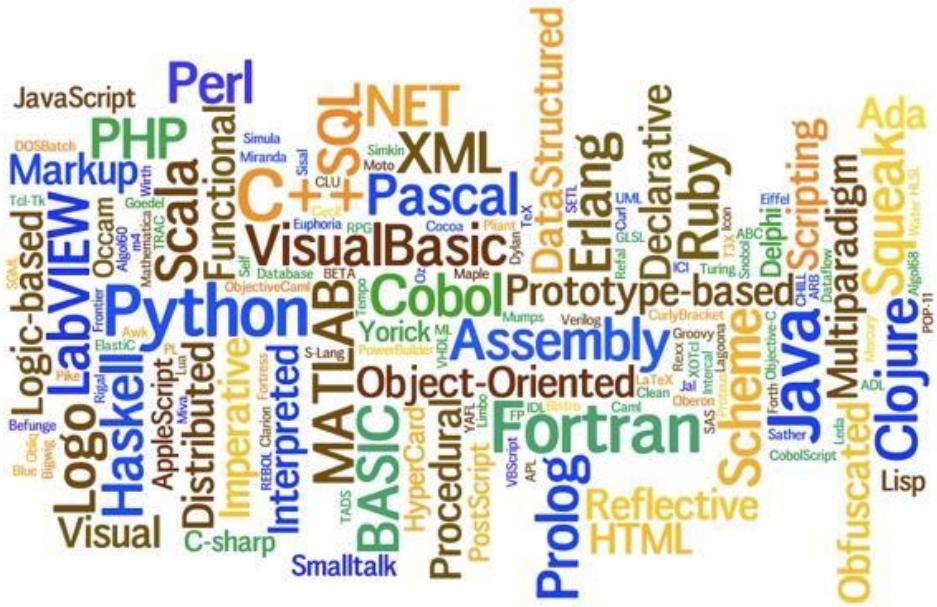



# Object Oriented Programming

## Support for OOP in Python

**Programming Languages**

**Dr. Tamer ABUHMED**

**College of Computing**

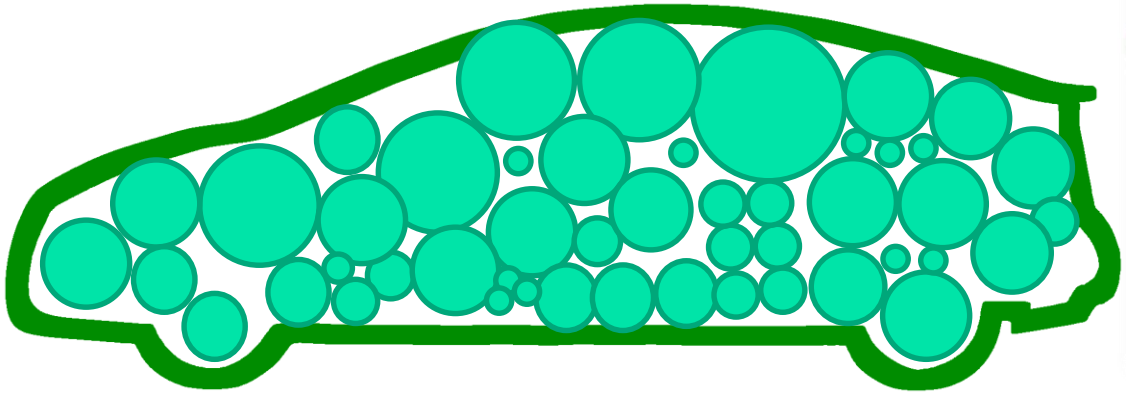
SUNG KYUN KWAN UNIVERSITY

# Outline

- Object-Oriented Programming Concepts
- Why Object Oriented Programming?
- Design Principles of OOP
- Python Support of OOP

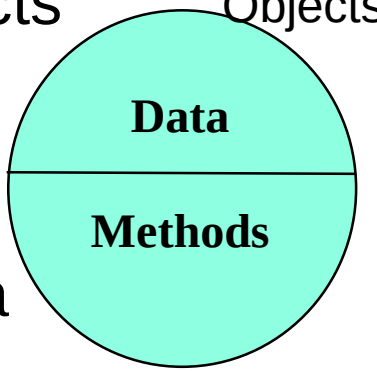
# Object-Oriented Programming Concepts
# Objects and Methods

- Java is an object-oriented programming (OOP) language
  - Programming methodology that views a program as consisting of objects that interact with one another by means of actions (called **methods**)
  - Objects of the same kind are said to have the same type or be in the same class



# Object-Oriented Programming: OOP

- A design and programming technique
- Some terminology:
  - **Object** - usually a person, place or thing (a noun)
  - **Method** - an action performed by an object (a verb)
  - **Type or Class** - a category of similar objects (such as automobiles)
- Objects have both data and methods
- Objects of the same class have the same data elements and methods
- Objects send and receive messages to invoke actions



# Example of an Object Class

# Example of an Object Class

## Class: Automobile

Data Items:

- manufacturer's name
- model name
- year made
- color
- number of doors
- Body shape
- size of engine
- etc.

Methods: (action)

- Define data items (specify manufacturer's name, model, year, etc.)
- Change a data item (color, engine, etc.)
- Display data items
- Calculate cost
- Assemble
- etc.

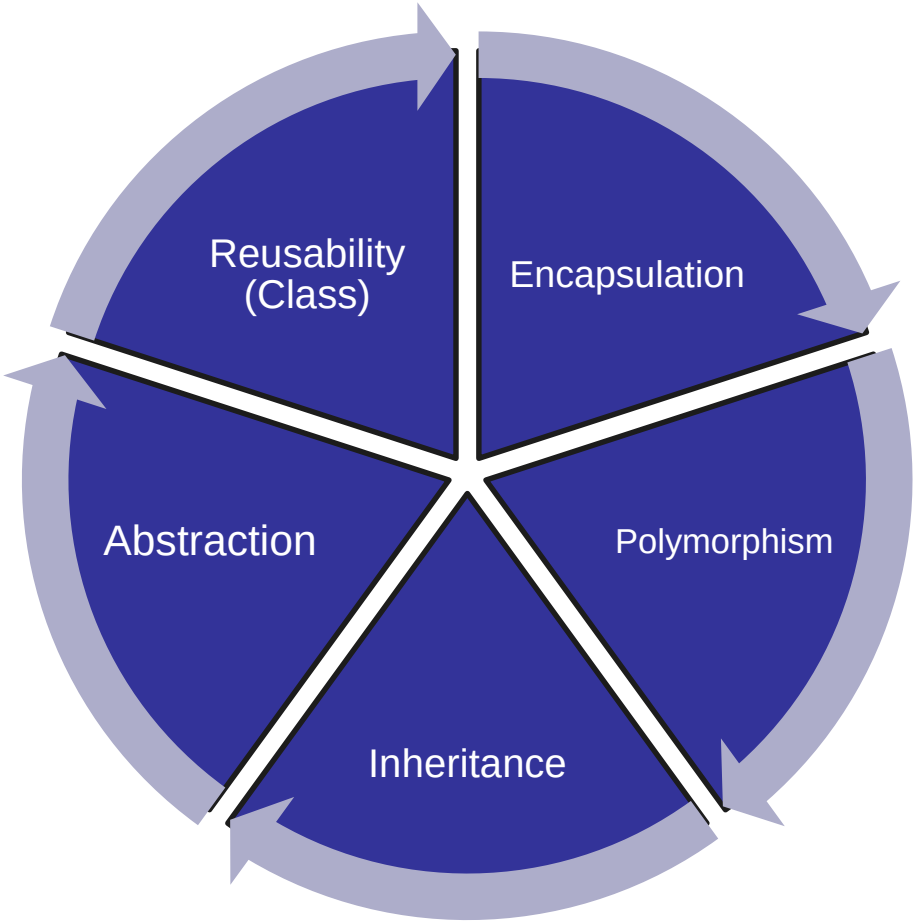
# Why OOP?

- Easy to design software as building blocks.
- Save development time (and cost) by reusing code
  - once a class is created, it can be used in other applications
- Easier debugging
  - classes can be tested independen
  - reused objects have already been

# Design Principles of OOP

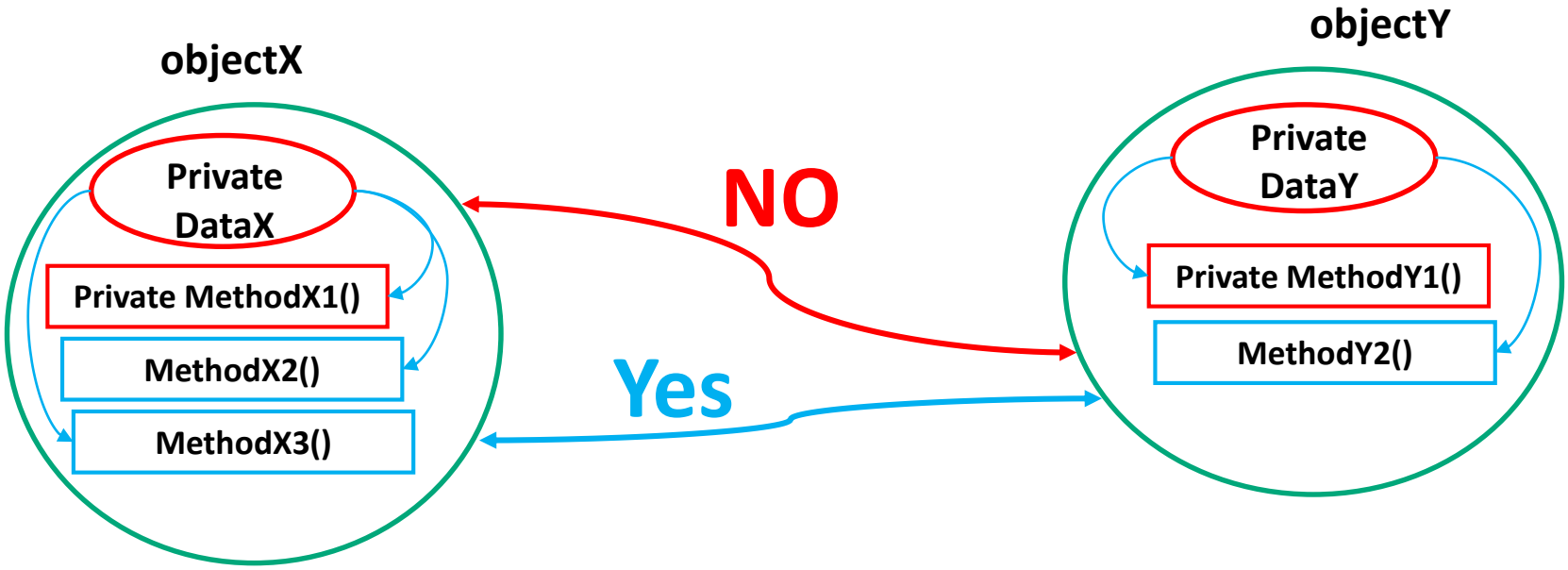
Three main design principles of Object-Oriented Programming (OOP):

- Encapsulation
- Polymorphism
- Inheritance



# OOP: Encapsulation

- Design software
  - can be easily used
  - without knowing the details of how it works.
- Also known as *information hiding*



# OOP: Reusable Components

Advantages of using reusable components:

- saves time and money
- components that have been used before
  - often better tested and more reliable than new software

Make your classes reusable:

- encapsulation
- general classes have a better chance of being reused than ad hoc classes

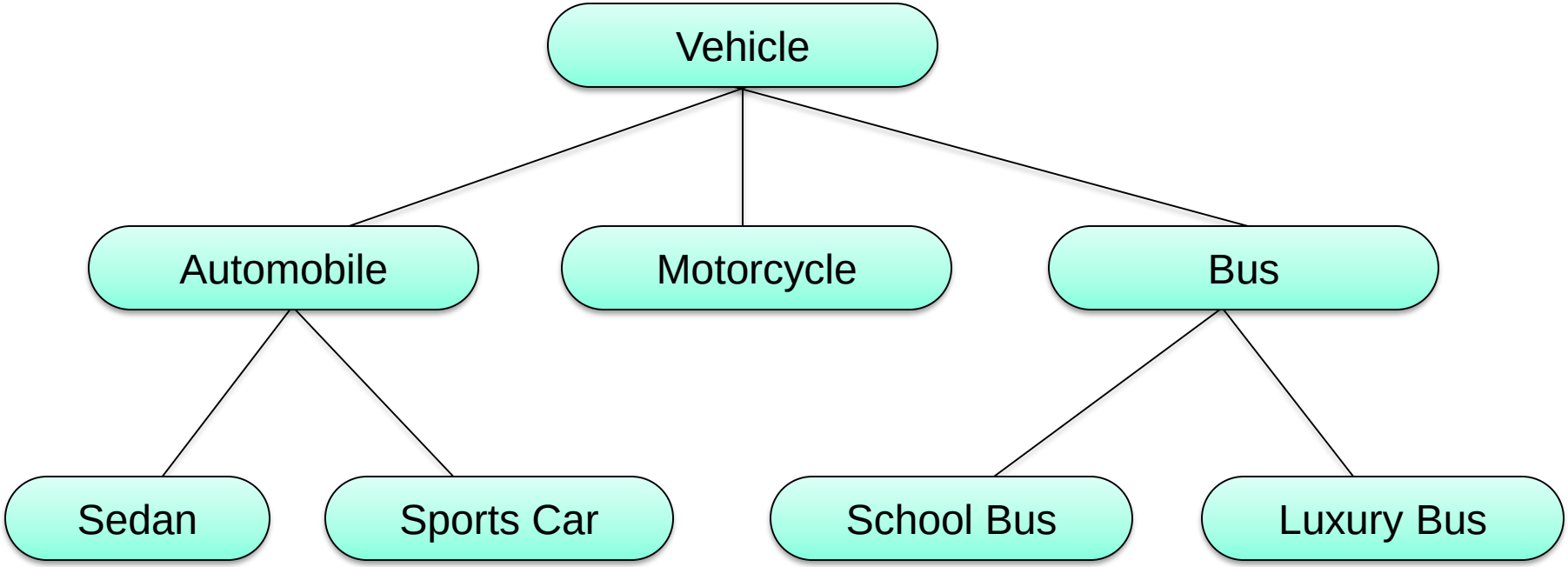
# OOP: Polymorphism

- Polymorphism—the same word or phrase can be mean different things in different contexts
- Analogy: in English, **bank** can mean:
  - side of a river or
  - a place to put money
- In Java, two or more classes could each have a method called **output**
- Each **output** method would do the “right thing” for the class that it was in.  E.g.
  - display a number (Integer class)
  - display an image (Photo class)

# OOP: Inheritance

- Inheritance—a way of organizing classes
- Term comes from inheritance of traits like eye color, hair color, and so on.
- Classes with attributes in common can be grouped so that their common attributes are only defined once.

# An Inheritance Hierarchy




What properties does each vehicle inherit from the types of vehicles above it in the diagram?

# Python
# Object Oriented Programming

# Python/Java/C++

Like C++ in

- Multiple inheritance
- Operator overloading
- Signature-based polymorphism
  - as if “everything was a template”

Like Java in

- everything inherits from "object "
- uniform object-reference semantics
  - assignment, argument passing, return
- garbage collection

# Defining a Class

Syntax:

```
class name[(base_class)]:
body
```

Create a class with default superclass

```
# old style
class name:
    body
```

```
# new style
class name(object):
    body
```

```
class MyClass(object):
    myvar = 30

>>> MyClass.myvar
30
```

# Defining Constructor and Methods

- All instance methods must must have explicit object reference (`self`) as the first parameter

**`self`** is the conventional name (like Java "this")

```
class Rectangle(object):
      def __init__(self, width, height):
         self.width = width
         self.height = height

      def area(self):
          return self.width * self.height
```

constructor

```
>>> r = Rectangle(10, 20)
>>> Rectangle.area(r)
200
>>> r.area()
200
```

No "new".

# "Implicit" Parameter (`self`)

- Java: `this`, implicit

```
public void translate(int dx, int dy) {
    x += dx;        // this.x += dx;
    y += dy;        // this.y += dy;
}
```

- Python: `self`, explicit

```
def translate(self, dx, dy):
    self.x += dx
    self.y += dy
```

# Weak Encapsulation

Everything is public.

Instance methods can be used in static way.

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print("Hello "+self.name)

>>> p = Person("Me")
>>> p.name                  # attributes are public
>>> p.greet()               # O-O style method call
>>> Person.greet(p)         # imperative style
```

# Hiding by name mangling

Any attribute name that begins with "_ _" is mangled to "_*Class*_ _name".

```
class Person:
     def __init__(self, name):
          self.__name = name
          # Mangled to _Person__name
     @property
     def name(self):
          return self.__name   #auto demangle
```

```
>>> p = Person("Lone Ranger")
>>> p.name                    # returns the name
Lone Ranger
>>> p.name = "Zoro"
builtins.AttributeError: can't set attribute
```

# Only One Constructor

Python classes can only have one **`__init__`**.

You can overload it using params with default values.

```
class Point:
    '''Point with x,y coordinates'''
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

>>> p = Point(3,4)
>>> q = Point( )          # same as Point(0,0)
```

# Static Methods

To define a static method, prefix with `@staticmethod` and no `self` parameter.

```
class Point:

    '''create a point using polar coordinates.'''
    @staticmethod
    def makePoint(r,a):
        return Point(r*math.cos(a), r*math.sin(a))

>>> p = Point.makePoint( 10, math.pi/4 )
```

Static *factory methods* can help overcome limitation of only one constructor.

# Properties

Properties are like synthetic attributes. Used like "get" properties in C#. Only parameter is **`self`**.

```
class Point:
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y
    @property
    def length(self):
        return math.hypot(self.x, self.y)

>>> p = Point(3,4)
>>> p.length
5
```

# "Set" properties

Can use `@property` and `@x.setter` to protect attributes.

```
class Person:
    def __init__(self,name):
        self.__name = name  # hide the name
    @property
    def name(self):         # name accessor
        return self.__name
    @name.setter
    def name(self,value):  # set the name
        if not isinstnace(value, str):
                raise TypeError("Name must be str")
        self.__name = value
```

# Restricting Keys in the Object dict

- You can restrict the allowed attribute names.
- Use a tuple named `__slots__`

```
class Point:
    __slots__ = ('x','y','name')
    f.name = "weak, weak"
    def __init__(x=0,y=0):
        self.x, self.y = x, y
```

```
>>> p = Point(2,3)
>>> p.junk = "ha ha"
builtins.AttributeError: 'Point' object has no attribute 'junk'
```

# Inheritance

Subclass must invoke parent's constructor explicitly

```
class Square(Rectangle):
    def __init__(self, width):
        Rectangle.__init__(self, width, width)
```

```
>>> s = Square(100)
>>> s.area()
10000
```

# accessing superclass

Python 3 has a super() method to access superclass.

```
class Square(Rectangle):
    def __init__(self, width):
        super().__init__( width, width)
```

```
>>> s = Square(100)
>>> s.area()
10000
```

# Polymorphism

- All methods are virtual

```
import math

class Circle(object):
def __init__(self, radius):
   self.radius = radius

def area(self):
   return math.pi*self.radius*self.radius
```

```
>>> shapes = [Square(5), Rect(2,8), Circle(3)]
>>> for x in shapes:
...     print (x.area())
25
16
28.2743338823
```

# Operator Overloading 1/2

- Objects can implement infix operators by defining specialy named methods, such as __add__
  - operators: `+`, `-`, `*`, `/`, `**`, `&`, `^`, `~`, `!=`

```
class Point:
    def __init__(self, x, y):
        this.x, this.y = x, y
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y);
```

```
>>> a = Point(1,2)
>>> b = Point(3,7)
>>> print(a + b)    # invokes Point.__add__(a,b)
(4, 9)              # assuming we also define Point.__str__
```

# Operator Overloading 2/2

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

| Operator | Class Method |
|---|---|
| - | `__neg__(self, other)` |
| + | `__pos__(self, other)` |
| * | `__mul__(self, other)` |
| / | `__truediv__(self, other)` |

Unary Operators

| | |
|---|---|
| - | `__neg__(self)` |
| + | `__pos__(self)` |

| Operator | Class Method |
|---|---|
| == | `__eq__(self, other)` |
| != | `__ne__(self, other)` |
| < | `__lt__(self, other)` |
| > | `__gt__(self, other)` |
| <= | `__le__(self, other)` |
| >= | `__ge__(self, other)` |

# Python Object Hooks

- Objects can support built-in operators by implementing special methods
  - operators: `+`, `-`, `*`, `/`, `**`, `&`, `^`, `~`, `!=`
  - Indexing (like sequences): `obj[idx]`
  - Calling (like functions): `obj(args,...)`
  - Iteration and containment tests

```
for item in obj: ...
if item in obj: ...
```

# Object Data is Stored as a Dictionary

- Attributes are stored in a dictionary (`__dict__`).
- You can add attributes to existing object!

```
f = Fraction(2,3)
f.name = "weak, weak"
f.__dict__
{'num': 2, 'denom':3, 'name':'weak, weak'}
```

`f.name = value` invokes f.__setattr__("name",value)

`del f.name` invokes f.__delattr__("name")

# Testing object type

Get the type of a variable:

```
>>> p = Point( 1, 3)
>>> type(p)
<class '__main__.Point'>
```

Test the type using `isinstance`:

```
>>> p = Point( 1, 3)
>>> isinstance(p, Point)
True
>>> isinstance(p, float)
False
```

# Summary

- Object-Oriented Programming Concepts
- Why Object Oriented Programming?
- Design Principles of OOP
- Python Support of OOP