

Special Topics in Computer Science- CSC 4992

Class

More Modeling (inheritance, polymorphism, ...)

SavingsAccount Class

```
class SavingsAccount(object):  
    """This class represents a savings account."""  
  
    def __init__(self, name, pin, balance = 0.0):  
        self._name = name  
        self._pin = pin  
        self._balance = balance  
  
    def deposit(self, amount):  
        self._balance += amount  
  
    def withdraw(self, amount):  
        self._balance -= amount
```

The Interface of the **SavingsAccount** Class

```
SavingsAccount(name, pin, bal)    # Returns a new object  
  
getBalance()                      # Returns the current balance  
  
deposit(amount)                  # Makes a deposit  
  
withdraw(amount)                 # Makes a withdrawal  
  
computeInterest()                # Computes the interest and  
                                # deposits it
```

The Interface of the SavingsAccount Class

```
SavingsAccount(name, pin, bal)    # Returns a new object

getBalance()                      # Returns the current balance

deposit(amount)                   # Makes a deposit

withdraw(amount)                  # Makes a withdrawal

computeInterest()                # Computes the interest and
                                # deposits it

str(account)                      # String representation of account

a1 == a2                          # Test for equality
```

Accessing Data in an Object

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)

>>> print('Name:      ' + account.getName() + '\n' + \
          'PIN:       ' + account.getPin() + '\n' + \
          'Balance:   ' + str(account.getBalance()))
Name:      Ken
PIN:       3322
Balance:   1000.00
```

An object's data can be viewed or accessed by using its *accessor methods*

String Representation

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)

>>> print(str(account))    # Same as account.__str__()
Name:      Ken
PIN:       3322
Balance:   1000.00
```

Each class can include a string conversion method named `__str__`

This method is automatically called when the `str` function is called with the object as a parameter

String Representation

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)
```

```
>>> print(str(account))
```

```
Name:      Ken  
PIN:       3322  
Balance:   1000.00
```

```
>>> print(account)                # Better still
```

```
Name:      Ken  
PIN:       3322  
Balance:   1000.00
```

Each class can include a string conversion method named `__str__`

`print` runs `str` if it's given an object to print - way cool!

The `__str__` Method

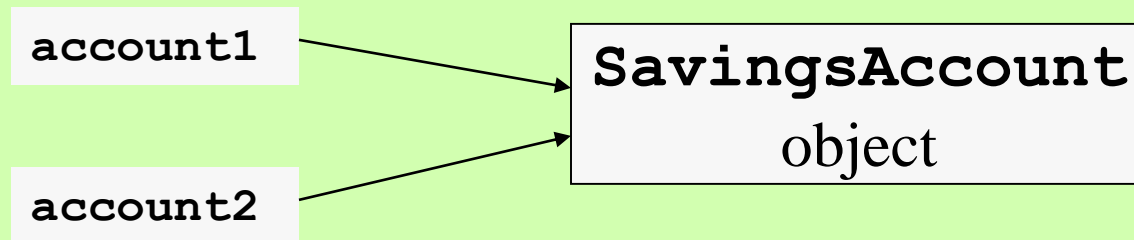
```
class SavingsAccount(object):  
    """This class represents a savings account."""  
  
    def __init__(self, name, pin, balance = 0.0):  
        self._name = name  
        self._pin = pin  
        self._balance = balance  
  
    def __str__(self):  
        return 'Name:      ' + self._name + '\n' + \  
               'PIN:       ' + self._pin + '\n' + \  
               'Balance: ' + str(self._balance)
```

As a rule of thumb, you should include an `__str__` method in each new class that you define

Equality with ==

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 == account2
True
```

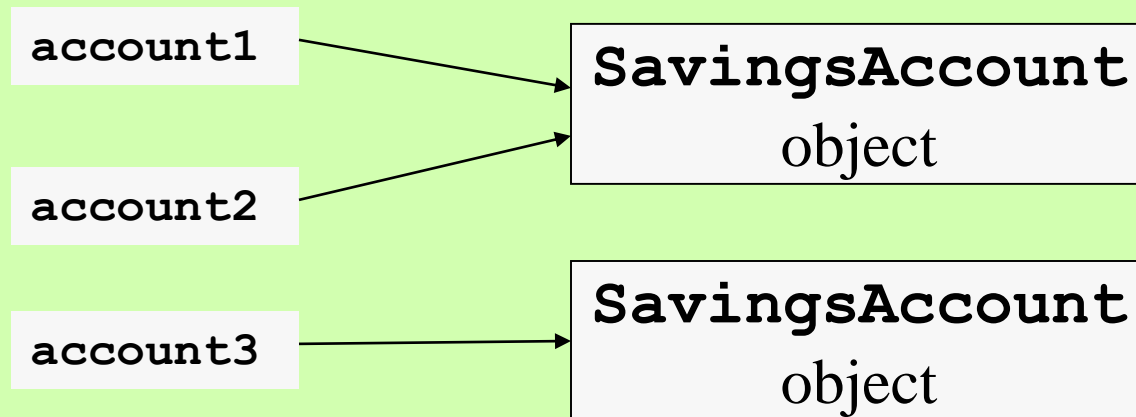
The two variables refer to the same, identical object



Equality with ==

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 == account2
True
>>> account3 = SavingsAccount("ken", "1000", 4000.00)
>>> account1 == account3
False
```

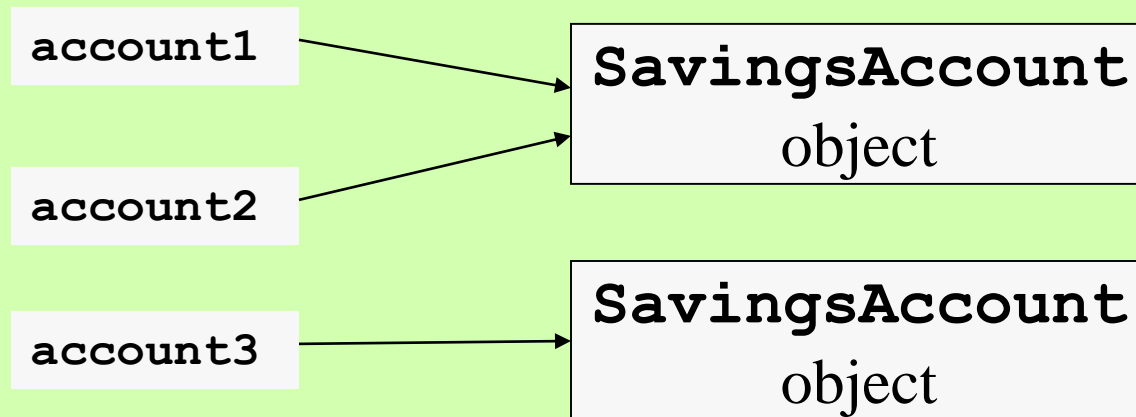
The two account objects have the same contents but aren't equal



Equality with **is**

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 is account2
True
>>> account3 = SavingsAccount("ken", "1000", 4000.00)
>>> account1 is account3
False
```

By default, `==` uses `is`, which tests for object identity



What Is Equality?

- *Object identity*: two variables refer to the exact same object
- *Structural equivalence*: two variables refer to distinct objects that have the same contents
- Object identity is pretty strict, maybe too strict
- `==` actually tests for structural equivalence with Python's data structures

Define the Method `__eq__`

```
class SavingsAccount(object):  
    """This class represents a savings account."""  
  
    def __init__(self, name, pin, balance = 0.0):  
        self._name = name  
        self._pin = pin  
        self._balance = balance  
  
    def __eq__(self, other):  
        if self is other: return True  
        if type(other) != SavingsAccount: return False  
        return self._name == other._name and \  
            self._pin == other._pin
```

Test for identity, then type, then equality of selected attributes

The operator `==` actually calls the method `__eq__`

Other Common Methods

Python allows you to define other methods that are automatically called when objects are used with certain functions or operators

Function or Operator	Method Called
<code>len(obj)</code>	<code>obj.__len__()</code>
<code>obj1 in obj2</code>	<code>obj2.__contains__(obj1)</code>
<code>obj1 + obj2</code>	<code>obj1.__add__(obj2)</code>
<code>obj1 < obj2</code>	<code>obj1.__lt__(obj2)</code>
<code>obj1 > obj2</code>	<code>obj1.__gt__(obj2)</code>
<code>obj[index]</code>	<code>obj.__getitem__(index)</code>
<code>obj1[index] = obj2</code>	<code>obj1.__setitem__(index, obj2)</code>

Example: Rational Numbers

```
from rational import Rational
```

```
oneHalf = Rational(2, 4)
```

```
oneThird = Rational(1, 3)
```

```
print(oneHalf) # Prints 1/2
```

```
print(oneThird < oneHalf) # Prints True
```

```
theSum = oneHalf + oneThird
```

```
print(theSum) # Prints 5/6
```

Example: Rational Numbers

```
class Rational(object):  
    """This class represents a rational number."""  
  
    def __init__(self, numerator = 1, denominator = 1):  
        self._numer = numerator  
        self._denom = denominator  
        self._reduce()  
  
    def __str__(self):  
        return str(self._numer) + "/" + str(self._denom)
```

The `__init__` and `__str__` methods should always be defined first; then you can test the class to verify that its objects are appropriately instantiated.

Addition of Rational Numbers

```
class Rational(object):  
    """This class represents a rational number."""  
  
    def __add__(self, other):  
        """Returns the sum of self and other."""  
        numerSum = self._numer * other._denom + \  
                    other._numer * self._denom  
        denomSum = self._denom * other._denom  
        return Rational(numerSum, denomSum)
```

$$n_{\text{sum}} / d_{\text{sum}} = (n_1 * d_1 + n_2 * d_1) / (d_1 * d_2)$$

Comparison of Rational Numbers

```
class Rational(object):  
    """This class represents a rational number."""  
  
    def __lt__(self, other):  
        """Returns True if self < other or False otw."""  
        extremes = self._numer * other._denom  
        means = other._numer * self._denom  
        return means < extremes
```

Operator

Method

Implementation

$r1 < r2$

`__lt__`

`means < extremes`

$r1 > r2$

`__gt__`

`means > extremes`

$r1 == r2$

`__eq__`

`means == extremes`

$r1 \leq r2$

`__le__`

`means <= extremes`

$r1 \geq r2$

`__ge__`

`means >= extremes`

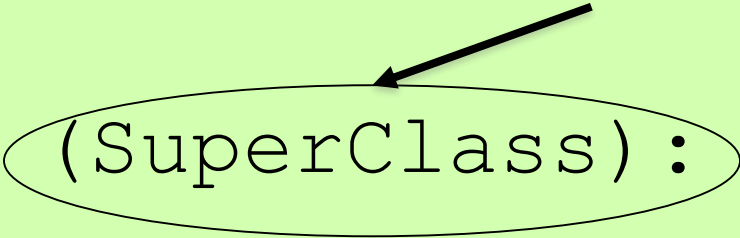
Class-Class relations

- Classes can have a separate relationship with other classes
- the relationships forms a hierarchy
 - **hierarchy**: A body of persons or things ranked in grades, orders or classes, one above another
- when we create a class, which is itself another object, we can state how it is related to other classes
- the relationship we can indicate is the class that is 'above' it in the hierarchy

class statement

name of the class above
this class in the hierarchy

```
class MyClass (SuperClass) :  
    pass
```

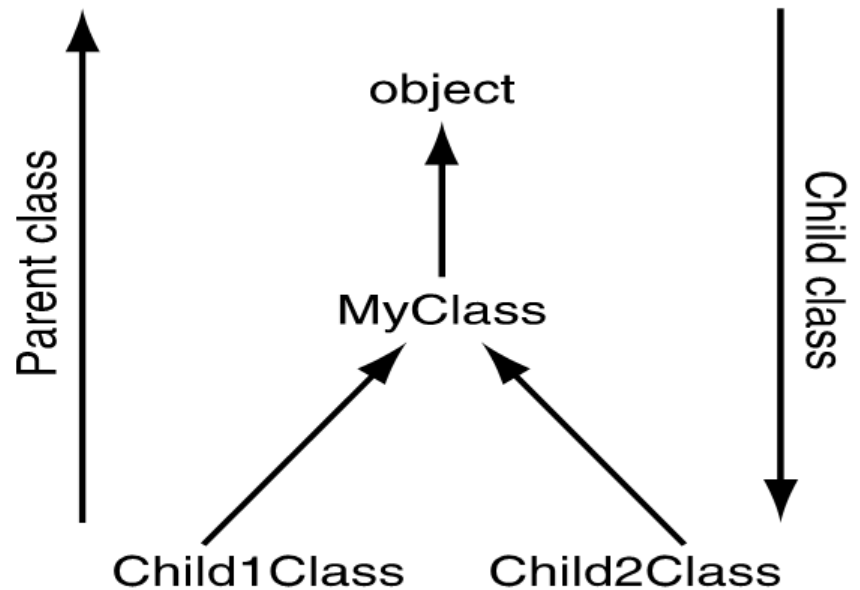


- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

```
class MyClass (object):  
    pass
```

```
class Child1Class (MyClass):  
    pass
```

```
class Child2Class (MyClass):  
    pass
```



A simple class hierarchy.

Initializing the New Class

- Consider an example. We want to specialize a new class as a subclass of list.

`class MyClass (SupperClass):`

- easy enough, but we want to make sure that we get our new class instances initialized the way they are supposed to, by calling
- `__init__` of the `super class`

Calling the super class init

If we don't explicitly say so, our class may inherit stuff from the super class, but we must make sure we call it in the proper context. For example, our `__init__` would be:

```
def __init__(self):  
    SupperClass.__init__(self)  
    # do anything else special to MyClass
```

Inheritance

Defining a new class with no or little modification to an existing class. The new class is called **subclass (or child) class** and the one from which it inherits is called the **superclass(or parent) class**.

Example of Inheritance (Polygon)

```
class Polygon:
```

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

```
        self.n = no_of_edges
```

```
        self.sides = [0 for i in range(no_of_sides )]
```

```
    def inputSides(self):
```

```
        self.edges = [float(input("Enter Edge"+str(i+1)+" : "))
```

```
        for i in range(self.n)]
```

```
    def dispSides(self):
```

```
        for i in range(self.n):
```

```
            print("Edge",i+1,"is",self.sides[i])
```

Example of Inheritance (Triangle)

```
class Triangle (Polygon):  
    def __init__(self):  
        Polygon.__init__(self , 3 )  
    def findArea (self):  
        a, b, c = self.sides  
        . . .  
  
    def dispSides(self):  
        for i in range(self.n):  
            print("Edge",i+1,"is",self.sides[i])
```

Example of Inheritance (Triangle)

```
>>> t = Triangle ()
```

```
>>> t.inputSides()
```

```
Enter side 1 : 3
```

```
Enter side 2 : 5
```

```
Enter side 3 : 4
```

```
>>> t.dispSides()
```

```
Side 1 is 3.0
```

```
Side 2 is 5.0
```

```
Side 3 is 4.0
```

```
>>> t.findArea()
```

```
The area of the triangle is 6.00
```

First Design

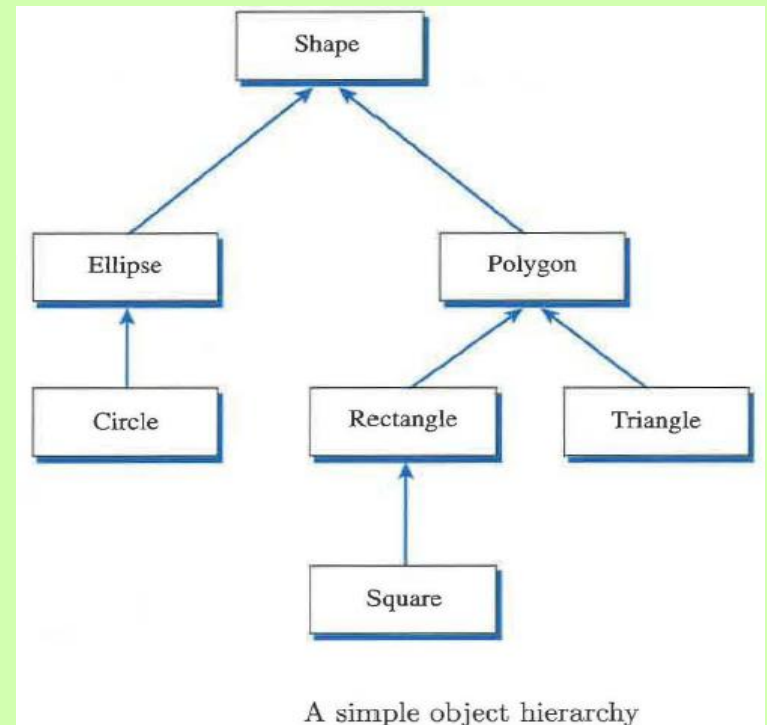
- We will design and implement a set of classes using a powerful programming concept called **inheritance**, an object-oriented programming technique.
- When designing a project, it is a good idea to begin by making a list of the different kinds of objects involved and the relationships between those objects.
- As we make the list of objects, we will try to identify two important kinds of relationships between the objects. The relationships we are looking for are **IS-A** and **HAS-A.**,

First Design (cont'd.)

- The **IS-A relationship** describes two objects where one object is a more special instance of the other.
- For example, a square is *a* more specific instance of a rectangle, and a circle is *a* more specific instance of an ellipse.
- As we identify these IS-A relationships, we also look for functionality that each of the instances have in common.
- A **HAS-A relationship** describes two objects where one object uses another object.
- For example, if you think about a circle, each circle *has* *a* center point. A rectangle *has* *a* lower-left-corner point and an upper-right-corner point.

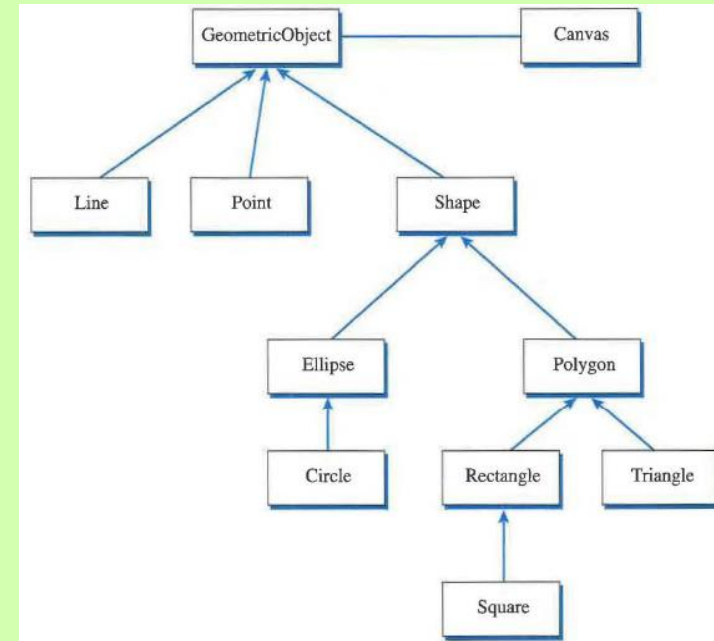
First Design (cont'd.)

- An initial set of objects might look like the following:
 - Square
 - Circle
 - Oval
 - Rectangle
 - Triangle
 - Polygon
 - Line
 - Point
 - Canvas



First Design (cont'd.)

- In object-oriented programming, the **IS-A** links define an **inheritance** hierarchy.
- Inheritance is the idea that a more general class is a parent class of a more specific class (a child class).
- The parent class may have methods that can be shared with any child. This idea of shared methods is how we avoid **duplicating code**.
- When two classes are connected by an IS-A link, we call a class that is directly above another class in the inheritance hierarchy a superclass.
- The class where the IS-A link originates is called the subclass.



An expanded object hierarchy

First Design (cont'd.)

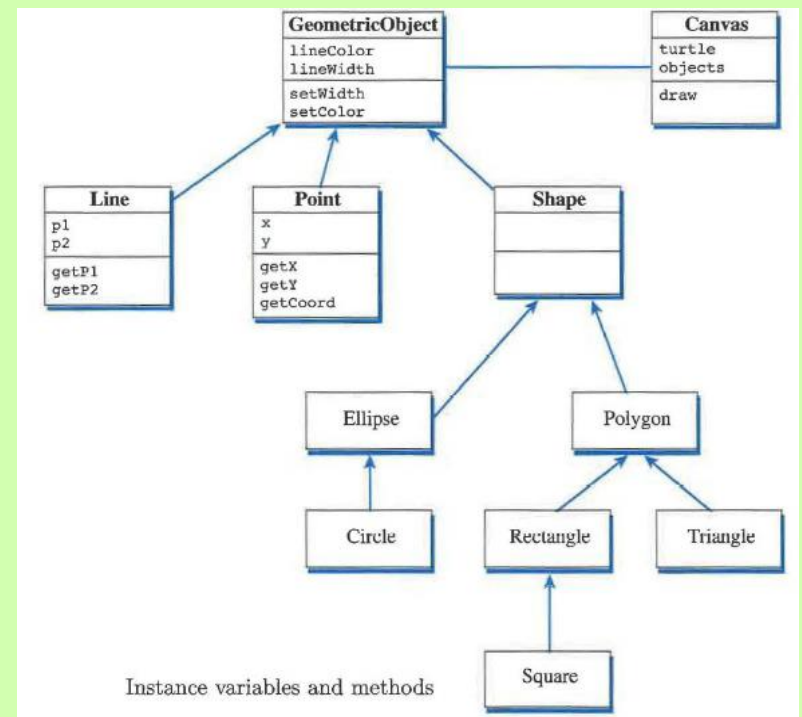
- In the next stage of our design we need to ask the following questions:
 - What things should each object know?
 - What things should each object be able to do?
- The things we want each object to know will lead us to the set of instance variables.
- The things that each object can do will give us the set of methods we need to write.

First Design (cont'd.)

- There are several things we want our objects to remember about themselves:
 - Fill color
 - Outline color
 - Position on the canvas
 - Line width
- The list of things that our shapes should be able to do:
 - Set or change the outline color: `setOutline (color)`.
 - Set or change the fill color: `setFill (color)` .
 - Move to a new position on the canvas: `move(dx,dy)` .
 - Set or change the width of a line or outline: `setWidth(w)` .
 - Set or change the color of a line or point: `setColor(color)`.

First Design (cont'd.)

- The Line class has two instance variables: a beginning point and an ending point.
- Like Point, Line inherits the instance variables lineColor and lineWidth from GeometricObject.



GeometricObject Class

- GeometricObject and Canvas are peers at the top level of our inheritance hierarchy.
- The **GeometricObject** class is called an **abstract** class. This is because it is an abstraction of its child classes: Point, Line, Rectangle, and others.
- The `_draw` method prints an error message; this is because we should never actually create an instance of the GeometricObject class.
- An abstract class provides us with one place to define instance variables and methods that are used by all of the child classes.
- The GeometricObject class contains two instance variables: `lineColor` and `lineWidth`. Because the instance variables are initialized by the GeometricObject constructor, they do not need to be initialized by any of the child classes.

GeometricObject Class (cont'd.)

```
1 class GeometricObject:
2
3     def __init__(self):
4         self.lineColor = 'black'
5         self.lineWidth = 1
6
7     def getColor(self):
8         return self.lineColor
9
10    def getWidth(self):
11        return self.lineWidth
12
13    def setColor(self, color):
14        self.lineColor = color
15
16    def setWidth(self, width):
17        self.lineWidth = width
18
19    def _draw(self):
20        print("Error: You must define _draw in subclass")
```

Point Class

- The Point class is a child of GeometricObject, so the class statement designates GeometricObject as the parent.
- The syntax we use to designate one class as the parent of another is to place the parent in parentheses after the child.
- Point begins with the instance variables and methods provided by GeometricObject and adds new instance variables and methods of its own.

Point Class (cont'd.)

- The line *super().__init__()* makes sure that the `__init__` method in the parent class is called and that the instance variables inherited from `GeometricObject` have their proper initial values.
- The `super` function returns a special super-object that knows how to properly call the `__init__` method for the parent class.
- The `_draw` method implements the drawing of a `Point`, using the `turtle` that is passed as a parameter.
- `turtle` is moved to the right place on the canvas with its tail up using the `goto` method.
- The parameters passed to `goto` are the instance variables `self.x` and `self.y`.
- Once the `turtle` is in position, a dot is drawn using the instance variables `self.lineWidth` and `self.lineColor`.

Point Class (cont'd.)

```
1 class Point(GeometricObject):
2     def __init__(self, x,y):
3         super().__init__()
4         self.x = x
5         self.y = y
6
7     def getCoord(self):
8         return (self.x,self.y)
9
10    def getX(self):
11        return self.x
12
13    def getY(self):
14        return self.y
15
16    def _draw(self,turtle):
17        turtle.goto(self.x,self.y)
18        turtle.dot(self.lineWidth,self.lineColor)
```

Line Class

- The Line class is very similar to the Point class with the exception that Line has **two instance variables** that are **Points**. The other difference is that `_draw` draws a line rather than a simple point.
- Notice that we need to make two calls to the `turtle` to **set the line color and the line width**. This is an important step because we cannot make any assumptions

Line Class (cont'd.)

```
1 class Line(GeometricObject):
2     def __init__(self, p1,p2):
3         super().__init__()
4         self.p1 = p1
5         self.p2 = p2
6
7     def getP1(self):
8         return self.p1
9
10    def getP2(self):
11        return self.p2
12
13    def _draw(self, turtle):
14        turtle.color(self.getColor())
15        turtle.width(self.getWidth())
16        turtle.goto(self.p1.getCoord())
17        turtle.down()
18        turtle.goto(self.p2.getCoord())
```

Canvas Class

- A place *for* us to draw GeometricObjects

```
1 myCanvas = Canvas(800,600)
2 myLine = Line(Point(-100,-100),Point(100,100))
3 myCanvas.draw(myLine)
```

- The Canvas class is responsible in some way for drawing a GeometricObject.
- With all of the turtle's functionality, implementing the Canvas class should be pretty easy.
- We will have to write a draw method for the Canvas that can take any shape we give it and draw it using the turtle.

Canvas Class (cont'd.)

```
class canvas:

    def __init__(self, w, h):
        self.turtle=Turtle()
        self.width=w
        self.height=h

        self.turtle.hideturtle()

    def __draw__(self,gObject):
        self.turtle.up()
        gObject._draw(self.turtle)
```

Testing Our Implementation

- With our first four classes written, let's try our first test program and see what we can learn about our implementation.

```
>>> from draw import *  
>>> myCanvas = Canvas(800,600)  
>>> myLine = Line(Point(-100,-100),Point(100,100))  
>>> myCanvas.draw(myLine)
```

Testing Our Implementation(cont'd.)

- Let's continue to explore this example and see what we can learn about the objects we have created.

```
>>> myLine
<draw.Line object at 0x106f6b0>
>>> myCanvas
<draw.Canvas instance at 0x1070328>
>>> isinstance(myLine, Line)
True
>>> myLine.getColor()
'black'
>>> myLine.getWidth()
1
>>> p = myLine.getP1()
>>> p
<draw.Point object at 0x6c950>
>>> p.getX()
-100
>>> p.getY()
-100
>>> p.getWidth()
1
>>> p.getColor()
'black'
```

Polymorphism

- Polymorphism: an object's ability to take different forms
- Essential ingredients of polymorphic behavior:
 - Ability to define a method in a superclass and override it in a subclass
 - Subclass defines method with the same name
 - Ability to call the correct version of overridden method depending on the type of object that called for it

Polymorphism (cont'd.)

- In previous inheritance examples showed how to override the `__init__` method
 - Called superclass `__init__` method and then added onto that
- The same can be done for any other method
 - The method can call the superclass equivalent and add to it, or do something completely different

The `isinstance` Function

- Polymorphism provides great flexibility when designing programs
- `isinstance` function: determines whether object is an instance of a class
 - Format: `isinstance(object, class)`