10.5 Polymorphism

- `QuestionDemo2` passed two `ChoiceQuestion` objects to the `presentQuestion()` method
  - Can we write a `presentQuestion()` method that displays both `Question` and `ChoiceQuestion` types?
  - **With inheritance, this goal is very easy to realize!**
  - In order to present a question to the user, we need not know the exact type of the question.
  - We just display the question and check whether the user supplied the correct answer.

```
def presentQuestion(q) :
    q.display()
    response = input("Your answer: ")
    print(q.checkAnswer(response))
```

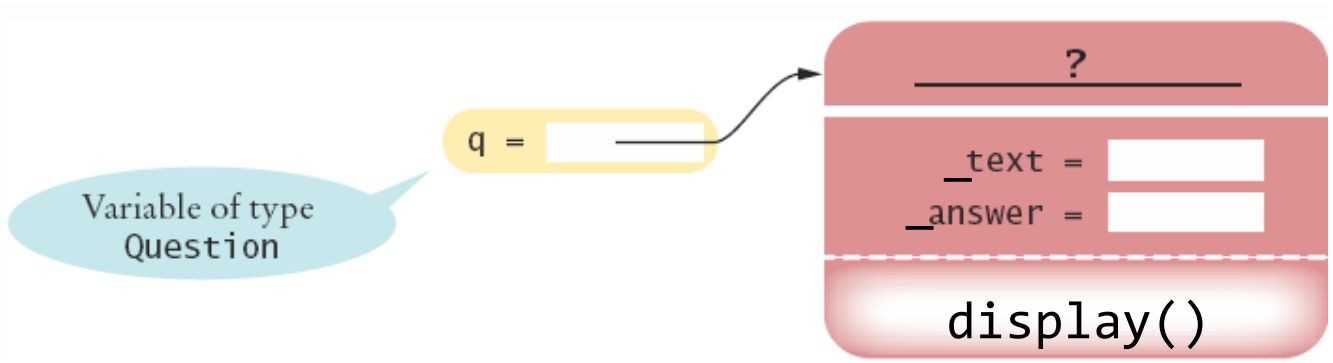
Which `Display()` method was called?

- `presentQuestion()` simply calls the `display()` method of whatever type is passed:

```
def presentQuestion(q) :
  q.display()
  . . .
```

  - If passed an object of the `Question` class:
    - `Question display()`
  - If passed an object of the `ChoiceQuestion` class:
    - `ChoiceQuestion display()`

- The variable q does not know the type of object to which it refers:



Why Does This Work?

- As discussed in Section 10.1, we can substitute a subclass object whenever a superclass object is expected:

```
second = ChoiceQuestion()
presentQuestion(second)   # OK to pass a ChoiceQuestion
```

- Note however you cannot substitute a superclass object when a subclass object is expected.
  - An `AttributeError` exception will be raised.
  - The parent class has fewer capabilities than the child class (you cannot invoke a method on an object that has not been defined by that object's class).

Polymorphism Benefits

- In Python, method calls *are always determined by the type of the actual object,* **not** the type of the variable containing the object reference
  - This is called *dynamic method lookup*
  - Dynamic method lookup allows us to treat objects of different classes in a uniform way
- This feature is called **polymorphism**
- We ask multiple objects to carry out a task, and each object does so in its own way
- Polymorphism makes programs *easily extensible*

Questiondemo3.py (1)

```
1  ##
2  #  This program shows a simple quiz with two question types.
3  #
4
5  from questions import Question, ChoiceQuestion
6
7  def main() :
8     first = Question()
9     first.setText("Who was the inventor of Python?")
10    first.setAnswer("Guido van Rossum")
11
12    second = ChoiceQuestion()
13    second.setText("In which country was the inventor of Python born?")
14    second.addChoice("Australia", False)
15    second.addChoice("Canada", False)
16    second.addChoice("Netherlands", True)
17    second.addChoice("United States", False)
18
19    presentQuestion(first)
20    presentQuestion(second)
21
```

Creates an object of the `Question` class

Creates an object of the `ChoiceQuestion` class, uses new `addChoice()` method.

Calls `presentQuestion()` - next page - passed both types of objects.

Questiondemo3.py (2)

```
22  ## Presents a question to the user and checks the response.
23  #  @param q the question
24  #
25  def presentQuestion(q) :
26     q.display()    # Uses dynamic method lookup
27     response = input("Your answer: ")
28     print(q.checkAnswer(response))    # Uses dynamic method lookup
29
30  # Start the program.
31  main()
```

Receives a parameter of the super-class type

Uses appropriate display method.

Special Topic 10.2

- Subclasses and Instances:
  - You learned that the `isinstance()` function can be used to determine if an object is an instance of a specific class.
  - But the `isinstance()` function can also be used to determine if an object is an instance of a subclass.
  - For example, the function call:

```
isinstance(q, Question)
```

  - will return `True` if `q` is an instance of the `Question` class or of any subclass that extends the `Question` class,
  - otherwise, it returns `False`.

Use of `Isinstance()`

- A common use of the `isinstance()` function is to verify that the arguments passed to a function or method are of the correct type.

```
def presentQuestion(q) :
    if not isintance(q, Question) :
        raise TypeError("The argument is not a Question or
           one of its subclasses.")
```

Special Topic 10.3

- Dynamic Method Lookup
  - Suppose we move the `presentQuestion()` method to inside the `Question` class and call it as follows:

```
cq = ChoiceQuestion()
cq.setText("In which country was the inventor of Python born?")
. . .
cq.presentQuestion()
```

```
def presentQuestion(self) :
   self.display()
   response = input("Your answer: ")
   print(self.checkAnswer(response))
```

  - Which `display()` and `checkAnswer()` methods will be called?

Dynamic Method Lookup

- If you look at the code of the `presentQuestion()` method, you can see that these methods are executed on the `self` reference parameter.
  - Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display()` and `checkAnswer()` methods are called automatically.
  - This happens even though the `presentQuestion()` method is declared in the `Question` class, which has no knowledge of the `ChoiceQuestion` class.

```
class Question :
   def presentQuestion(self) :
       self.display()
       response = input("Your answer: ")
       print(self.checkAnswer(response))
```

Special Topic 10.4

- Abstract Classes and methods
  - If it is desirable to ***force*** subclasses to override a method of a base class, you can declare a method as `abstract`.
  - You cannot instantiate an object that has `abstract` methods
    - Therefore the class is considered `abstract` (it has 1+ abstract methods)
    - It's a tool to force programmers to create subclasses (avoids the trouble of coming up with useless default methods that others might inherit by accident).
  - In Python, there is no explicit way to specify that a method is an abstract method. Instead, the common practice among Python programmers is to have the method raise a `NotImplementedError` exception as its only statement:

```
class Account :
    . . .
    def deductFees(self) :
       raise NotImplementedError
```

Common Error 10.3

- Don't Use Type Tests
  - Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if isinstance(q, ChoiceQuestion) :   # Don't do this.
    # Do the task the ChoiceQuestion way.
elif isinstance(q, Question) :
    # Do the task the Question way.
```

  - This is a poor strategy.
  - If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
elif isinstance(q, NumericQuestion) :
    # Do the task the NumericQuestion way.
```

Alternate to Type Tests

- Polymorphism
  - Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead.
  - Declare a method `doTheTask()` in the superclass, override it in the subclasses, and call

```
q.doTheTask()
```

Steps to Using Inheritance

- As an example, we will consider a bank that offers customers the following account types:

  1) A savings account that earns interest. The interest compounds monthly and is based on the minimum monthly balance.

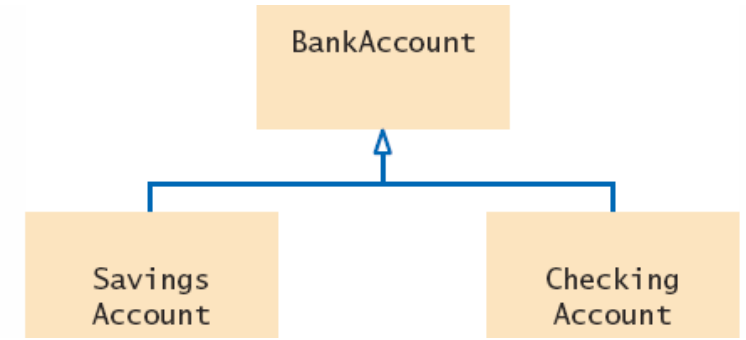
  2) A checking account that has no interest, gives you three free withdrawals per month, and charges a $1 transaction fee for each additional withdrawal.

- The program will manage a set of accounts of both types
  - It should be structured so that other account types can be added without affecting the main processing loop.
- The menu: `D)eposit W)ithdraw M)onth end Q)uit`
  - For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.
  - In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Steps to Using Inheritance

1) List the classes that are part of the hierarchy.
   - `SavingsAccount`
   - `CheckingAccount`

2) Organize the classes into an inheritance.
   hierarchy
   Base on superclass `BankAccount`



3) Determine the common responsibilities.
   a. Write Pseudocode for each task
   b. Find common tasks

Using Inheritance: Pseudocode

```
For each user command
    If it is a deposit or withdrawal
        Deposit or withdraw the amount from the specified account.
        Print the balance.
    If it is month end processing
        For each account
            Call month end processing.
            Print the balance.
```

```
Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
```

Steps to Using Inheritance

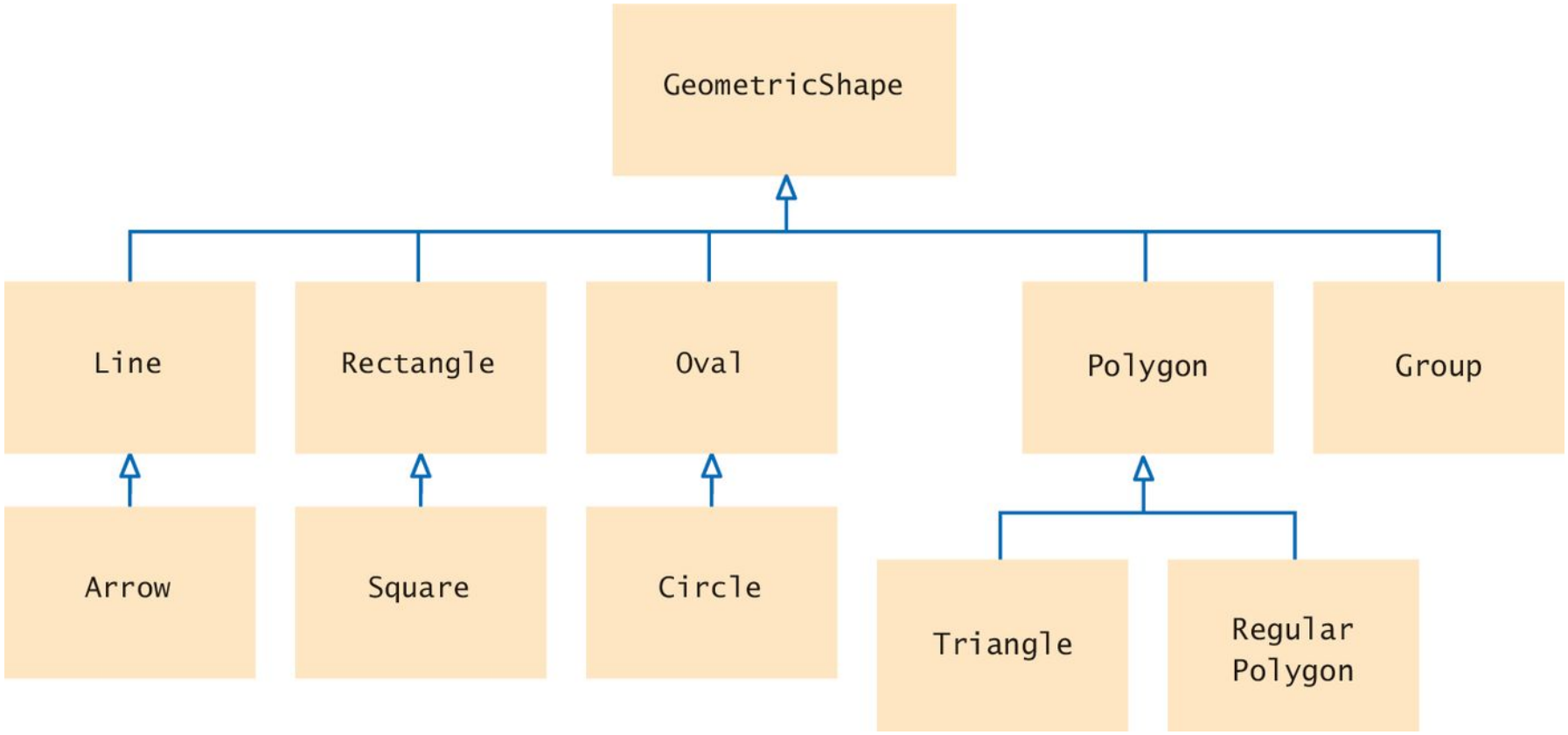
4) Decide which methods are overridden in subclasses.
   - For each subclass and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden
5) Declare the public interface of each subclass.
   - Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden.
   - You also need to specify how the objects of the subclasses should be constructed.
6) Identify instance variables.
   - List the instance variables for each class. Place instance variables that are common to all classes in the base of the hierarchy.
7) Implement constructors and methods.
8) Construct objects of different subclasses and process them.

Eye Break

10.6 Application

- Creating a geometric shape class hierarchy
  - To create complex scenes beyond the simple graphics introduced in Chapter 2, you may need a large number of shapes that vary in color, size, or location.
  - Rather than calling the various methods again and again, it would be useful to have classes that model the various geometric shapes.
  - Using shape classes, a programmer can create a shape object with specific characteristics, then use the same object to draw multiple instances of the shape with only minor changes.

Inheritance Diagram of Geometric Shapes



The Base Class

- The `GeometricShape` class should provide the functionality that is common among the various subclasses:
  - Setting the colors used to draw the shape.
  - Getting and setting the coordinates for the upper-left corner of a bounding box.
  - Computing the width and height of the shape (or the bounding box used to define the shape).
  - Drawing the shape on a canvas.
- Due to the amount of variation between shapes all subclasses will have to override the `draw()` method.

Attributes of the Base Class

- The instance variables `_fill` and `_outline can store` the fill and outline colors used by the shapes.
- Coordinates of the top left hand corner of the bounding box for the shape can be stored in the instance variables _x and _y

Setting the Attributes

- The constructor of the `GeometricShape` base class needs to define the common instance variables.

```
class GeometricShape :
    ## Construct a basic geometric shape.
    # @param x the x-coordinate of the shape
    # @param y the y-coordinate of the shape
    def _ _init_ _(self, x, y) :
        self._x = x
        self._y = y
        self._fill = None
        self._outline = "black"
```

Accessor Methods

- As expected they will return the values stored in the instance variables.

```
def getX(self) :
    return self._x

def getY(self) :
    return self._y
```

- Because the `getWidth()` and `getHeight()` methods return zero they should be overridden by subclasses.

```
def getWidth(self) :
     return 0

def getHeight(self) :
     return 0
```

Mutator Methods

- We define three mutator methods for setting the colors.
- Two methods set the outline or fill color individually, and the third method sets both to the same color:

```
def setFill(self, color = None) :
    self._fill = color

def setOutline(self, color = None) :
    self._outline = color

def setColor(self, color) :
    self._fill = color
    self._outline = color
```

Other Methods

- The `move()` method moves the shape by a given amount (x, y) coordinates.

```
def moveBy(self, dx, dy) :
    self._x = self._x + dx
    self._y = self._y + dy
```

- As indicated earlier, the `draw()` method has to be overridden for each subclass's specific shape but the common operation (setting the drawing colors) is included here.

```
def draw(self, canvas) :
    canvas.setFill(self._fill)
    canvas.setOutline(self._outline)
```

The `Rectangle` Class (1)

- The `Rectangle` class inherits from `GeometricShape`.
- The constructor passes the upper-left corner to the superclass and stores the width and height.

```
def _ _init_ _(self, x, y, width, height) :
    super()._ _init_ _(x, y)
    self._width = width
    self._height = height
```

The Rectangle Class (2)

- The draw method is overridden in the Rectangle subclass to include the call to the appropriate canvas method.

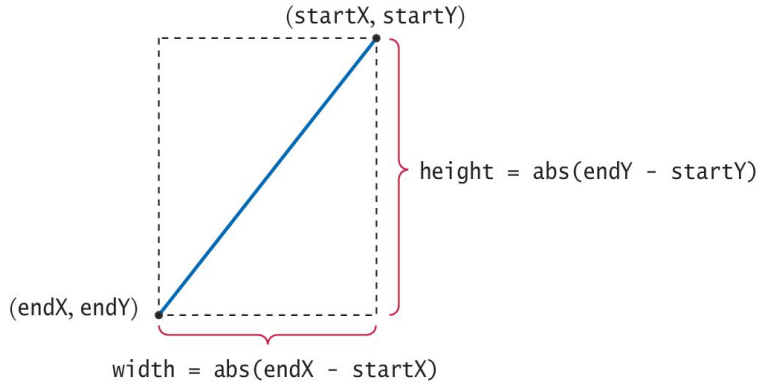
```
def draw(self, canvas) :
    super().draw(canvas)   # Parent method sets colors
    canvas.drawRect(self.getX(), self.getY(), self._width,
       self._height)
```

```
def getWidth(self) :
    return self._width

def getHeight(self) :
    return self._height
```

The `Line` Class (1)

- A line is specified by its start and end points.



- It is possible that neither of these points is the upper-left corner of the bounding box.
- Instead, we need to compute the smaller of the *x*- and *y*-coordinates and pass those values to the superclass constructor.

The `Line` Class (2)

- We also need to store the start and end points in instance variables because we need them to draw the line.

```
def _ _init_ _(self, x1, y1, x2, y2) :
    super()._ _init_ _(min(x1, x2), min(y1, y2))
    self._startX = x1
    self._startY = y1
    self._endX = x2
    self._endY = y2
```

`Line` Class: Methods (1)

- The width and height are the differences between the starting and ending *x*- and *y*-coordinates.
- However, if the line isn't sloping downward, we need to take the absolute values of the difference.

```
def getWidth(self) :
    return abs(self._endX - self._startX)

def getHeight(self) :
    return abs(self._endY - self._startY)
```

`Line` Class: Methods (2)

- As noted the `draw()` method must be overridden

```
def draw(self, canvas) :
    super().draw(canvas)
    canvas.drawLine(self._startX, self._startY, self._endX,
       self._endY)
```

- Also the `moveBy()` method must be overridden so that it adjusts the starting and ending points, in addition to the top-left corner.

```
def moveBy(self, dx, dy) :
    super().moveBy(dx, dy)
    self._startX = self._startX + dx
    self._startY = self._startY + dy
    self._endX = self._endX + dx
    self._endY = self._endY + dy
```

Wrapper Class: Square

- A wrapper class wraps or encapsulates the functionality of another class to provide a more convenient interface.
- For example, we could draw a square using the `Rectangle` subclass.
  - But it requires that we supply both the width and height.
  - Because a square is a special case of a rectangle, we can define a Square subclass that extends, or *wraps*, the Rectangle class and only requires one value, the length of a side.

```
class Square(Rectangle) :
    def _ _init_ _(self, x, y, size) :
        super()._ _init_ _(x, y, size, size)
```

Testshapes.py

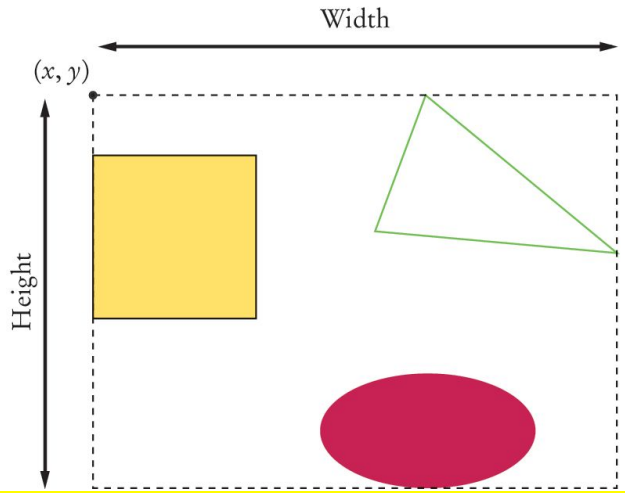
```
1  ##
2  #  This program tests several of the geometric shape classes.
3  #
4
5  from graphics import GraphicsWindow
6  from shapes import Rectangle, Line
7
8  # Create the window.
9  win = GraphicsWindow()
10 canvas = win.canvas()
11
12 # Draw a rectangle.
13 rect = Rectangle(10, 10, 90, 60)
14 rect.setFill("light yellow")
15 rect.draw(canvas)
16
17 # Draw another rectangle.
18 rect.moveBy(rect.getWidth(), rect.getHeight())
19 rect.draw(canvas)
20
21 # Draw six lines of different colors.
22 colors = ["red", "green", "blue", "yellow", "magenta", "cyan"]
23
24 line = Line(10, 150, 300, 150)
25
26 for i in range(6) :
27    line.setColor(colors[i])
28    line.draw(canvas)
29    line.moveBy(10, 10)
30
31 win.wait()
```

Groups of Shapes

- The Group subclass does not actually draw a geometric shape.
- Instead it can be used to group basic geometric shapes to create a complex shape.
  - For example, suppose you construct a door using a rectangle, a circle for the doorknob, and a circle for the peep hole.
  - The three components can be stored in a Group in which the individual shapes are defined relative to the position of the group.
  - This allows the entire group to be moved to a different position without having to move each individual shape.

The Group Class

- To create a Group, you provide the coordinates of the upper-left corner of its bounding box.
- The class defines an instance variable that stores the shapes in a list.
- As new shapes are added to a Group object, the width and height of the bounding box expands to enclose the new shapes.



Group Class Methods (1)

- Create the group with its bounding box positioned at (*x*, *y*).

```
def _ _init_ _(self, x = 0, y = 0) :
    super()._ _init_ _(x, y)
    self._shapeList = []
```

- Adding a shape to the group involves several steps.
  - First, the shape has to be appended to the list:

```
def add(self, shape) :
    self._shapeList.append(shape)
```

Group Class Methods (2)

- The individual shapes are positioned relative to the upper-left corner of the group's bounding box.
- We must ensure that each shape is positioned below and to the right of this point. If it is not, it must be moved.

```
# Keep the shape within top and left edges of the
# bounding box.
if shape.getX() < 0 :
    shape.moveBy(-shape.getX(), 0)
if shape.getY() < 0 :
    shape.moveBy(0, -shape.getY())
```

Group Class Methods (3)

- The width of the group is determined by the rightmost extent of any of the group's members.
- The rightmost extent of a shape is `shape.getX()` + `shape.getWidth()`. The following method computes the maximum of these extents.

```
def getWidth(self) :
    width = 0
    for shape in self._shapeList :
        width = max(width, shape.getX() +
            shape.getWidth())
    return width
```

Group Class Methods (4)

- The height of the group (the bottommost extent) is computed in the same way as the width.

```
def getHeight(self) :
    height = 0
    for shape in self._shapeList :
        height = max(height, shape.getY() +
           shape.getHeight())
    return height
```

Group Class Methods (5)

- The entire group can be drawn on the canvas.
- The shapes contained in the group are defined relative to the upper-left corner of its bounding box.
- Before a shape can be drawn, it has to be moved to its position relative to the upper-left corner of the group's bounding box.

```
def draw(self, canvas) :
    for shape in self._shapeList :
        shape.moveBy(self.getX(), self.getY())
        shape.draw(canvas)
        shape.moveBy(-self.getX(), -self.getY())
```

Summary: Inheritance

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object.
- A subclass inherits all methods that it does not override.
- A subclass can override a superclass method by providing a new implementation.
- In Python a class name inside parentheses in the class header indicates that a class inherits from a superclass.

Summary: Overriding Methods

- An overriding method can extend or replace the functionality of the superclass method.
- Use the reserved word `super` to call a superclass method.
- To call a superclass constructor, use the `super` reserved word before the subclass defines its own instance variables.
- The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Summary: Polymorphism

- A subclass reference can be used when a superclass reference is expected.
- Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
- An `abstract` method is a method whose implementation is not specified.

Summary: Use Inheritance for Designing a Hierarchy of Shapes

- The `GeometricShape` class provides methods that are common to all shapes.
- •Each subclass of `GeometricShape` must override the `draw()` method.
- A shape class constructor must initialize the coordinates of its upper-left corner.
- Each shape subclass must override the methods for computing the width and height.
- A `Group` contains shapes that are drawn and moved together.