

Lecture 20

Subclasses & Inheritance

Announcements for Today

Reading

- Today: Chapter 18
- Online reading for Thursday

- **Prelim, Nov 12th 7:30-9:00**

- Material up to Thursday
- Review posted on Thursday
- Recursion + Loops + Classes

- **S/U Students are exempt**

- **Conflict with Prelim time?**

- Prelim 2 Conflict on CMS
- Submit by Thursday

Assignments

- A4 graded by end of week
 - Survey is still open
- A5 was posted Friday
 - Shorter written assignment
 - Due Thursday at Midnight
- A6 also posted Friday
 - Due a **week after** prelim
 - Designed to take two weeks
 - Finish first part before exam

An Application

- **Goal:** Presentation program (e.g. PowerPoint)
- **Problem:** There are many types of content
 - **Examples:** text box, rectangle, image, etc.
 - Have to write code to display each one
- **Solution:** Use object oriented features
 - Define class for every type of content
 - Make sure each has a draw method:

```
for x in slide[i].contents:  
    | x.draw(window)
```

Sharing Work

- These classes will have a lot in common
 - Drawing handles for selection
 - Background and foreground color
 - Current size and position
 - And more (see the formatting bar in PowerPoint)
- **Result:** A lot of repetitive code
- **Solution:** Create one class with shared code
 - All content are *subclasses* of the *parent* class

Abbreviate
as SC to right

Defining a Subclass

```
class SlideContent(object):  
    """Any object on a slide."""  
    def __init__(self, x, y, w, h): ...  
    def draw_frame(self): ...  
    def select(self): ...
```

Superclass
Parent class
Base class

SlideContent

Subclass
Child class
Derived class

TextBox

Image

```
class TextBox(SlideContent):  
    """An object containing text."""  
    def __init__(self, x, y, text): ...  
    def draw(self): ...
```

SC

```
__init__(x,y,w,h)  
draw_frame()  
select()
```

```
class Image(SlideContent):  
    """An image."""  
    def __init__(self, x, y, image_file): ...  
    def draw(self): ...
```

TextBox(SC)

Image(SC)

```
__init__(x,y,text)  
draw()
```

```
__init__(x,y,img_f)  
draw()
```

Class Definition: Revisited

class *<name>*(*<superclass>*):

"""Class specification"""

getters and setters

initializer (`__init__`)

definition of operators

definition of methods

anything else

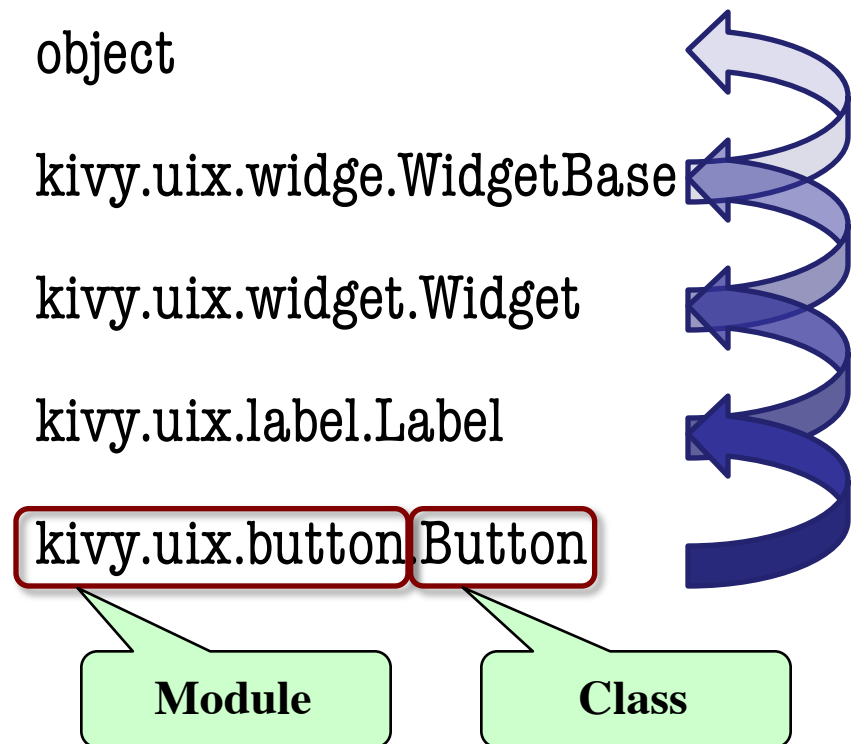
Class type to extend
(may need module name)

- Every class must extend *something*
- Previous classes all extended *object*

object and the Subclass Hierarchy

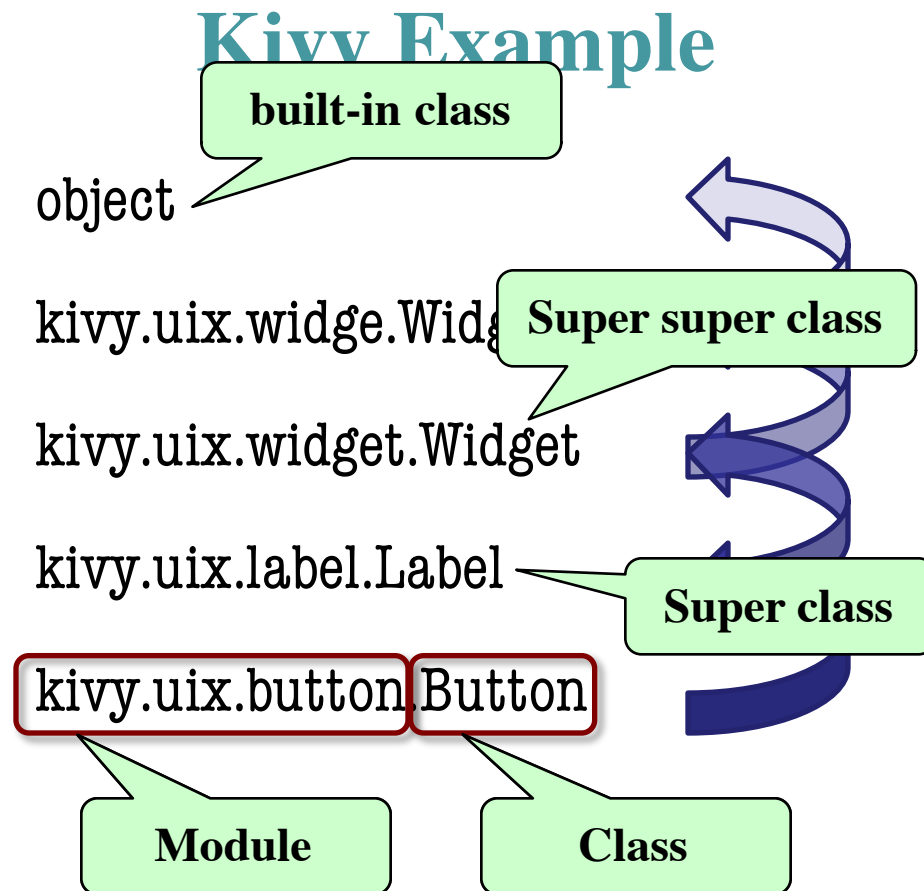
- Subclassing creates a **hierarchy** of classes
 - Each class has its own super class or parent
 - Until object at the “top”
- object has many features
 - Special built-in fields: `__class__`, `__dict__`
 - Default operators: `__str__`, `__repr__`

Kivy Example



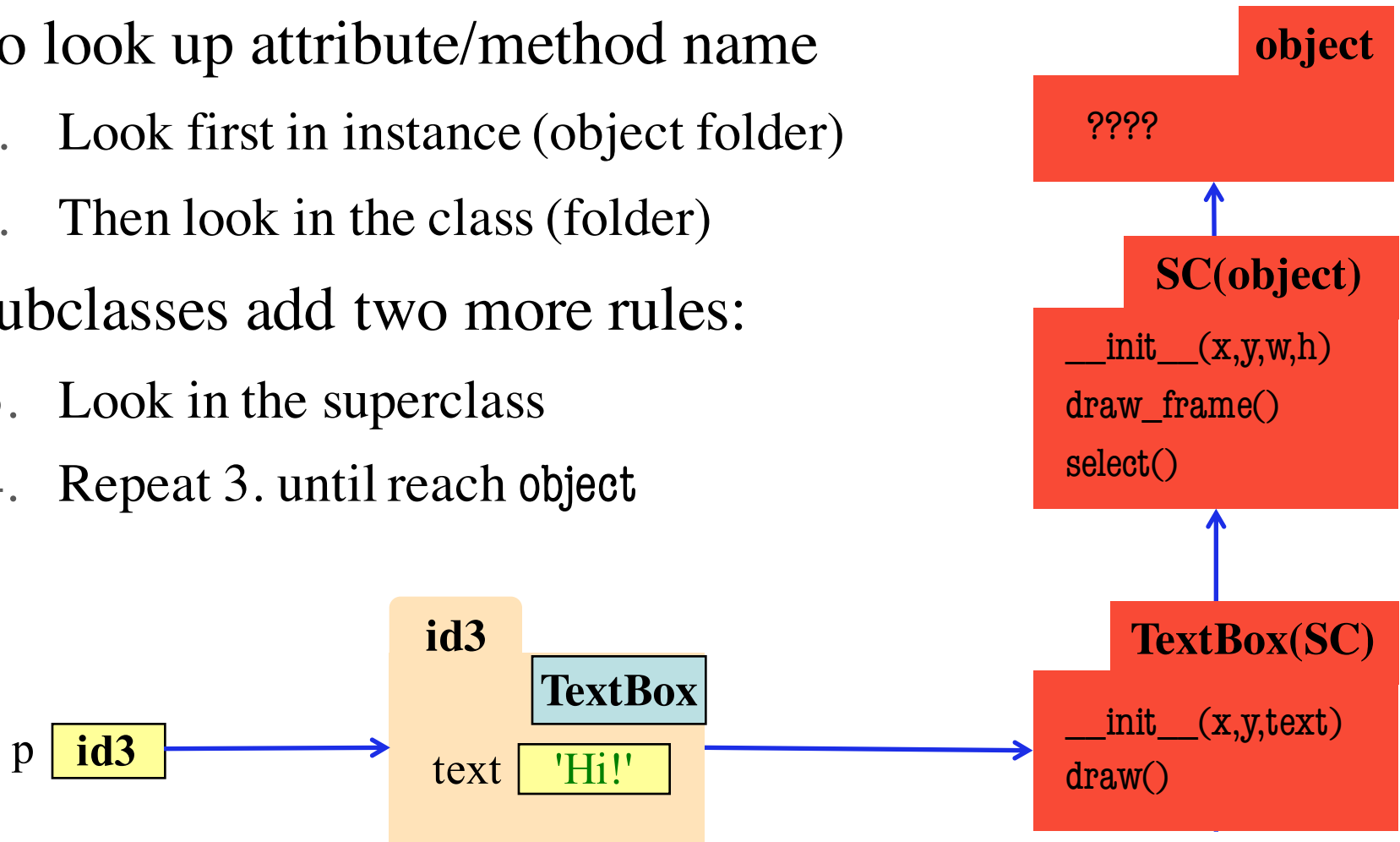
object and the Subclass Hierarchy

- Subclassing creates a **hierarchy** of classes
 - Each class has its own super class or parent
 - Until object at the “top”
- object has many features
 - Special built-in fields: `__class__`, `__dict__`
 - Default operators: `__str__`, `__repr__`



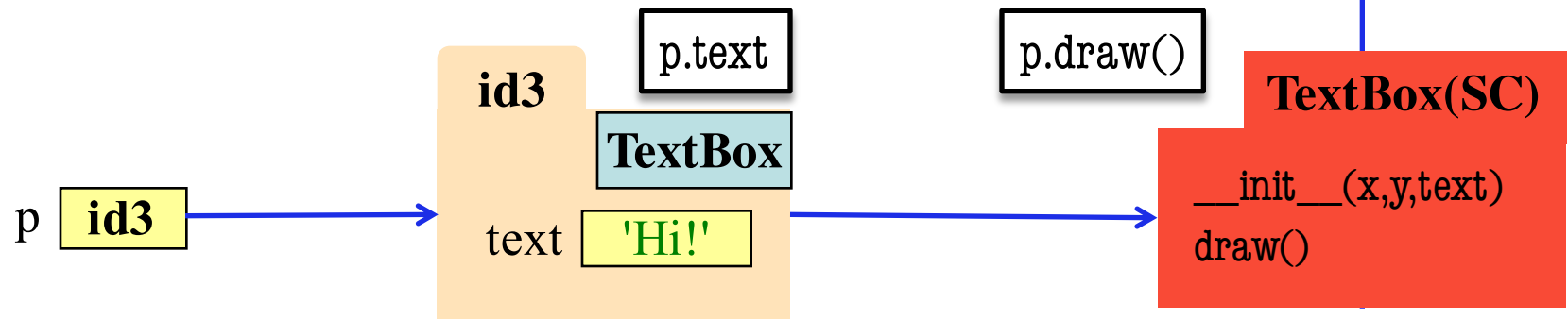
Name Resolution Revisited

- To look up attribute/method name
 1. Look first in instance (object folder)
 2. Then look in the class (folder)
- Subclasses add two more rules:
 3. Look in the superclass
 4. Repeat 3. until reach object



Name Resolution Revisited

- To look up attribute/method name
 1. Look first in instance (object folder)
 2. Then look in the class (folder)
- Subclasses add two more rules:
 3. Look in the superclass
 4. Repeat 3. until reach object



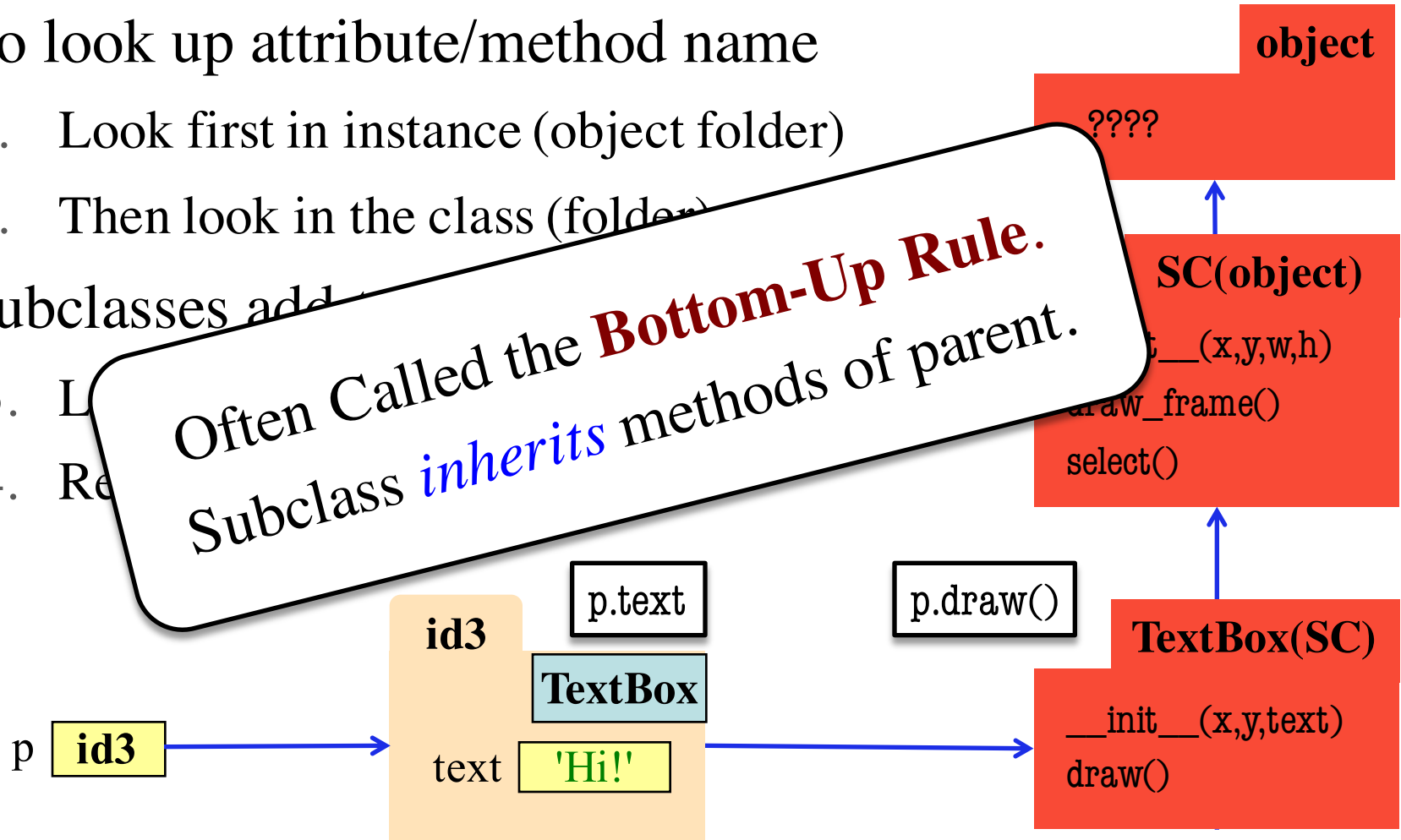
Name Resolution Revisited

- To look up attribute/method name
 1. Look first in instance (object folder)
 2. Then look in the class (folder)

- Subclasses add

3. Look
4. Re

Often Called the **Bottom-Up Rule**.
Subclass *inherits* methods of parent.



A Simpler Example

```
class Employee(object):
```

```
    """Instance is salaried worker
```

```
    INSTANCE ATTRIBUTES:
```

```
        name [string]: full name
```

```
        start [int ≥ -1, -1 if unknown]:
```

```
            first year hired
```

```
        salary [float]: yearly wage"""
```

```
class Executive(Employee):
```

```
    """An Employee with a bonus
```

```
    INSTANCE ATTRIBUTES:
```

```
        bonus [float]: annual bonus"""
```

object

```
__init__()
```

```
__str__()
```

```
__eq__()
```

Employee

```
__init__(n,d,s)
```

```
__str__()
```

```
__eq__()
```

Executive

```
__init__(n,d,b)
```

```
__str__()
```

```
__eq__()
```

A Simpler Example

```
class Employee(object):  
    """Instance is salaried worker  
    INSTANCE ATTRIBUTES:  
        name [string]: full name  
        start [int ≥ -1, -1 if unknown]:  
            first year hired  
        salary [float]: yearly wage"""
```

```
class Executive(Employee):  
    """An Employee with a bonus  
    INSTANCE ATTRIBUTES:  
        bonus [float]: annual bonus"""
```

object

```
__init__()  
__str__()  
__eq__()
```

All double
underscore
methods are
in class object

Employee

```
__init__(n,d,s)  
__str__()  
__eq__()
```

Executive

```
__init__(n,d,b)  
__str__()  
__eq__()
```

Method Overriding

- Which `__str__` do we use?
 - Start at bottom class folder
 - Find first method with name
 - Use that definition
- New method definitions **override** those of parent
- Also applies to
 - Initializers
 - Operators
 - Propertiesall “methods”

object

```
__init__()  
__str__()  
__eq__()
```

Employee

```
__init__(n,d,s)  
__str__()  
__eq__()
```

Executive

```
__init__(n,d,b)  
__str__()  
__eq__()
```



Accessing the “Previous” Method

- What if you want to use the original version method?
 - New method = **original**+**more**
 - Do not want to repeat code from the original version
- Call old method **explicitly**
 - Use method as a function
 - Pass object as first argument
- **Example:**
Employee.__str__(self)
- **Cannot do with properties**

object

```
__init__()  
__str__()  
__eq__()
```

Employee

```
__init__(n,d,s)  
__str__()  
__eq__()
```

Executive

```
__init__(n,d,b)  
__str__()  
__eq__()
```



Accessing the “Previous” Method

- What if you want to use the original version method?
 - New method = **original**+**more**
 - Do not want to repeat code from the original version
- Call old method **explicitly**
 - Use method as a function
 - Pass object as first argument
- **Example:**
Employee.__str__(self)
- **Cannot do with properties**

```
class Employee(object):  
    """An Employee with a salary"""  
    ...  
    def __str__(self):  
        return (self.name +  
                ', year ' + str(self.start) +  
                ', salary ' + str(self.salary))
```

```
class Executive(Employee):  
    """An Employee with a bonus."""  
    ...  
    def __str__(self):  
        return (Employee.__str__(self)  
                + ', bonus ' + str(self.bonus) )
```


Primary Application: Initializers

```
class Employee(object):  
    ...  
    def __init__(self,n,d,s=50000.0):  
        self._name = n  
        self._start = d  
        self._salary = s
```

```
class Executive(Employee):  
    ...  
    def __init__(self,n,d,b=0.0):  
        Employee.__init__(self,n,d)  
        self._bonus = b
```

object

```
__init__()  
__str__()  
__eq__()
```

Employee

```
__init__(n,d,s)  
__str__()  
__eq__()
```

Executive

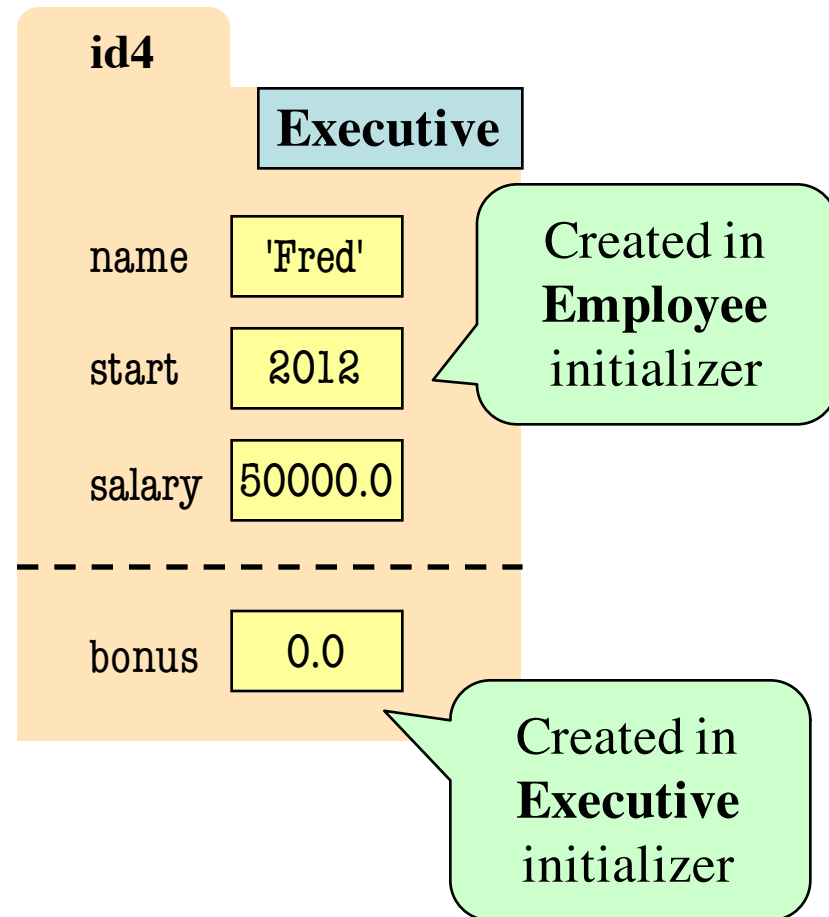
```
__init__(n,d,b)  
__str__()  
__eq__()
```



Instance Attributes are (Often) Inherited

```
class Employee(object):  
    ...  
    def __init__(self,n,d,s=50000.0):  
        self._name = n  
        self._start = d  
        self._salary = s
```

```
class Executive(Employee):  
    ...  
    def __init__(self,n,d,b=0.0):  
        Employee.__init__(self,n,d)  
        self._bonus = b
```

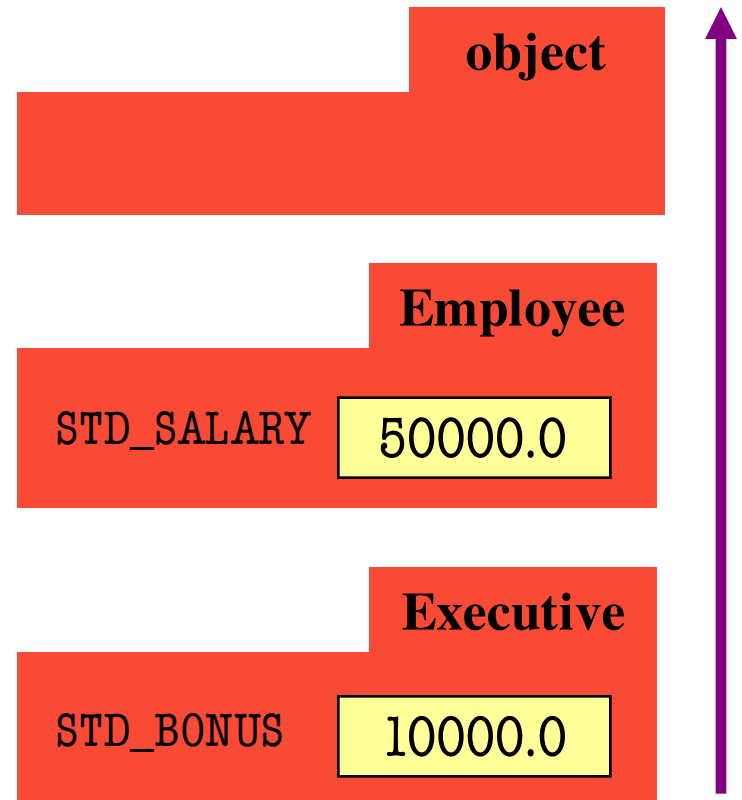


Also Works With Class Attributes

Class Attribute: Assigned outside of any method definition

```
class Employee(object):  
    """Instance is salaried worker"""  
    # Class Attribute  
    STD_SALARY = 50000.0
```

```
class Executive(Employee):  
    """An Employee with a bonus."""  
    # Class Attribute  
    STD_BONUS = 10000.0
```



Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of a.f()?

A: 10

B: 14

C: 5

D: **ERROR**

E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of a.f()?

A: 10 **CORRECT**

B: 14

C: 5

D: **ERROR**

E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:

```
>>> a = A()
>>> b = B()
```
- What is value of `b.f()`?

A: 10

B: 14

C: 5

D: **ERROR**

E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of b.f()?

A: 10
B: 14 **CORRECT**
C: 5
D: **ERROR**
E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of b.x?

A: 4

B: 3

C: 42

D: **ERROR**

E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:

```
>>> a = A()
>>> b = B()
```
- What is value of `b.x`?

A: 4
B: 3 **CORRECT**
C: 42
D: **ERROR**
E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of `a.z`?

A: 4

B: 3

C: 42

D: **ERROR**

E: I don't know

Name Resolution and Inheritance

```
class A(object):
    x = 3 # Class Attribute
    y = 5 # Class Attribute

    def f(self):
        | return self.g()

    def g(self):
        | return 10
```

```
class B(A):
    y = 4 # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        | return 14

    def h(self):
        | return 18
```

- Execute the following:
 >>> a = A()
 >>> b = B()
- What is value of a.z?

A: 4

B: 3

C: 42

D: **ERROR** **CORRECT**

E: I don't know

Properties and Inheritance

- Properties: all or nothing
 - Typically inherited
 - Or fully overridden (both getter and setter)
- When override property, **completely** replace it
 - Cannot use `super()`
- **Very rarely** overridden
 - **Exception:** making a property read-only
 - See `employee2.py`

```
class Employee(object):
```

```
    ...  
    @property  
    def salary(self):  
        | return self._salary  
  
    @salary.setter  
    def salary(self,value):  
        | self._salary = value
```

```
class Executive(Employee):
```

```
    ...  
    @property # no setter; now read-only  
    def salary(self):  
        | return self._salary
```