the object instance act likes a container, array in this case. object[index] as an element in an array. (LHS = value)

__call__: this function makes the object instance act likes a function, object(parameter) as a function call.



Function Closure Class

Overload `__call__(self,arg)` to make a class behave like a function

class gaussian:

```
    def __init__(self,exponent):
```

```
        self.exponent = exponent
```

```
    def __call__(self,arg):
```

```
        return math.exp(-self.exponent*arg*arg)
```

```
>>> func = gaussian(1.)
```

```
>>> func(3.)
```

```
0.0001234
```



Demo Program:

`gaussian.py`

Go PyCharm!!!

Function Closure Example

Python

```
def step_range(step):  
    def get_range(m, n):  
        return range(m, n + 1, step)  
    return get_range
```

```
step1 = step_range(1)
```

```
# step1(3, 7) ~> [3, 4, 5, 6, 7]
```

Note:

Function closure and Functional closure class are different



Container Class

Overload `__getitem__(self,index)` to make a class act like an array

```
class molecule:
```

```
    def __getitem__(self,index):
```

```
        return self.atomlist[index]
```

```
>>> mol = molecule('Water') #defined as before
```

```
>>> for _atom in mol:    #use like a list!
```

```
    print(_atom)
```

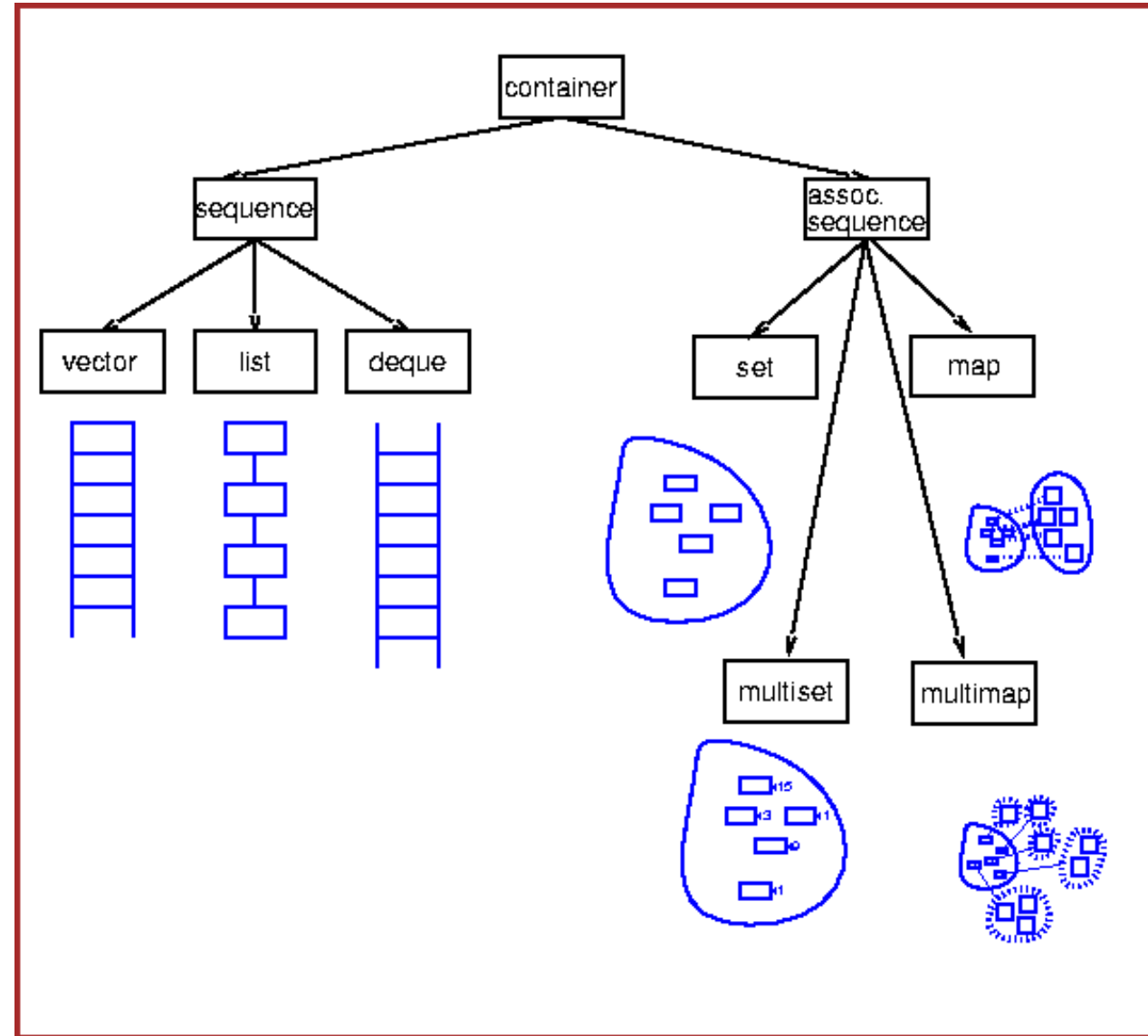
```
    # use this semi-private name _atom to prevent from redefining of atom
```

```
>>> mol[0].translate(1.,1.,1.)
```

Previous lectures defined molecules to be arrays of atoms.

This allows us to use the same routines, but using the molecule class instead of the old arrays.

An example of focusing on the interface!





Demo Program:
Molecule2 project (atom.py+molecule.py)

Go PyCharm!!!



Object as Container

When you implement a container type in your codebase you make working with those types a lot more enjoyable and the code using those types will be a lot more readable.

Functions to support object as container:

Object string representation using

`__str__`

Length of an object using

`__len__`

Iterating over an object using

`__iter__`

Getting an item using

`__getitem__`

Setting an item using

`__setitem__`

Deleting an item using

`__delitem__`

Testing membership using

`__contains__`

Python Names and Objects

LECTURE 2



Introduction to namespaces and scopes

Namespaces

- Roughly speaking, namespaces are just **containers for mapping names to objects**. As you might have already heard, everything in Python - literals, lists, dictionaries, functions, classes, etc. - is an object.
- Such a “name-to-object” mapping allows us to access an object by a name that we’ve assigned to it. E.g., if we make a simple string assignment via **a_string = "Hello string"**, we created a reference to the **"Hello string"** object, and henceforth we can access via its variable name **a_string**.
- We can picture a namespace as a Python **dictionary** structure, where the dictionary keys represent the names and the dictionary values the object itself (and this is also how namespaces are currently implemented in Python), e.g.,

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
```



Introduction to namespaces and scopes

Namespaces

- Now, the tricky part is that we have multiple independent namespaces in Python, and names can be reused for different namespaces (only the objects are unique, for example:

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}  
b_namespace = {'name_a':object_3, 'name_b':object_4, ...}
```

- For example, every time we call a **for-loop** or define a function, it will create its own namespace. Namespaces also have different levels of hierarchy (the so-called “scope”), which we will discuss in more detail in the next section.



Names and Objects

Objects in python have aliasing

- Have individuality
- Multiple names can be bound to the same object

An object's alias behaves like a pointer

- Easier and quicker for program to pass a pointer rather than an object



Scopes and Namespaces

Namespace: mapping from names to objects

- Ex: functions such as `abs()`, global names in a module

Names within namespaces do not relate to each other

- Two modules may define a function with the same name without the program becoming confused as to which to use

Attributes within namespaces can be both read-only and writable

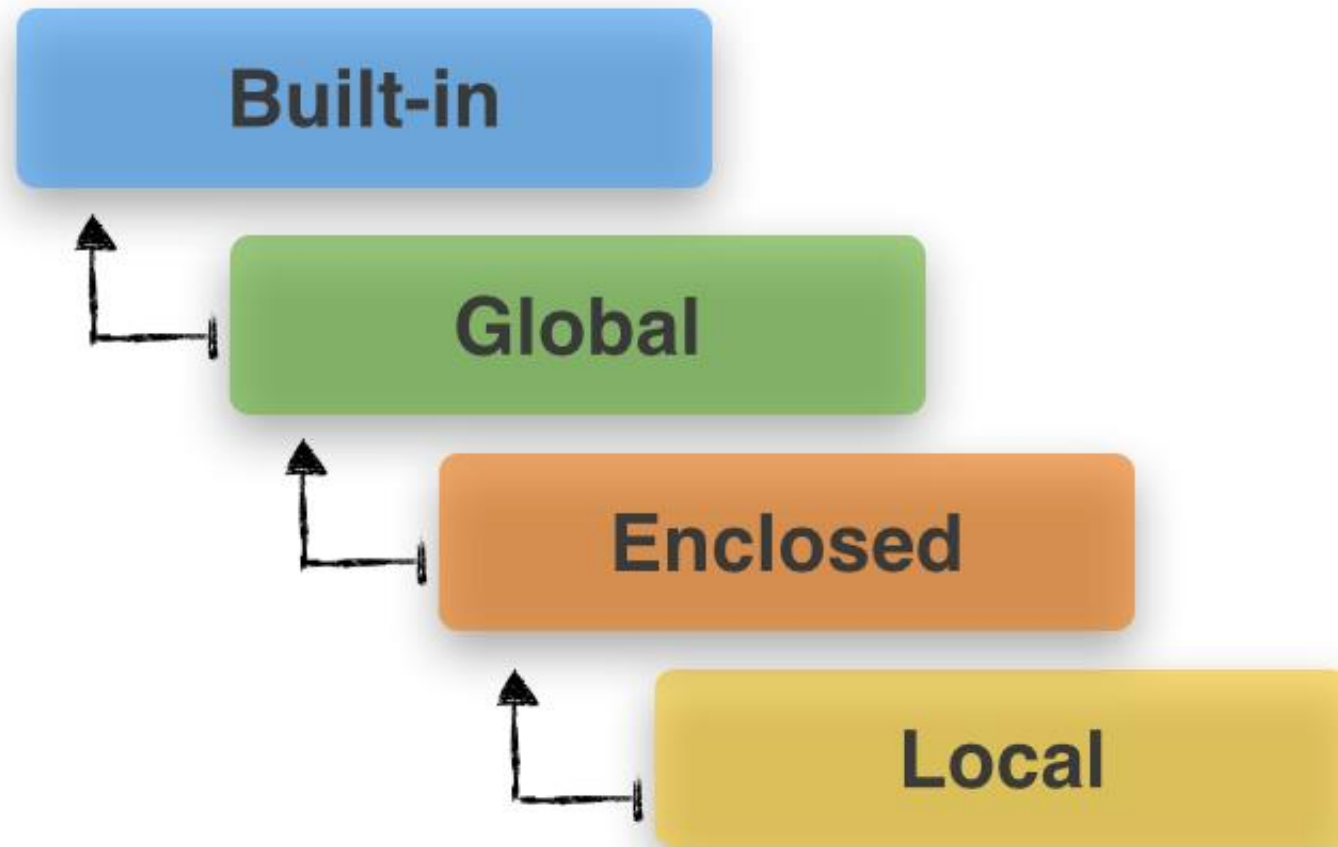
Ex:

```
modname.the_answer = 42
```

To delete attributes, use “del”

```
del modname.the_answer
```

- the attribute “the_answer” will be deleted from modname



Python Namespace

Scope resolution for variable names via the LEGB rule.



Namespace Lifetimes

- Namespaces with built-in names is created when the program begins and is never deleted
- A global namespace for a module is created when the module is read by the program interpreter, and last until the program exits
- Local namespaces are created when functions run, and usually deleted when the function returns a value or raises an exception it cannot handle



Scopes

A scope is a region in the code where the namespace is directly accessible

- References to a name will look here for a definition of the name in the namespace

Nested scopes whose names are directly accessible:

- **Innermost scope** (contains local names)
- **Scopes of any functions** (contains names specific to the function)
- **Next to last scope** (contains global names)
- **Outermost scope** (contains built-in names from namespaces of python)



Demo Program: scope1.py

Go PyCharm!!!

```
Run scope1
C:\Python\Python36\python.exe "C:/Eric_Chou/
a is local variable
called my len() function: 14
6
a is global
Process finished with exit code 0
```

```
a = 'global'

def outer():
    def len(in_var):
        print('called my len() function: ', end="")
        l = 0
        for i in in_var:
            l += 1
        return l

    a = 'local'

    def inner():
        global len
        nonlocal a
        a += ' variable'

    inner()
    print('a is', a)
    print(len(a))

outer()
print(len(a))
print('a is', a)
```



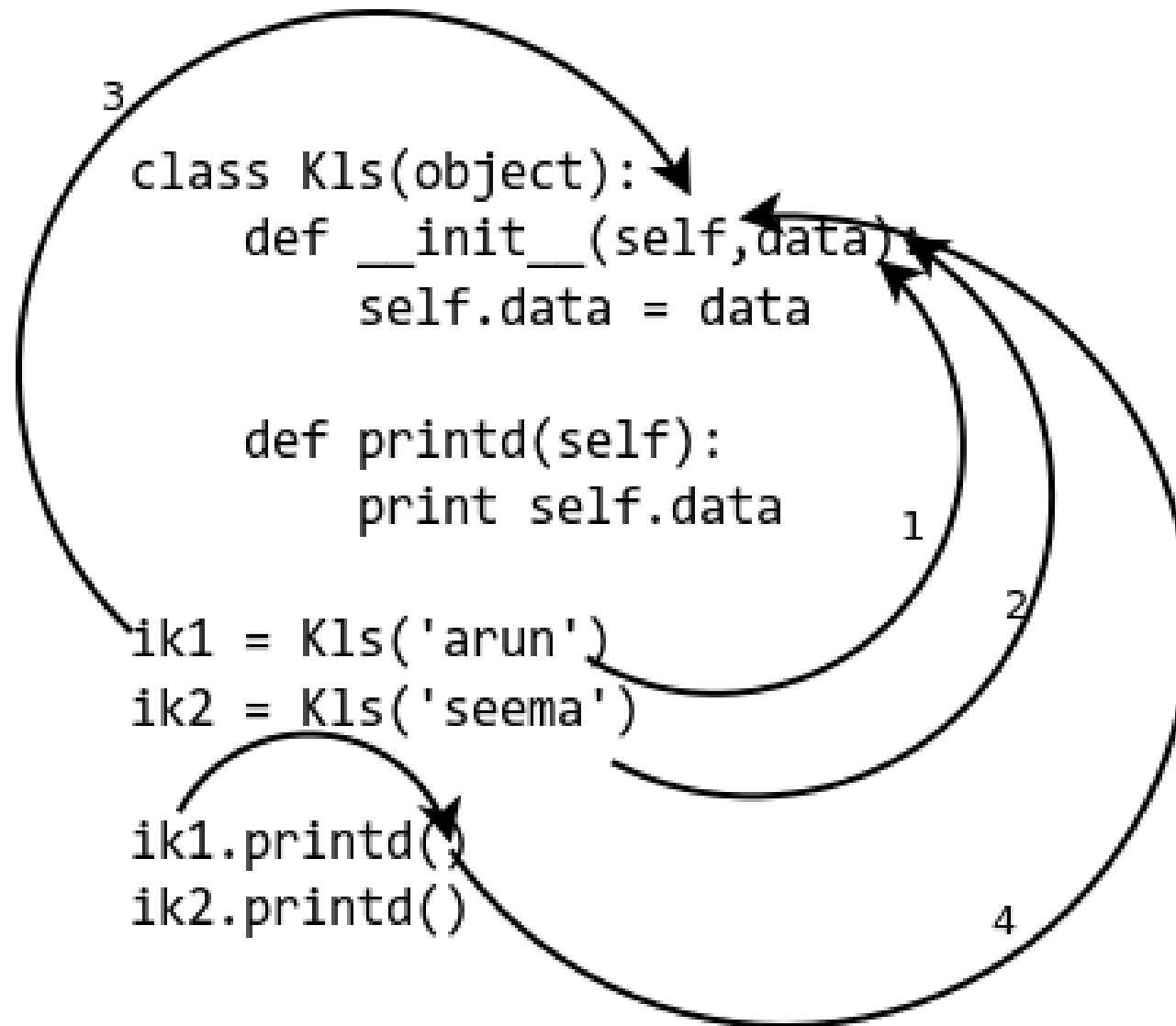
self, super(), master

- self: current object identifier
- super: parent class identifier (see inheritance section)
- master: owner (container) identifier (see tkinter GUI)



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
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*





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)
```

Python Alias

LECTURE 2



Making and Breaking an Alias

alias on immutable objects

- Not all variable names refer to different variables. When two identifiers refer to the same variable (and therefore value), this is known as an alias.
- An alias identifier to an existing variable is created using the form **<identifier>=<identifier>**.
- Note the difference with the more general case of an **<identifier>=<expression>** assignment, where a new value is assigned to a new or existing variable!
- To figure out whether two identifiers refer to the same variable, use the built-in **id()** function, which returns a unique number for each variable. (id() like hashCode() in Java)



Demo Program: alias_make_1.py

- Importantly however, any assignment of a value to the alias identifier will break the alias, and create a separate variable by the same name instead!
- As a result, there is often little practical consequence to Python creating an alias instead of a new variable.

```

# Assign a value to a new variable
a = 5
# Create an alias identifier for this variable
b = a
# Observe how they refer to the same variable!
print(id(a), id(b))
# Create another alias
c = b
# Now assign a new value to b!
b = 3
)# And observe how a and c are still the same variable
)# But b is not
print(a, b, c)
print(id(a), id(b), id(c))
# Now for another quirk, suppose we do this:
b = a
b = 5
)# We used an assignment, but the value didn't actually change
)# So the alias remains unbroken
print(id(a), id(b))

```

```

Run alias_make_1
C:\Python\Python36\python.exe "C:/...
1944167648 1944167648
5 3 5
1944167648 1944167584 1944167648
1944167648 1944167648
Process finished with exit code 0

```

Note: 5 is an immutable object at the same location like Java's literals

alias_make_1.py



Alias on Mutable objects

- Things become more tricky in case of mutable data structures such as `<list>`.
- As we have seen, these variables have methods to change their value without an assignment statement.
- As a consequence, the alias remains in place and both identifiers continue to refer to the same, changed value.
- For instance, Demo Program: `alias_make_2.py`

```
# Create a new <list>
a = [5]

# Create an alias identifier for this list
b = a
print(id(a), id(b))

# Now change the <list> b in-place
b.append(1)

# And observe how this also changes a
# The alias is not broken by in-place operations
print(a, b)
print(id(a), id(b))
```

```
Run alias_make_2
C:\Python\Python36\python.exe "C:/Eric
2306345197448 2306345197448
[5, 1] [5, 1]
2306345197448 2306345197448
Process finished with exit code 0
```

Note:
Both a and b are just pointers

alias_make_2.py



Forcing a copy

- Obviously, an alias might not always be what you want. You might want to use the value of an existing <list> to perform further operations on, without changing the original value.
- The best and most general way to create a copy instead of an alias is to explicitly construct its type, or in case of a <list>, to use a full slice: Demo Program: `alias_make_3.py`



List Constructor (Copy Constructor)

Copy and Create a New List

list() constructor

```
a = [1, 2, 3]
```

```
b = list(a) # list() is a list constructor to create a new list based on the old list a.
```

List slicing constructor

```
b = a[:] # slice
```

```
# Create a <list>
```

```
a = [5]
```

```
# Create a new <list> with the same value
```

```
b = list(a)
```

```
# We now have two separate variables with identical but separate values
```

```
print(a, b)
```

```
print(id(a), id(b))
```

```
# Same with the full slice technique:
```

```
b = a[:]
```

```
print(a, b)
```

```
print(id(a), id(b))
```

```
Run alias_copy_1
C:\Python\Python36\python.exe
[5] [5]
2497502339144 2497502342088
[5] [5]
2497502339144 2497502340872
```

alias_copy_1.py



The Use of Aliases

- Why would you want to refer to one and the same variable by two different names? Ordinarily, you don't. However, some Python programming constructs automatically make use of aliases.
- In the next section on functions, we will see that function argument identifiers are actually an alias to the variable they represent outside the function, with some consequences.
- One other case to consider is the for loop. Consider this case. As the loop iterates over a sequence **v_list**, each item of the sequence is referred to by the identifier **i**. This is expected behavior:
- Demo Program: `alias_use_1.py`

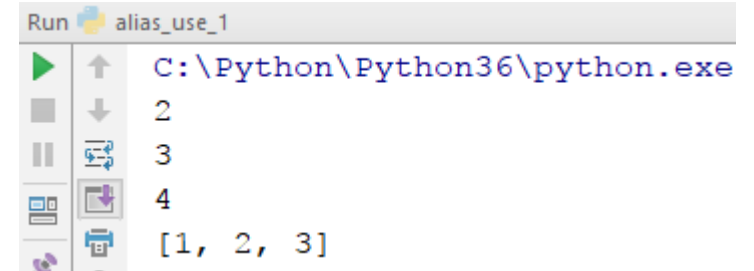


Demo Program: alias_use_1.py

```
v_list = [1, 2, 3]
```

```
for i in v_list:  
    i = i + 1  
    print(i)
```

```
print(v_list)
```



Note: i is pointing to a new object after this update statement



Demo Program: alias_use_2.py

- The fact that each `i` is modified inside the loop, does not change the original `v_list`. But now, if each `i` is a mutable data structure:

Demo Program: aliases_use_2.py

- This reveals that `i` is not a copy of each value in the sequence, but an alias to it. If you hadn't known about aliases and how a new assignment breaks an alias, I bet you would be puzzled by these outputs!
- Can you explain to me why in the previous example where `i` was an `<int>`, the `v_list` was not modified?

```
v_list = [[1], [2], [3]]
```

```
for i in v_list:  
    i.append(0)  
    print(i)
```

```
print(v_list)
```

Note: i is pointing to a new list which is mutable

```
Run alias_use_2  
C:\Python\Python36\python.exe  
[1, 0]  
[2, 0]  
[3, 0]  
[[1, 0], [2, 0], [3, 0]]
```

alias_copy_2.py

Python Objects as Expandable Record

LECTURE 2



A Useful Data Type

Similar to “struct” type in C or C++

- Binds together attributes

```
class Employee: # primitive class as record creator
    pass
john = Employee() # Create an empty employee record
# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```








Demo Program: emptyclass2.py

Go PyCharm!!!

```
class Employee: # primitive class as record creator
    pass

john = Employee() # Create an empty employee record
# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
print("Name=%s, Department=%s, Salary:%d" % (john.name, john.dept, john.salary))
```

Run  emptyclass2



C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course,
Name=John Doe, Department=computer lab, Salary:1000

Method Object

LECTURE 2



Method Objects

Method objects are objects that have been applied to a function, then stored as an object

```
xf = x.f
while True:
    print(xf())
```

The object is passed as the first argument of the function when there are no parameters

So in this case, `x.f()` is equivalent to `MyClass.f(x)`

```
class A:
    def f(self):
        print("f functional call")

print("Instance call")
x = A()
x.f()
print()
# use method as variable, ceating functional alias
print("Functional alias call")
Xf = x.f
Xf()
print()
# calling instance method in class method way
print("Class method call")
A.f(x)
print()
```

methodcalls.py

Importing and Modules

LECTURE 2



Importing and Modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*
- Three formats of the command:

```
import somefile
```

```
from somefile import *
```

```
from somefile import className
```

- The difference? What gets imported from the file and what name refers to it after importing



import ...

module imported but not treated as the same namespace as global

```
import somefile
```

Everything in somefile.py gets imported.

To refer to something in the file, append the text “somefile.” to the front of its name:

```
somefile.className.method("abc")
```

```
somefile.myFunction(34)
```



from ... import *

```
from somefile import *
```

Everything in somefile.py gets imported

To refer to anything in the module, just use its name. **Everything in the module is now in the current namespace.**

Take care! Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")
```

```
myFunction(34)
```



from ... import ...

```
from somefile import className
```

Only the item *className* in somefile.py gets imported.

After importing *className*, you can just use it without a module prefix. **It's brought into the current namespace.**

Take care! Overwrites the definition of this name if already defined in the current namespace!

```
className.method("abc") ← imported
```

```
myFunction(34) ← Not imported
```



Directories for module files

Where does Python look for module files?

The list of directories where Python will look for the files to be imported is `sys.path`

This is just a variable named 'path' stored inside the 'sys' module

```
>>> import sys
```

```
>>> sys.path
```

```
['', '/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/setuptools-0.6c5-py2.5.egg', ...]
```

- To add a directory of your own to this list, append it to this list

```
sys.path.append( '/my/new/path' )
```




from ... import Class as NewNamespace

```
from somefile import className as NewNamespace
```

Only the item *className* in somefile.py gets imported.

After importing *className*, a new **namespace named** NewNamespace is created. All access to this namespace must use NewNamespace.object.

```
NewNamespace.function()
```

UML (Modeling of The Relationship Among Classes)

LECTURE 2



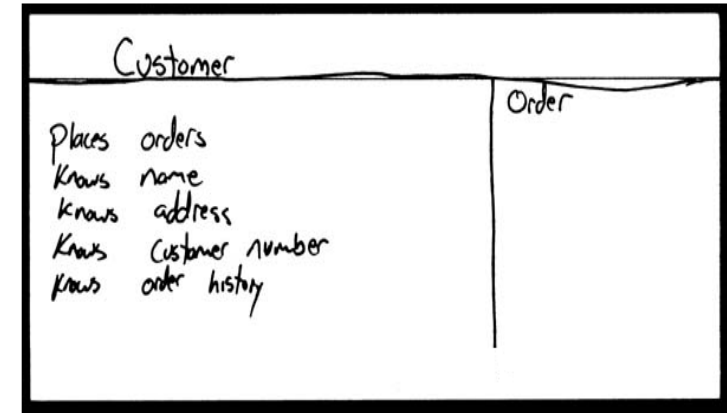
How do we design classes?

class identification from project spec / requirements

- nouns are potential classes, objects, fields
- verbs are potential methods or responsibilities of a class

CRC card exercises

- write down classes' names on index cards
- next to each class, list the following:
 - **responsibilities:** problems to be solved; short verb phrases
 - **collaborators:** other classes that are sent messages by this class (asymmetric)



UML

- class diagrams (today)
- sequence diagrams
- ...



Introduction to UML

Unified Modeling Language (UML): depicts an OO system

- programming languages are not abstract enough for OO design
- UML is an open standard; lots of companies use it
 - many programmers either know UML or a "UML-like" variant

UML is ...

- a *descriptive* language: rigid formal syntax (like programming)
- a *prescriptive* language: shaped by usage and convention
- UML has a rigid syntax, but some don't follow it religiously
- it's okay to omit things from UML diagrams if they aren't needed by team/supervisor/instructor



Diagram of one class

class name in top of box

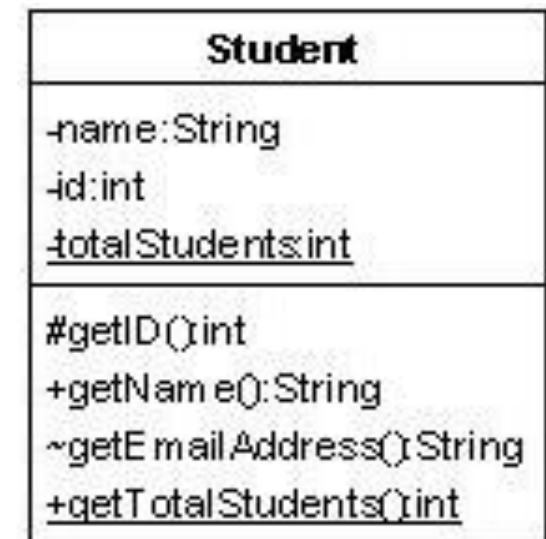
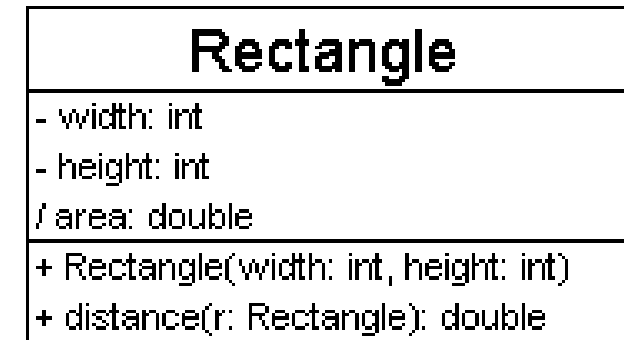
- ~~write <<interface>> on top of interfaces' names~~
- use *italics* for an *abstract class* name

attributes

- should include all fields of the object
- also includes derived "properties"

operations / methods

- may omit trivial (get/set) methods
 - but don't omit any methods from an interface!
- should not include inherited methods





Class attributes

Attributes (fields, instance variables)

- ~~visibility name : type [count] = defaultValue~~
- ~~visibility: + public~~
~~# protected~~
~~- private~~
~~~ package (default)~~  
~~/ derived~~
- underline static attributes
- **derived attribute**: not stored, but can be computed from other attribute values
- attribute example:
  - balance : double = 0.00

| Rectangle                                                                |
|--------------------------------------------------------------------------|
| - width: int<br>- height: int<br>/ area: double                          |
| + Rectangle(width: int, height: int)<br>+ distance(r: Rectangle): double |

| Student                                                                                      |
|----------------------------------------------------------------------------------------------|
| -name:String<br>-id:int<br><u>-totalStudents:int</u>                                         |
| #getID()int<br>+getName():String<br>~getEmailAdress()String<br><u>+getTotalStudents()int</u> |



# Class operations / methods

operations / methods

- ~~visibility~~ ~~name~~ (~~parameters~~): ~~returnType~~
- ~~underline~~ static methods
- parameter types listed as (name: type)
- omit *returnType* on constructors and when return is `void`
- method example:  
+ distance(p1: Point, p2: Point): double

| Rectangle                                                                |
|--------------------------------------------------------------------------|
| - width: int<br>- height: int<br>/ area: double                          |
| + Rectangle(width: int, height: int)<br>+ distance(r: Rectangle): double |

| Student                                                                                         |
|-------------------------------------------------------------------------------------------------|
| -name:String<br>-id:int<br><u>-totalStudents:int</u>                                            |
| #getID():int<br>+getName():String<br>~getEmailAdress():String<br><u>+getTotalStudents():int</u> |

# IS\_A Relationship

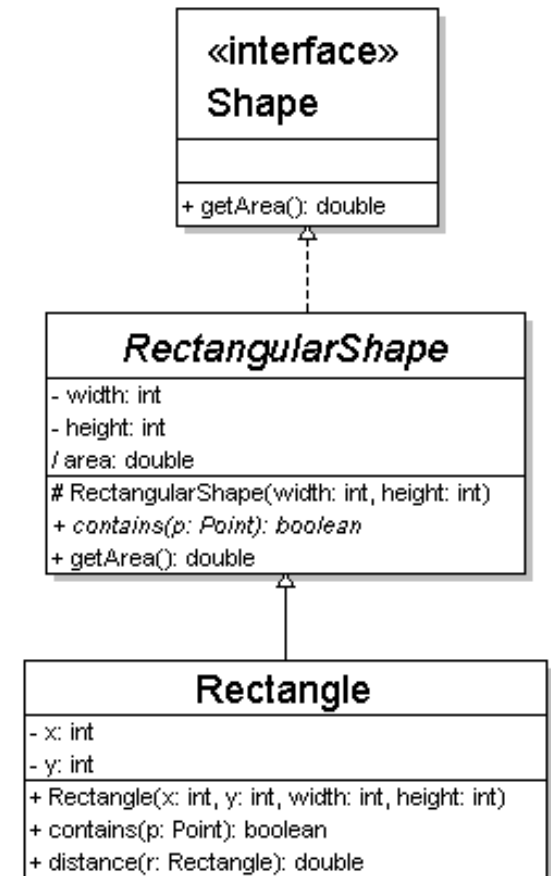
LECTURE 2





# Inheritance relationships

- hierarchies drawn top-down with arrows pointing upward to parent
- line/arrow styles differ based on parent:
  - *class* : solid, black arrow
  - *abstract class* : solid, white arrow
  - *interface* : dashed, white arrow
- we often don't draw trivial / obvious relationships, such as drawing the class `object` as a parent



# Has\_A Relationship

LECTURE 2



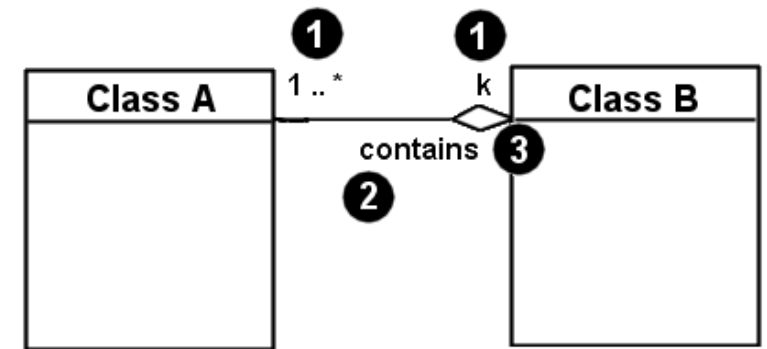
# Associational relationships

1. multiplicity (how many are used)

- \*  $\Rightarrow$  0, 1, or more
- 1  $\Rightarrow$  1 exactly
- 2..4  $\Rightarrow$  between 2 and 4, inclusive
- 3..\*  $\Rightarrow$  3 or more

2. name (what relationship the objects have)

3. navigability (direction)





# Multiplicity

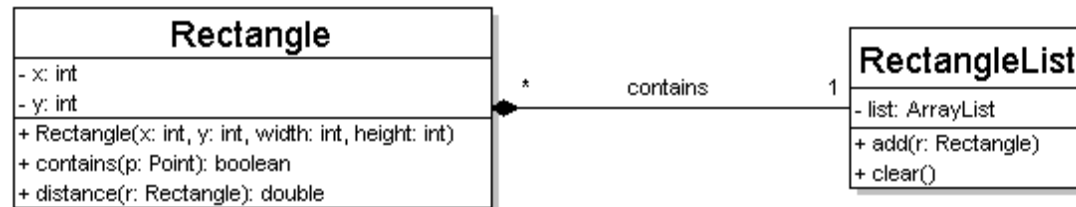
## one-to-one

- each student must have exactly one ID card



## one-to-many

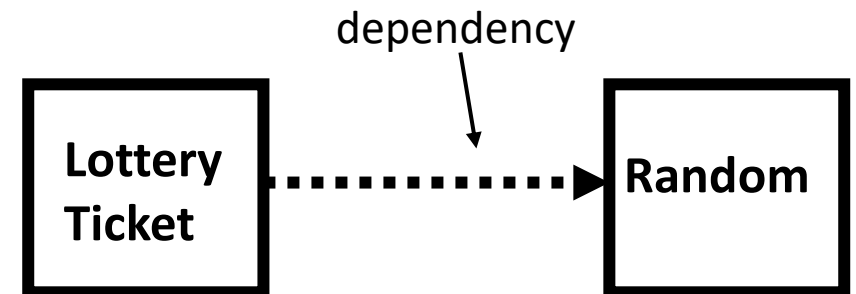
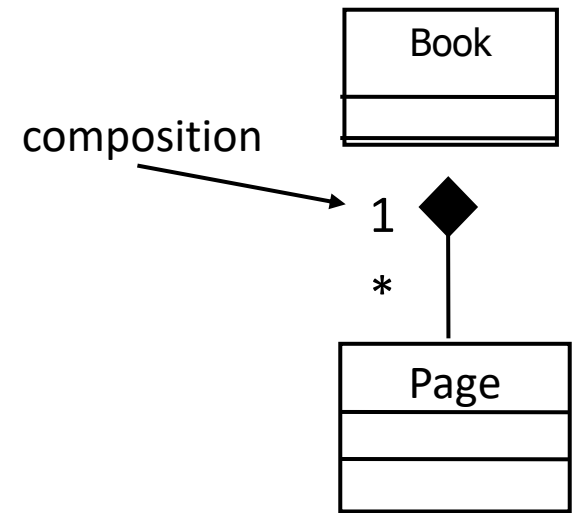
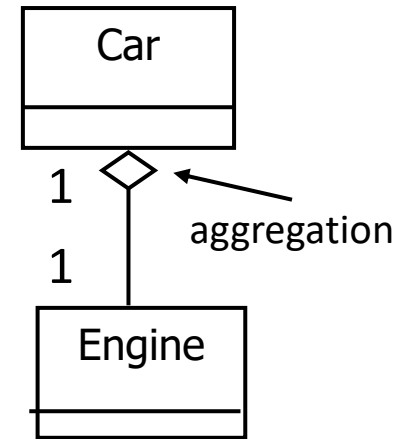
- a RectangleList can contain 0, 1, 2, ... rectangles





# Association types

- **aggregation**: "is part of"
  - clear white diamond
- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - black diamond
- **dependency**: "uses temporarily"
  - dotted line or arrow
  - often is an implementation detail, not an intrinsic part of that object's state





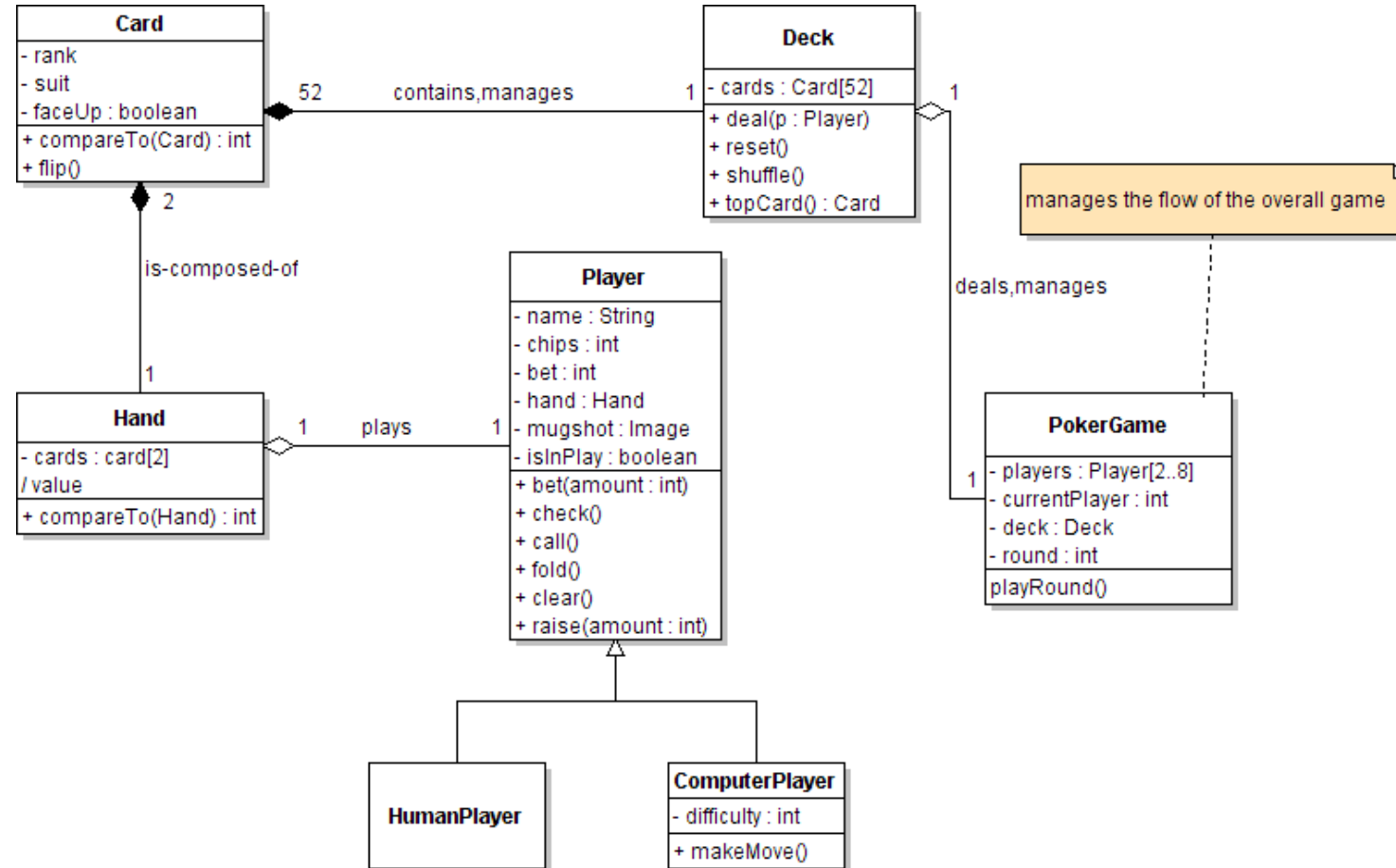
# Class design exercise

---

- Consider this Texas Hold 'em poker game system:
  - 2 to 8 human or computer players
  - Each player has a name and stack of chips
  - Computer players have a difficulty setting: easy, medium, hard
  - Summary of each hand:
    - Dealer collects ante from appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck.
    - A betting round occurs, followed by dealing 3 shared cards from the deck.
    - As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise.
    - At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet.
  - What classes are in this system? What are their responsibilities? Which classes collaborate?
  - Draw a class diagram for this system. Include relationships between classes (generalization and associational).



# Poker class diagram





# Class diag. pros/cons

---

Class diagrams are great for:

- discovering related data and attributes
- getting a quick picture of the important entities in a system
- seeing whether you have too few/many classes
- seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
- spotting dependencies between one class/object and another

Not so great for:

- discovering algorithmic (not data-driven) behavior
- finding the flow of steps for objects to solve a given problem
- understanding the app's overall control flow (event-driven? web-based? sequential? etc.)



# Quality of Classes (Modules)

LECTURE 2



# Qualities of Modules

---

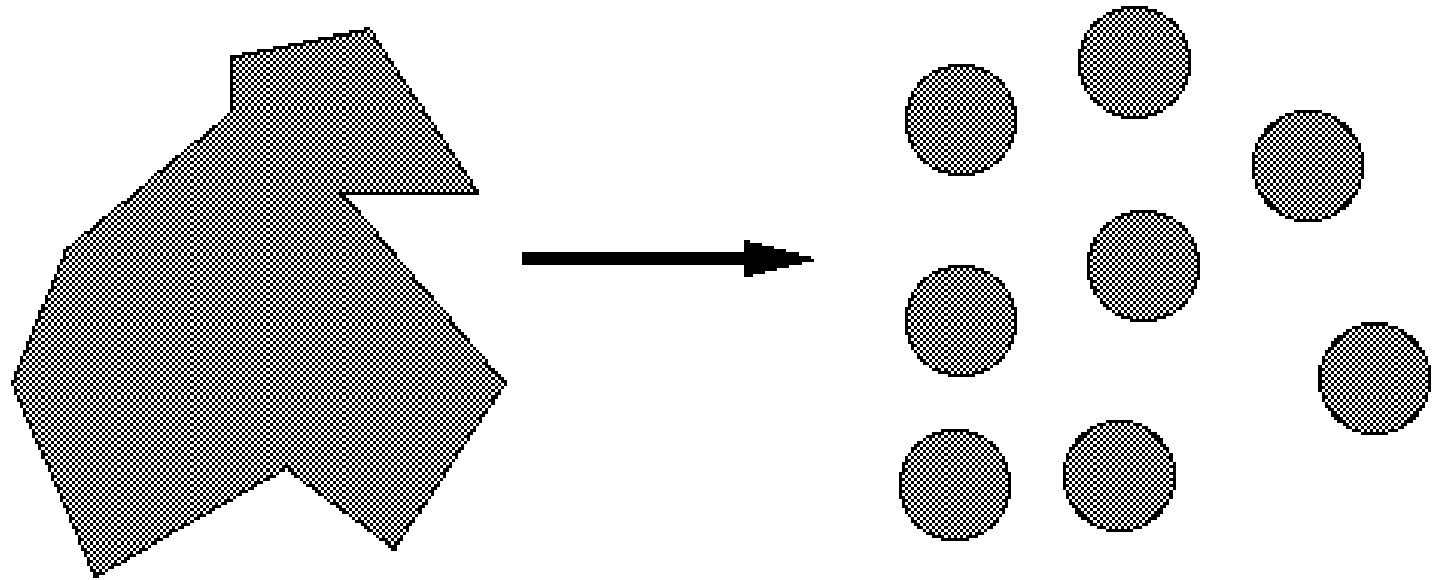
These quality of modules decides the following thing:

- Re-usability.
- Coherence (Single Goal for the Class)
- Low Coupling
- Software Scalability
- Readability
- Tractability

# Qualities of modular software

## Decomposable

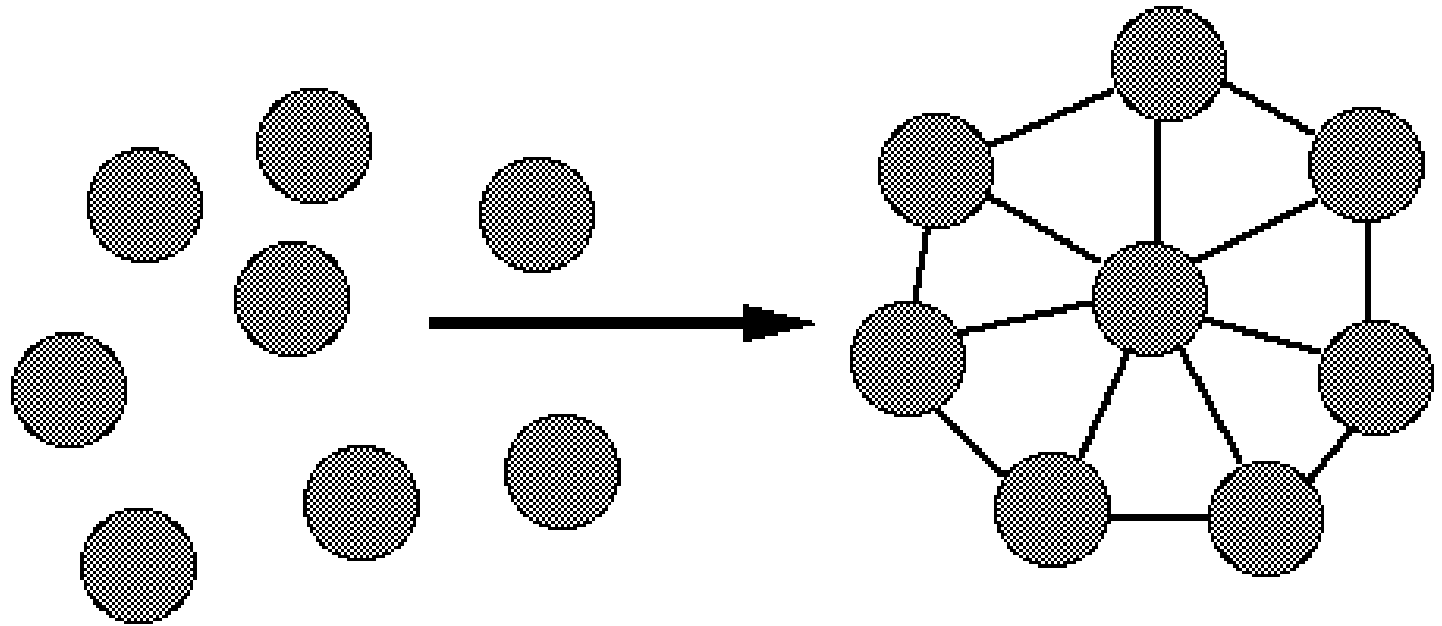
- can be broken down into pieces



# Qualities of modular software

## Composable

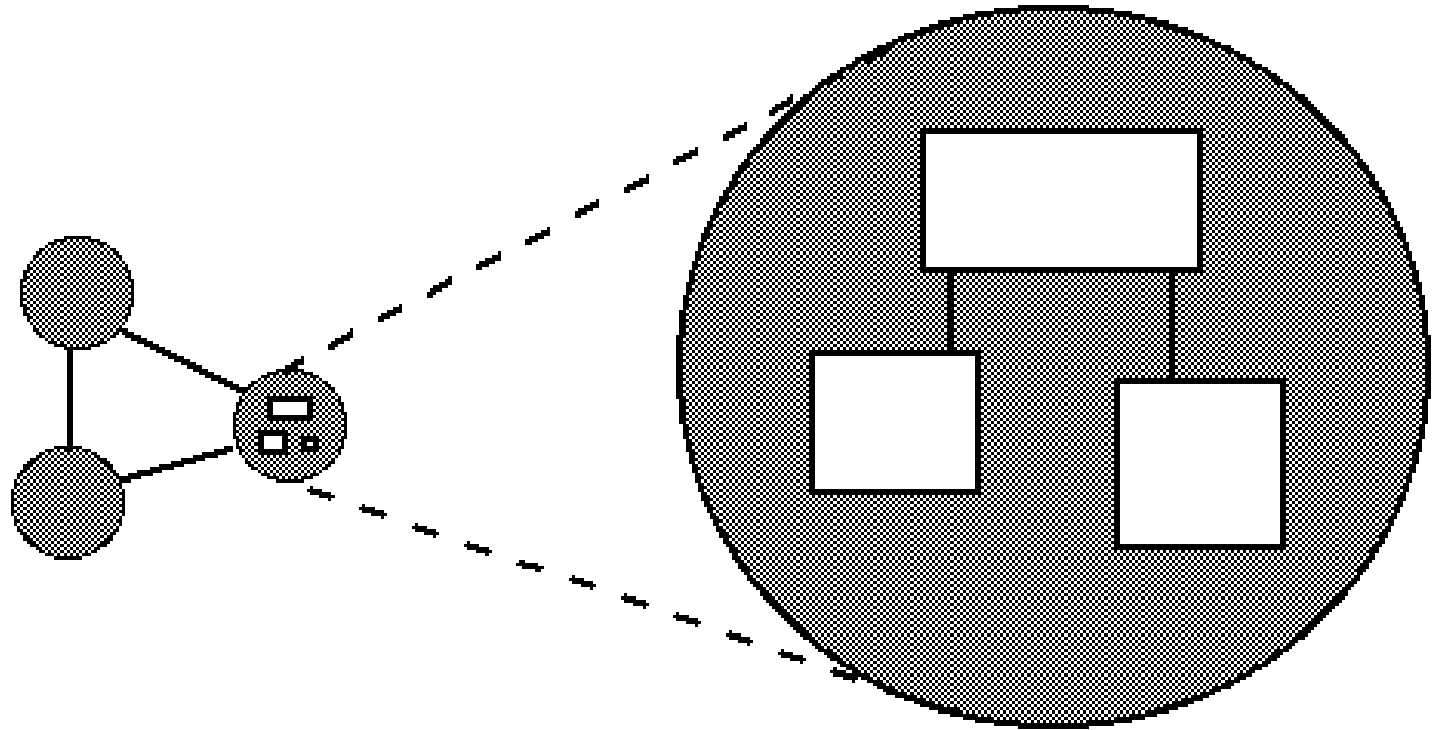
- Pieces are useful and can be combined



# Qualities of modular software

## Understandable

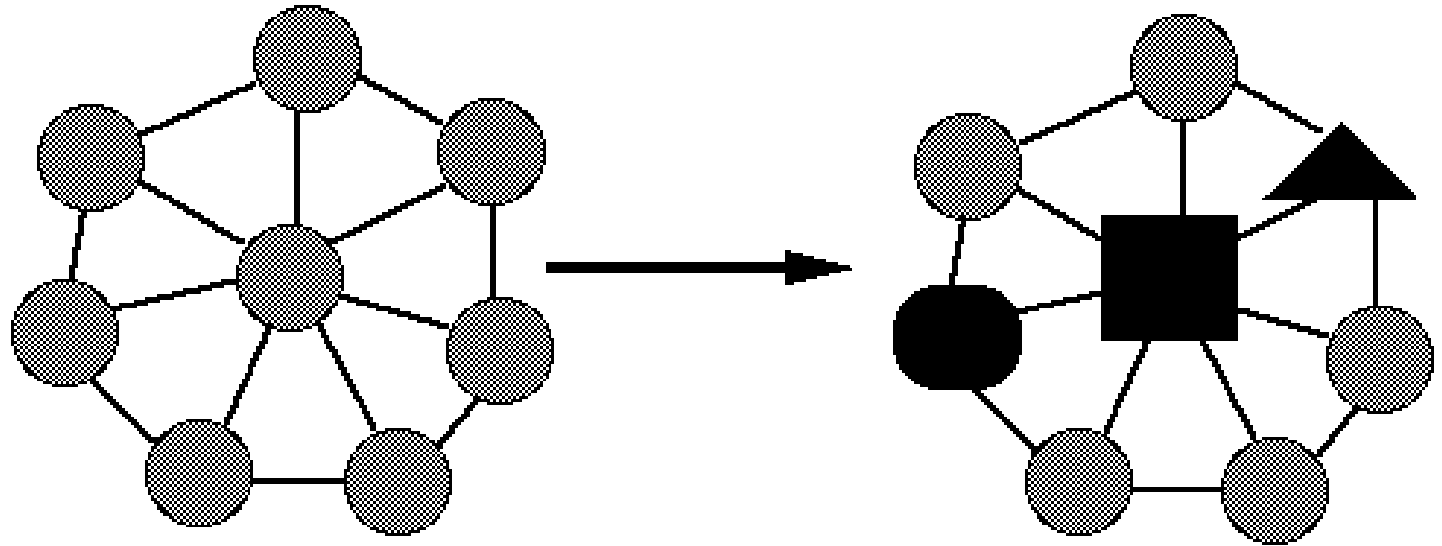
- One piece can be examined in isolation



# Qualities of modular software

## Has Continuity

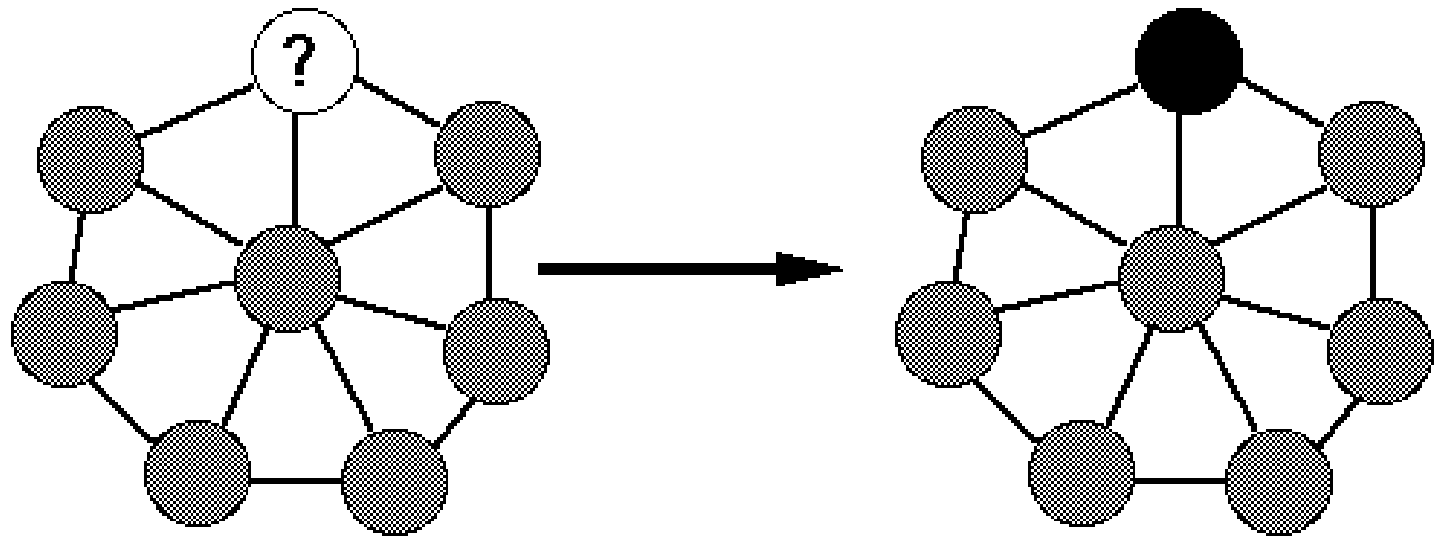
- Reqs. change affects few modules
- Improve Tractability



# Qualities of modular software

## Protected / safe

- An error affects few other modules.
- Low coupling reduce the impacts.



# Coupling and Cohesion

LECTURE 2





# Cohesion and coupling

---

**cohesion:** how complete and related things are in a class  
*(a good thing)*

**coupling:** when classes connect to / depend on each other  
*(too much can be a bad thing)*

- **Heuristic 2.7:** Classes should only exhibit nil or export coupling with other classes; that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
- (in other words, minimize unnecessary coupling)



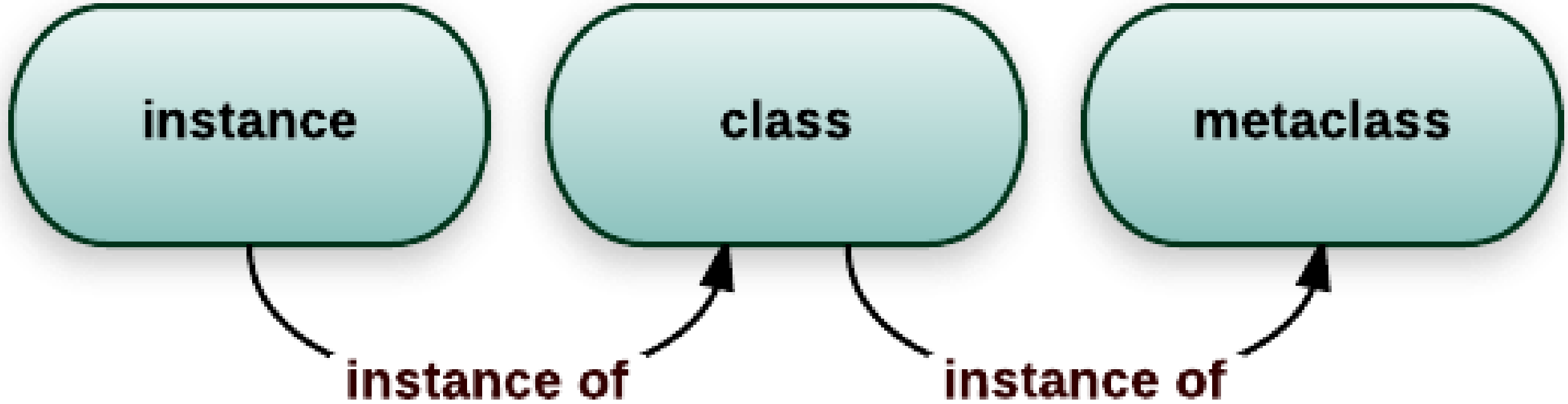
# Reducing coupling

---

- combine 2 classes if they don't represent a whole abstraction
  - example: `Bet` and `PlayRound`
- make a coupled class an inner class
  - example: `list` and `list iterator`; `binary tree` and `tree node`
  - example: `GUI window frame` and `event listener`
- provide simpler communication between subsystems
  - example: provide methods (`newGame`, `reset`, ...) in `PokerGame` so that clients do not need to manually refresh the players, bets, etc.

# Python Object Model (Optional)

LECTURE 1



## Three Instantiation Level (Meta -> Class -> Instance)

Java has two only two level. The meta-class is a template for Class. It is a generic programming mechanism.



# Meta-Class

---

- In object-oriented programming, a **metaclass** is a class whose instances are classes.
- Just as an ordinary class defines the behavior of certain objects, a **metaclass** defines the behavior of certain classes and their instances.
- Not all object-oriented programming languages support **metaclasses**.
- Among those that do, the extent to which **metaclasses** can override any given aspect of class behavior varies.
- **Metaclasses** can be implemented by having classes be first-class citizen, in which case a **metaclass** is simply an object that constructs classes.
- Each language has its own metaobject protocol, a set of rules that govern how objects, classes, and **metaclasses** interact.

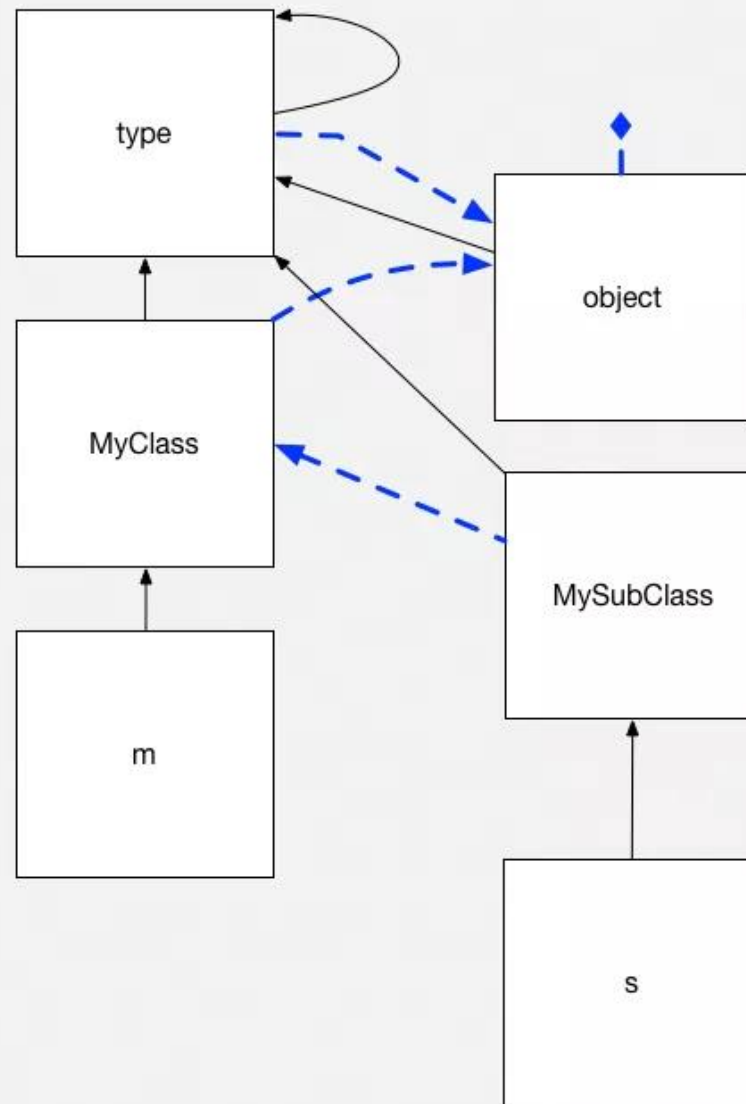


# Support in languages and tools

---

The following are some of the most prominent programming languages that support metaclasses.

- Common Lisp, via CLOS
- Delphi and other versions of Object Pascal influenced by it
- Groovy
- Objective-C
- **Python**
- **Perl, via the metaclass pragma, as well as Moose**
- **Ruby**
- Smalltalk



# Meta-Class (Optional)

LECTURE 1





# The fundamental types of Python

---

The aim of this post is to present a succinct diagram that correlates some basic properties of all Python objects with the fundamental types type and object.

In Python, every object has a type.

Types are also objects - rather special objects. A type object, like any other object, has a type of its own. It also has a sequence of "base types" - in other words, types from which it inherits.

This is unlike non-type objects, which don't have base types.

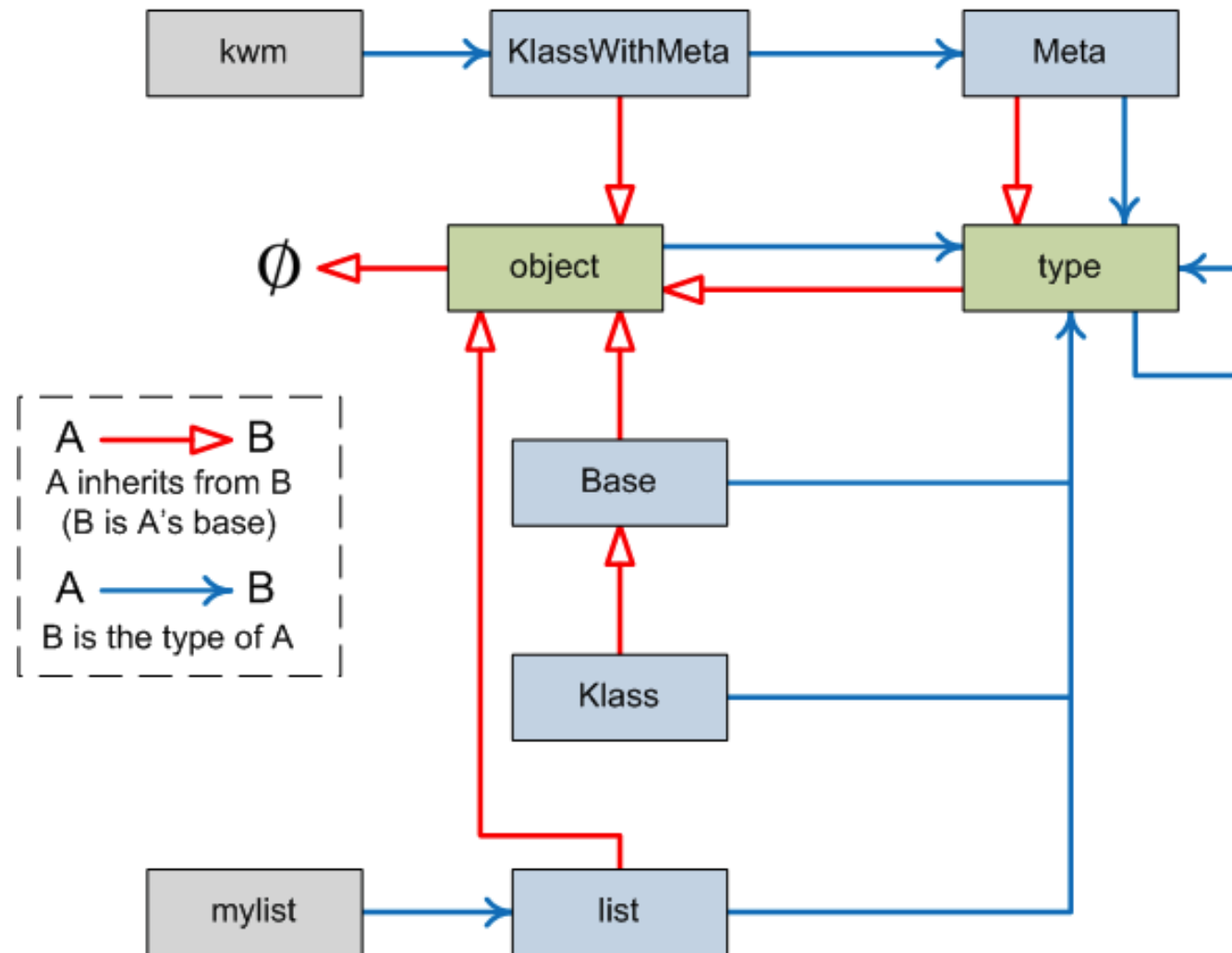
```
# Some types
class Base:
    pass

class Klass(Base):
    pass

class Meta(type):
    pass

class KlassWithMeta(metaclass=Meta):
    pass

# Non-types
kwm = KlassWithMeta()
mylist = []
```





# Type

---

Some interesting things to note:

- The default type of all new types is `type`. This can be overridden by explicitly specifying the **metaclass** for a type.
- Built-in types like `list` and user-defined types like `Base` are equivalent as far as Python is concerned.
- The special type **`type`** is the default type of all objects - including itself. It is an object, and as such, inherits from `object`.
- The special type `object` is the pinnacle of every inheritance hierarchy - it's the ultimate base type of all Python types.
- `type` and `object` are the only types in Python that really stand out from other types (and hence they are colored differently). `type` is its own type. `object` has no base type.