Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 6 – 17, 2020

Python for Data Analytics

Object-Oriented Programming

High-Level Programming Paradigms

- Control-oriented Programming (before mid 80's)
  - Real-world problem → a set of functions
  - Data and functions are separated treated
  - Fortran, Cobol, PL/1, Pascal, C (1972, Bell Lab.)

- Object-oriented Programming (after mid 80's)
  - Real-world problem → a set of classes
  - Data and functions are encapsulated inside classes
  - C++ (1983, Bell Lab.), Python (1991), Java (1993)
  - And most script languages (Ruby, PHP, R, ...)

Typical Control-oriented Programming

- Example C code for TV operations

```c
#include <stdio.h>

int power = 0;  // 0: off, 1: on
int channel = 1;
int caption = 0 // 0: off, 1: on

int main(void) {
    power();
    channel = 10;
    channelUp();
    printf("%d\n", channel);
    displayCaption("Hello, World");

    caption = 1;
    displayCaption("Hello, World");
}
```

```c
void power() {
    if (power)
        power = 0;      // power off -> on
    else
        power = 1;      // power on -> off
}

void channelUp() {    ++channel;    }

void channelDown() {  --channel;    }

void displayCaption(char *text) {
    if (caption)
        printf("%s\n", text);
}
```

Abstract Data Type

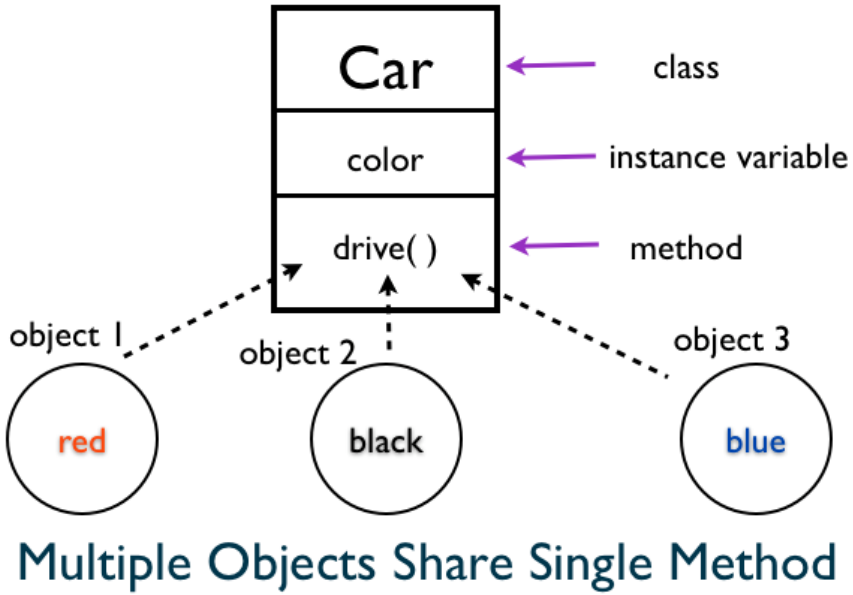
- An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects

- Abstraction is simply the removal of unnecessary detail
  - The idea is that to design a part of a complex system, you must identify what about that part others must know in order to design their parts, and what details you can hide
  - The part others must know is the abstraction

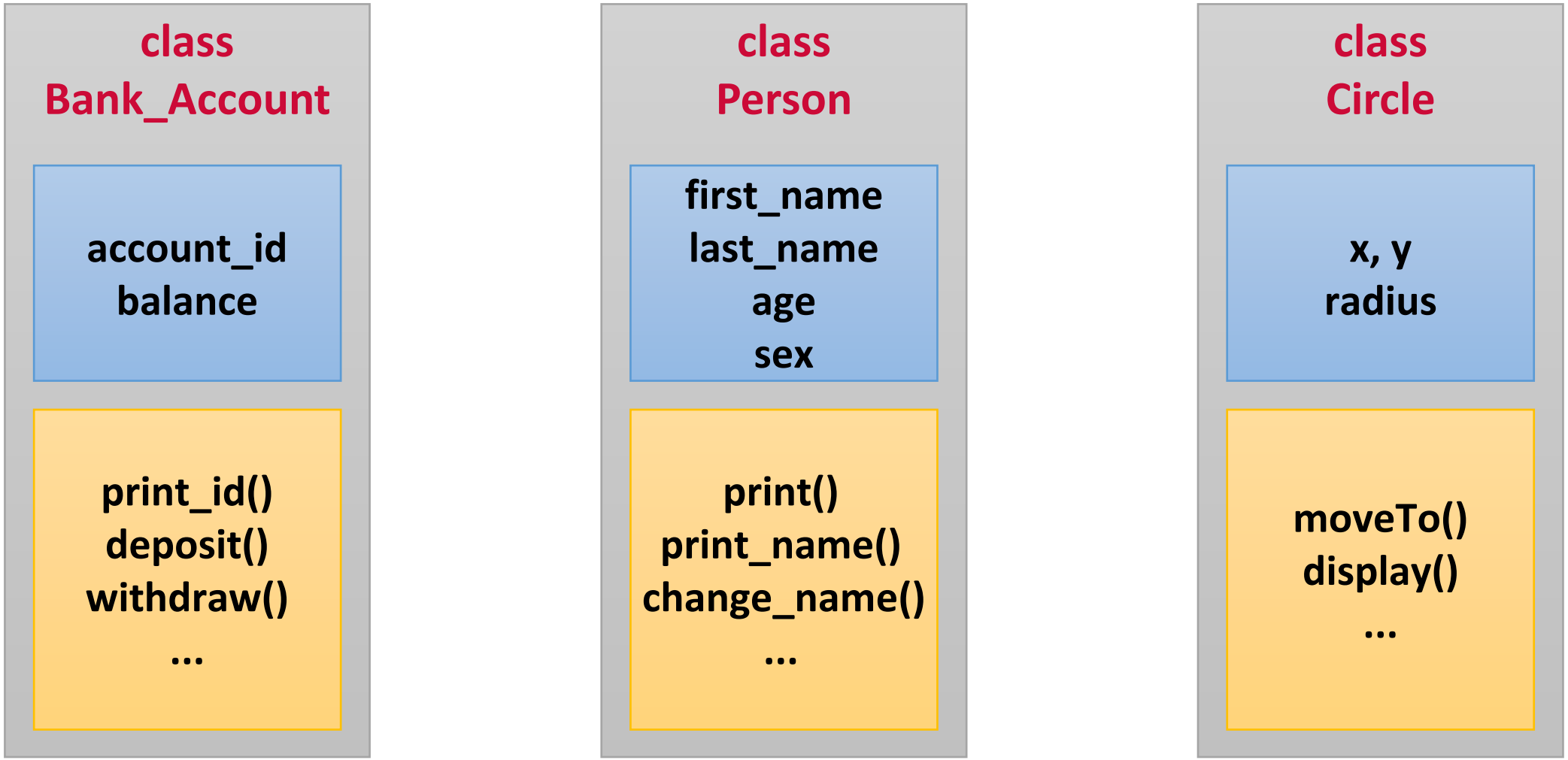
- Abstraction is information hiding

Definitions

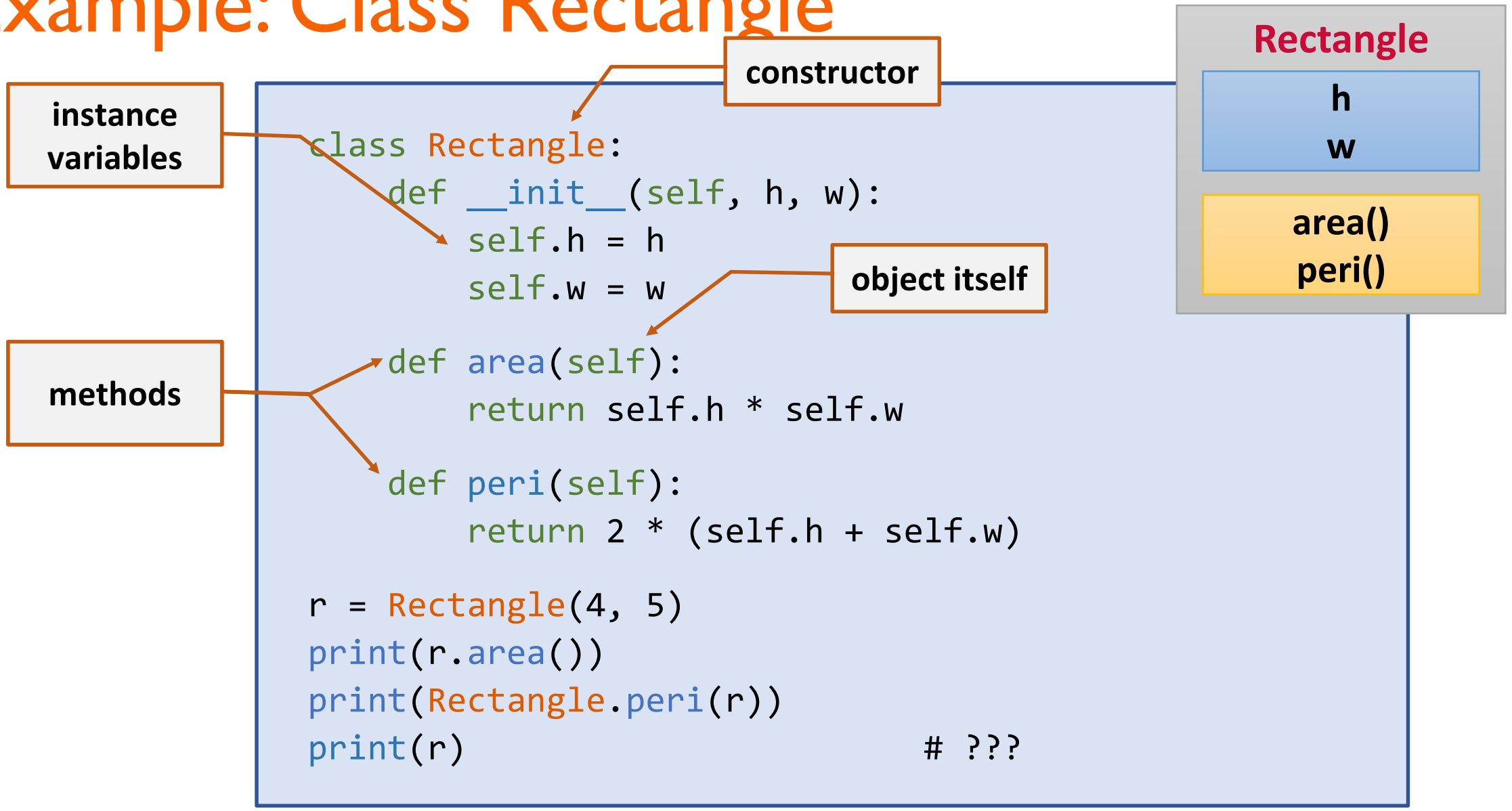
- Class
  - A template: data part (representation) + operation part (behavior)
- Operation part: Method or Action or Instance Method or Procedure or Function or Member Function or Operation
- Data part: Instance Variable or Attribute or Variable or Property or Data Member
- Object or Instance
  - A particular instance of a class



Class Examples



Example: Class Rectangle



Python Special Methods

- `__init__()`
  - Constructor, typically used for object initialization
- `__del__()`
  - Destructor, called when the object is destroyed by the Garbage Collector
- `__str__()`
  - The "informal" or nicely printable string representation of an object
- `__len__()`
  - The length of the object
- `__add__()`, `__sub__()`, `__eq__()`, `__lt()__`, `...`
  - Called to implement the binary arithmetic operations

Example: Class Rectangle (Revisited)

```
class Rectangle:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    def area(self):
        return self.h * self.w

    def peri(self):
        return 2 * (self.h + self.w)

    def __str__(self):
        return 'Rect(%d, %d)' %
            (self.h, self.w)
```

```
    def __eq__(self, a):
        return self.h == a.h and
            self.w == a.w

    def __add__(self, a):
        nh = self.h + a.h
        nw = self.w + a.w
        return Rectangle(nh, nw)

r1 = Rectangle(4, 5)
r2 = Rectangle(4, 5)
print(r1, r2)
print(r1 == r2)
print(r1 + r2)
print(r1 != r2)    # ???
```

dir()

- Shows the "capabilities" of the corresponding class

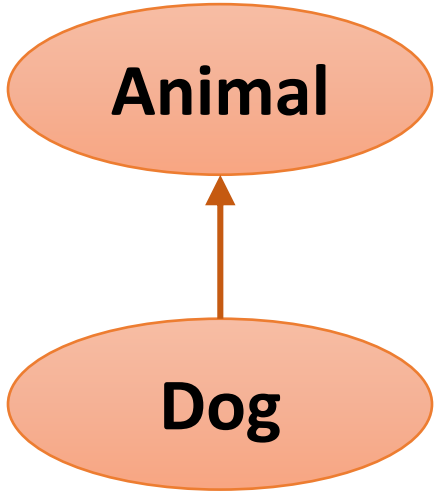
```
>>> dir(r1)
['__add__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'area', 'h', 'peri', 'w']

>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Class Inheritance

- A class (subclass) can be created from a preexisting class (superclass)
  - The child class inherits the instance variables and methods from its parent class
  - A child class can override instance variables and methods from the parent by defining its own members and methods with same name

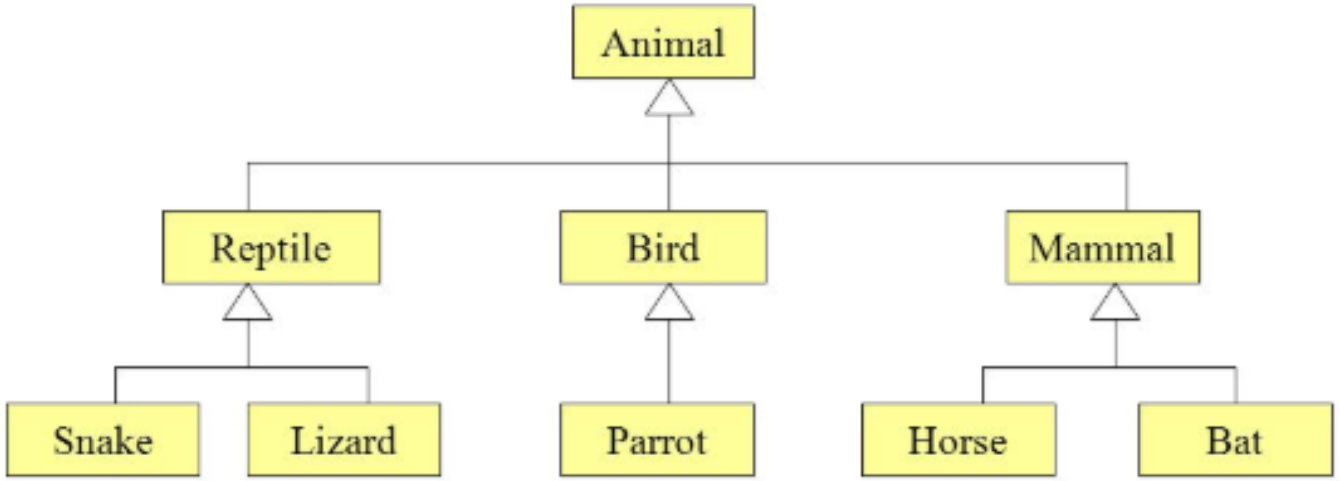
- Generalization vs. Specialization

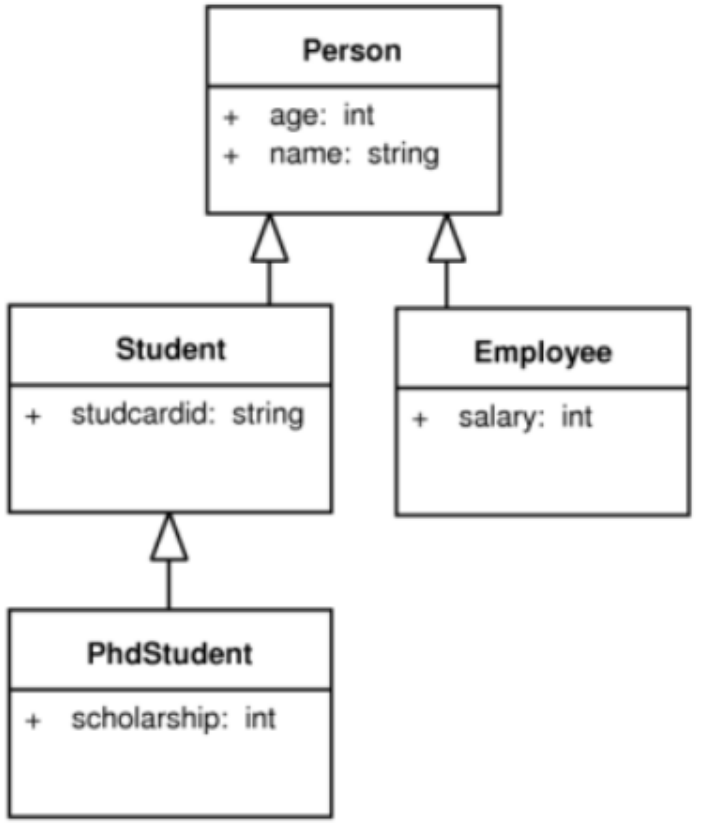
- Superclass = Parent class = Base class
- Subclass = Child class = Derived class



Class Inheritance (cont'd)

- Class inheritance provides code reusability and cleaner way of programing
- IS-A relationship
- Python supports multiple inheritance





Example: Class Window

- Window is a subclass of Rectangle
  - Rectangle with a position and resizable

```
class Window(Rectangle):
    def __init__(self, x, y, h, w):
        super().__init__(h, w)
        self.x = x
        self.y = y

    def resize(self, nh, nw):
        self.h = nh
        self.w = nw
```

new instance variables

new method

```
w1 = Window(0, 0, 5, 7)
w2 = Window(2, 3, 4, 7)
print(w1)
print(w2)
print(w1.area())
print(w2.peri())
print(w1 == w2)
w2.resize(5, 7)
print(w1 == w2)
```

Relationships among Classes and Objects

- `issubclass`(*class*, *classinfo*)
  - Return True if *class* is a subclass of *classinfo*
  - If *classinfo* is a tuple, every entry in *classinfo* will be checked

- `isinstance`(*object*, *classinfo*)
  - Return True if *object* is an instance of the *classinfo*
  - If *classinfo* is a tuple, every entry in *classinfo* will be checked

```
>>> issubclass(bool, int)
True
>>> issubclass(int, (float, object))
True
>>> issubclass(Window, Rectangle)
True
>>> r = Rectangle(2,3)
>>> w = Window(0,0,5,9)
>>> isinstance(r, Rectangle)
True
>>> isinstance(r, Window)
False
>>> isinstance(w, Rectangle)
True
>>> isinstance(w, Window)
True
```

Polymorphism

- Method overriding
  - Python allows us to define methods in the child class with the same name as defined in their parent class

- The version of the method called depends on the type of the object

```
class A:
    def f(self):
        return g()

    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()
print(a.f(), b.f())
print(a.g(), b.g())
```

Class Variables and Methods

- Class variables and class methods
  - Variables and methods defined at the class level
  - Shared by all the objects

- Built-in class variables
  - `__dict__`
  - `__doc__`
  - `__name__`
  - `__module__`
  - `__bases__`

```
class A:
    def foo(self):
        print('foo()')

    @classmethod
    def class_foo(cls):
        print(cls.__name__)
        print('class_foo()')

a = A()
A.foo()
a.foo()
A.class_foo()
a.class_foo()
```

Static Methods

- Static methods
  - Similar to class methods: bound to the class and not the object of the class
  - Can't access class states
  - Just like general functions but only accessible within the class

- A class method receives the class as implicit first argument
  - Class methods can access class variables

```
class B:
    def foo(self):
        print('foo()')

    @staticmethod
    def static_foo(x, y):
        print('static_foo()', x+y)

b = B()
B.foo()
b.foo()
B.static_foo(2, 3)
b.static_foo(5, 7)
```