CHAPTER 16 Object-Oriented Programming

16.1 HISTORY OF OBJECTS

16.2 WORKING WITH TURTLES

16.3 TEACHING TURTLES NEW TRICKS

16.4 AN OBJECT-ORIENTED SLIDE SHOW

16.5 OBJECT-ORIENTED MEDIA

16.6 JOE THE BOX

16.7 WHY OBJECTS?

Chapter Learning Objectives

- To use object-oriented programming to make programs easier to develop in teams, more robust, and easier to debug.
- To understand such features of object-oriented programs as polymorphism, encapsulation, inheritance, and aggregation.
- To be able to choose between different styles of programming for different purposes.

16.1 HISTORY OF OBJECTS

The most common style of programming today is **object-oriented programming**. We're going to define it in contrast with the procedural programming that we've been doing up until now.

Back in the 1960s and 1970s, procedural programming was the dominant form of programming. People used *procedural abstraction* and defined lots of functions at high and low levels, and reused their functions wherever possible. This worked reasonably well—up to a point. As programs got really large and complex, with many programmers working on them at the same time, procedural programming started to break down.

Programmers ran into problems with procedure conflicts. People would write programs that modified data in ways that other people didn't expect. They would use the same names for functions and find that their code couldn't be integrated into one large program.

There were also problems in *thinking* about programs and the tasks the programs were supposed to perform. Procedures are about *verbs*—tell the computer to do this,

M01_GUZD3514_03_SE_C16 30/1/2012 19:27 Page 376

tell the computer to do that. But it's not clear whether that's the way people think best about problems.

Object-oriented programming is *noun-oriented programming*. Someone building an object-oriented program starts by thinking about what the nouns are in the *domain* of the problem—what are the people and things that are part of this problem and its solution? The process of identifying the objects, what each of them knows about (with respect to the problem), and what each of them has to do is called **object-oriented analysis**.

Programming in an object-oriented way means that you define variables (called **instance variables**) and functions (called **methods**) *for the objects*. In the most object-oriented languages, programs have very few or even *no* global functions or variables—things that are accessible everywhere. In the original object-oriented programming language, Smalltalk, objects could *only* get things done by asking each other to do things via their methods. Adele Goldberg, one of the pioneers of object-oriented programming, calls this "Ask, don't touch." You can't just "touch" data and do whatever you want with it—instead, you "ask" objects to manipulate their data through their methods. That is a good goal even in languages like Python or Java where objects *can* manipulate each others' data directly.

The term *object-oriented programming* was invented by Alan Kay. Kay is a brilliant multidisciplinary character—he holds undergraduate degrees in mathematics and biology, a Ph.D. in computer science, and has been a professional jazz guitarist. In 2004, he was awarded the ACM Turing Award, which is sort of the Nobel Prize of computing. Kay saw object-oriented programming as a way of developing software that could truly scale to large systems. He described objects as being like biological *cells* that work together in well-defined ways to make the whole organism work. Like cells, objects would:

- Help manage *complexity* by distributing responsibility for tasks across many objects rather than one big program.
- Support *robustness* by making the objects work relatively independently.
- Support *reuse* because each object would provide *services* to other objects (tasks that the object would do for other objects, accessible through its methods), just as real-world objects do.

The notion of starting from nouns is part of Kay's vision. **Software**, he said, is actually a *simulation* of the world. By making software *model* the world, it becomes clearer how to make software. You look at the world and how it works, then copy that into software. Things in the world *know* things—these become **instance variables**. Things in the world can *do* things—these become **methods**.

Of course, we've been using objects already. Pictures, sounds, samples, and colors are all objects. Our lists of pixels and samples are examples of *aggregation*, which is creating collections of objects. The functions we've been using are actually just covering up the underlying methods. We can just call the objects' methods directly, which we will do later in this chapter.

16.2 WORKING WITH TURTLES

Seymour Papert, at MIT, used robot turtles to help children think about how to specify procedures in the late 1960s. The turtle had a pen in the middle of it that could be raised and lowered to leave a trail of its movements. As graphical displays became available, he used a virtual turtle on a computer screen instead of a robotic turtle.

Part of the media support in JES provides graphical turtle objects. Turtles make a great introduction to the ideas of objects. We manipulate turtle objects that move around a world. The turtles know how to move and turn. The turtles have a pen in the middle of them that leaves a trail to show their movements. The world keeps track of the turtles that are in it.

16.2.1 Classes and Objects

How does the computer know what we mean by a turtle and a world? We have to define what a turtle is, what it knows about, and what it can do. We have to define what a world is, what it knows about, and what it can do. In Python we do this by defining classes. A class defines what things or objects (instances) of that class know and can do. The media package for JES defines classes that define what we mean by a turtle and a world.

Object-oriented programs consist of objects. We create objects from classes. The class knows what each object of that class needs to keep track of and what it should be able to do. You can think of a class as an object factory. The factory can create many objects. A class is also like a cookie cutter. You can make many cookies from one cookie cutter and they will all have the same shape. Or you can think of the class as a blueprint and the objects as the houses that you can create from the blueprint.

To create and initialize a world you use `makeWorld()`. To create a turtle object, you can use `makeTurtle(world)`. That looks pretty similar to `makePicture` and `makeSound`—there is a pattern here, but we will introduce a new one, a more standard Python syntax in just a bit. Let's create a new world object.

```
>>> makeWorld()
```

This will create a world object and display a window that shows the world. It will just start as an all-white picture in a frame titled, "World." But we can't refer to it since we didn't name it.

Here we name the world object that gets created `earth`, and then create a turtle object in the world named `earth`. We will name the turtle object `tina`.

```
>>> earth = makeWorld()
>>> tina = makeTurtle(earth)
>>> print tina
No name turtle at 320, 240 heading 0.0.
```

The turtle object appears in the center of the world (320, 240) and facing north (*a heading of 0*) (Figure 16.1). The turtle hasn't been assigned a name yet.

The turtle support in JES allows us to create many turtles. Each new turtle will appear in the center of the world.

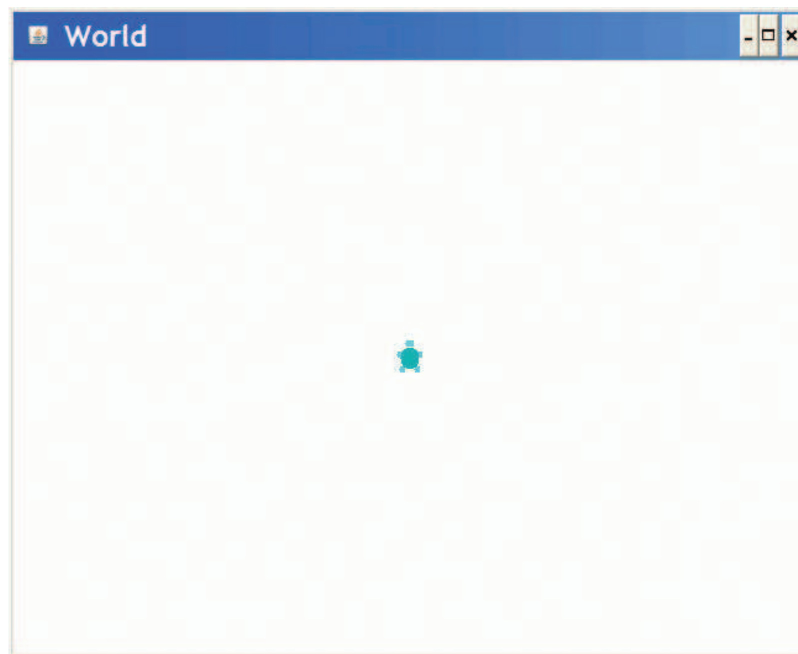

FIGURE 16.1
Creating a turtle in the world.

```
>>> sue = makeTurtle(earth)
```

16.2.2 Sending Messages to Objects

We can ask the turtle to do things by sending a message to the turtle object, which we also think of as calling a method on an object. We do this using *dot notation*. In dot notation we ask an object to do something by specifying the name of the object and then a '.' and then the function to execute (`name.function(parameterList)`). We saw dot notation with strings in Section 10.3.1.

```
>>> tina.forward()
>>> tina.turnRight()
>>> tina.forward()
```

Notice that only the turtle that we asked to do the actions moves (Figure 16.2). We can make the other one move by asking it to do things as well.

```
>>> sue.turnLeft()
>>> sue.forward(50)
```

Notice that different turtles have different colors (Figure 16.3). As you can see turtles know how to turn left and right, using `turnLeft()` and `turnRight()`. They also can go forward in the direction they are currently heading using `forward()`. By default they go forward 100 pixels, but you can also specify how many pixels to go forward,

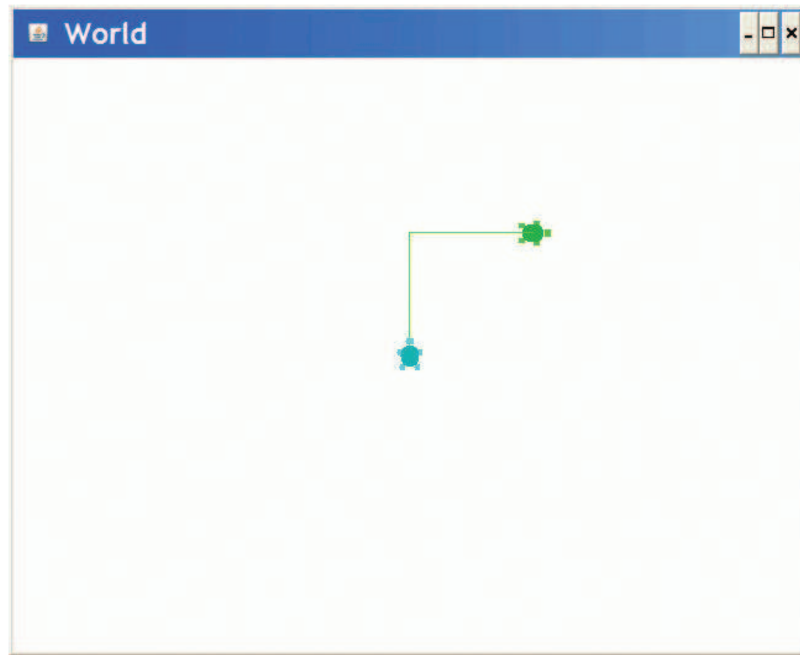

FIGURE 16.2
Asking one turtle to move and turn, while the other one remains.

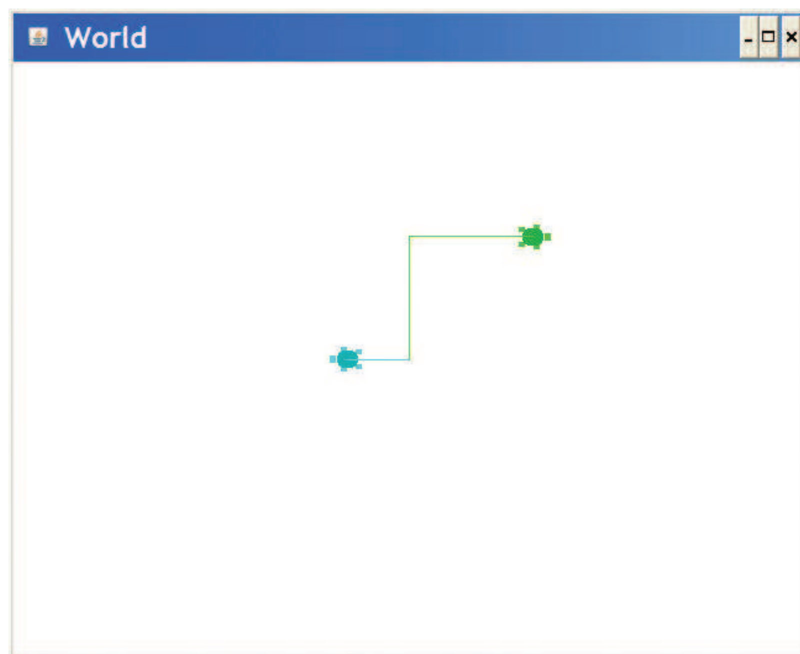

FIGURE 16.3
After the second turtle moves.

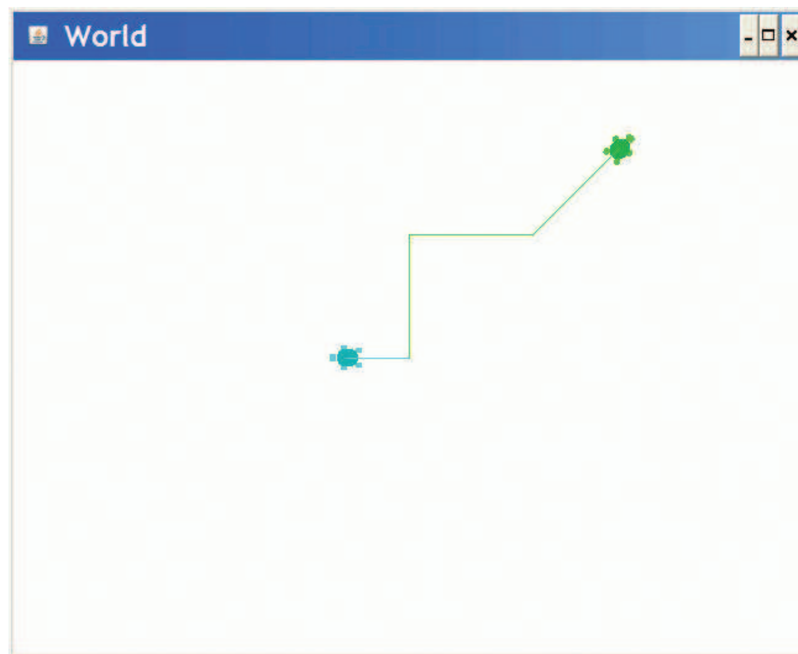

FIGURE 16.4
Turning a specified amount (−45).

forward(50). Turtles know how to turn a given number of degrees as well. Positive amounts turn the turtle to the right and negative amounts to the left (Figure 16.4).

```
>>> tina.turn(-45)
.>> tina.forward()
```

16.2.3 Objects Control Their State

In object-oriented programming we send messages to ask objects to do things. The objects can refuse to do what you ask. An object *should* refuse if you ask it to do something that would cause its data to be wrong. The world that the turtles are in is 640 pixels wide by 480 high. What happens if you try to tell the turtle to go past the end of the world?

```
>>> world1 = makeWorld()
>>> turtle1 = makeTurtle(world1)
>>> turtle1.forward(400)
>>> print turtle1
No name turtle at 320, 0 heading 0.0.
```

Turtles are first positioned at (320, 240) heading north (up). In the world the top left position is (0, 0) and x increases to the right and y increases going down. By asking the turtle to go forward 400, we are asking it to go to (320, 240 −400) which would result in a position of (320, −160) But, the turtle refuses to leave the world and instead

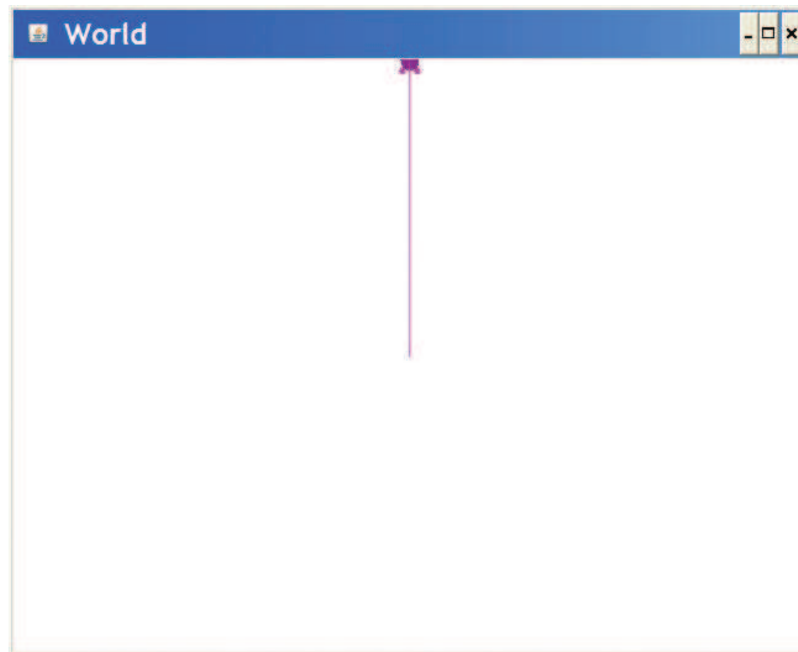

FIGURE 16.5
A turtle stuck at the edge of the world.

stops when the center of the turtle is at (320, 0) (Figure 16.5). This means we won't lose sight of any of our turtles.

The point of this exercise is to show how methods control access to the object's data. If you do not want variables to have certain values in its data, you control that through the methods. The methods serve as the gateway to and gatekeeper for the object's data.

Turtles can do lots of other things as well as go forward and turn. As you have probably noticed when the turtles move they draw a line that is the same color as the turtle. You can ask the turtle to pick up the pen using `penUp()`. You can ask the turtle to put down the pen using `penDown()`. You can ask the turtle to move to a particular position using `moveTo(x,y)`. If the pen is down when you ask the turtle to move to a new position, the turtle will draw a line from the old position to the new position (Figure 16.6).

```
>>> worldX = makeWorld()
>>> turtleX = makeTurtle(worldX)
>>> turtleX.penUp()
>>> turtleX.moveTo(0,0)
>>> turtleX.penDown()
>>> turtleX.moveTo(639,479)
```

You can change the color of a turtle using `setColor(color)`. You can stop drawing the turtle using `setVisible(false)`. You can change the width of the pen using `setPenWidth(width)`.

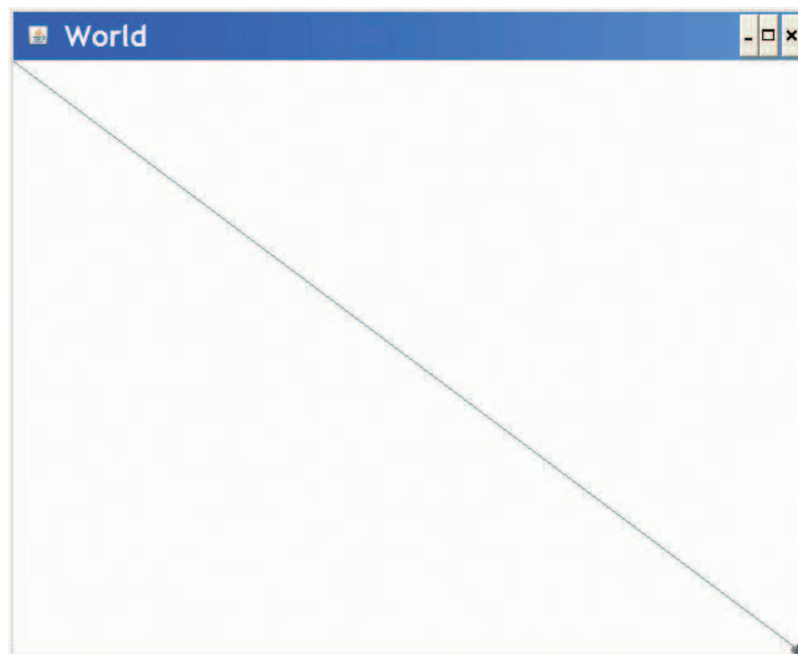

FIGURE 16.6
Using the turtle to draw a diagonal line.

16.3 TEACHING TURTLES NEW TRICKS

We have already defined a `Turtle` class for you. But, what if you want to create your own type of turtle and teach it to do new things? We can create a new type of turtle that will understand how to do all the things that turtle knows how to do, and we can also add some new functionality. This is called creating a *subclass*. Just like children inherit eye color from their parents, our subclass will *inherit* all the things that turtles know and can do. A subclass is also called a *child* class and the class that it inherits from is called the `parent` class or `superclass`.

We call our subclass `SmartTurtle`. We add a *method* that allows our turtle to draw a square. Methods are defined just like functions, but they are inside the class. Methods in Python *always* take as input a reference to the object of the class that the method is called on (usually called `self`). To draw a square our turtle will turn right and go forward 4 times. Notice that we inherit the ability to turn right and go forward from the `Turtle` class.

Program 147: Defining a Subclass

```
class SmartTurtle(Turtle):

  def drawSquare(self):
    for i in range(0,4):
      self.turnRight()
      self.forward()
```

Since the `SmartTurtle` is a kind of `Turtle`, we can use it in much the same way. But, we will need to create the `SmartTurtle` in a new way. We have been using `makePicture`, `makeSound`, `makeWorld`, and `makeTurtle` to make our objects. These are functions we have created to make it easier to make these objects. But, the actual way in Python to create a new object is to use `ClassName(parameterList)`. To create a world you can use `worldObj = World()` and to create a `SmartTurtle` you can use `turtleObj = SmartTurtle(worldObj)`.

```
>>> earth = World()
>>> smarty = SmartTurtle(earth)
>>> smarty.drawSquare()
```

Our `SmartTurtle` now knows how to draw a square (Figure 16.7). But, it can only draw squares of size 100. It would be nice to be able to draw different size squares. We can add another function that takes a parameter that specifies the width of the square.

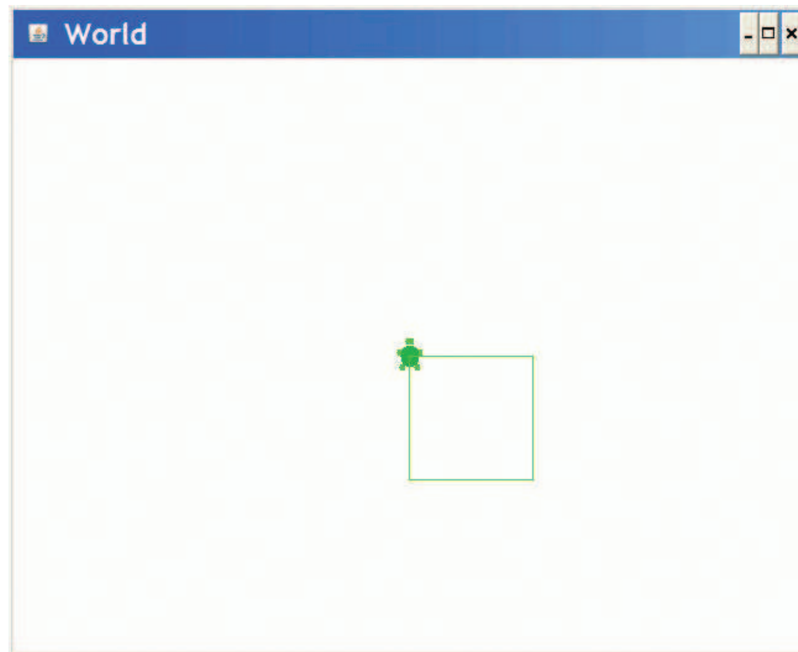

FIGURE 16.7
Drawing a square with our SmartTurtle.

Program 148: Defining a Subclass

```
class SmartTurtle(Turtle):

  def drawSquare(self):
    for i in range(0,4):
```

```
      self.turnRight()
      self.forward()

  def drawSquare(self, width):
    for i in range(0,4):
      self.turnRight()
      self.forward(width)
```

You can use this to draw different size squares (Figure 16.8).

```
>>> mars = World()
>>> tina = SmartTurtle(mars)
>>> tina.drawSquare(30)
>>> tina.drawSquare(150)
>>> tina.drawSquare(100)
```

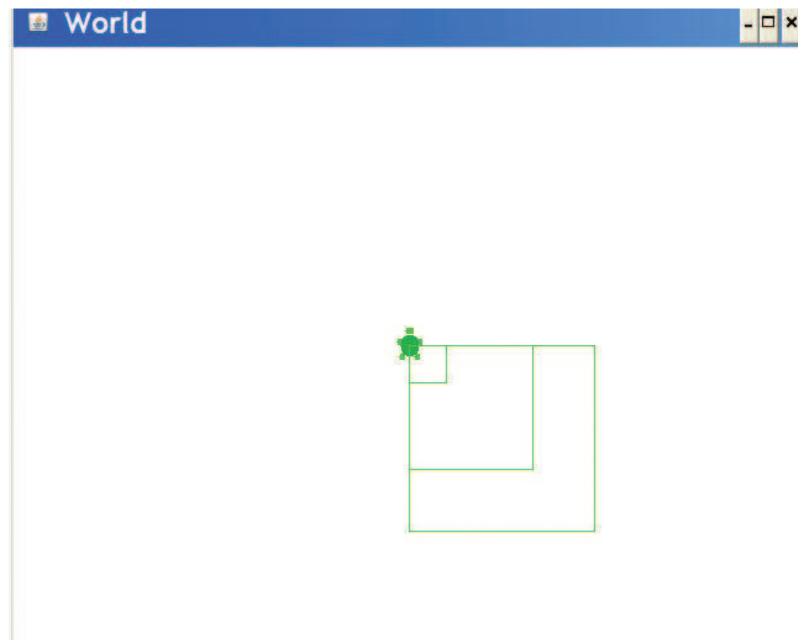

FIGURE 16.8
Drawing different size squares.

16.3.1 Overriding an Existing Turtle Method

A subclass can redefine a method that already exists in the superclass. You might do this to create a specialized form of the existing method.

Here's the class `ConfusedTurtle`, which redefines `forward` and `turn` so that it does the `Turtle` class's `forward` and `turn`, but by a random amount. You use it just

like a normal turtle—but it won't go forward or turn as much as you request. The below example will have `goofy` go forward not-quite-100 and turn nowhere-near-90.

```
>>> pluto = World()
>>> goofy = ConfusedTurtle(pluto)
>>> goofy.forward(100)
>>> goofy.turn(90)
```

Program 149: `ConfusedTurtle`, Which Goes and Turns a Random Amount

```
import random
class ConfusedTurtle(Turtle):
    def forward(self,num):
      Turtle.forward(self,int(num*random.random()))
    def turn(self,num):
      Turtle.turn(self,int(num*random.random()))
```

How It Works

We declare the class `ConfusedTurtle` to be a subclass of `Turtle`. We define two methods in `ConfusedTurtle`: `forward` and `turn`. Like any other method, they take `self` and whatever the method input is. In these cases, the input to both is a number, `num`.

What we want to do is to call the *superclass* (i.e., `Turtle`) and have it do the normal `forward` and `turn`, but with the input multiplied by a random number. Each method's body is only a single line, but it's a fairly complicated line.

- We have to tell Python explicitly to call `Turtle`'s `forward`.
- We have to pass in `self`, so that the right object's data gets used and updated.
- We multiply the input `num` by `random.random()`, but we need to convert it to an integer (using `int`). The random number returned will be between 0 and 1 (a floating-point number), but we need an integer for `forward` and `turn`.

16.3.2 Using Turtles for More

Turtles have a bunch of methods that allow for interesting graphical effects. For example, turtles are aware of each other. They can `turnToFace(anotherTurtle)` to change the heading so that the turtle is "facing" another turtle (so that if keeps going forward, it will reach the other turtle). In the below example, we set up four turtles (`al, bo, cy`, and `di`) in four corners of a square, then repeatedly have them move toward the one on the left. The result is Figure 16.9.

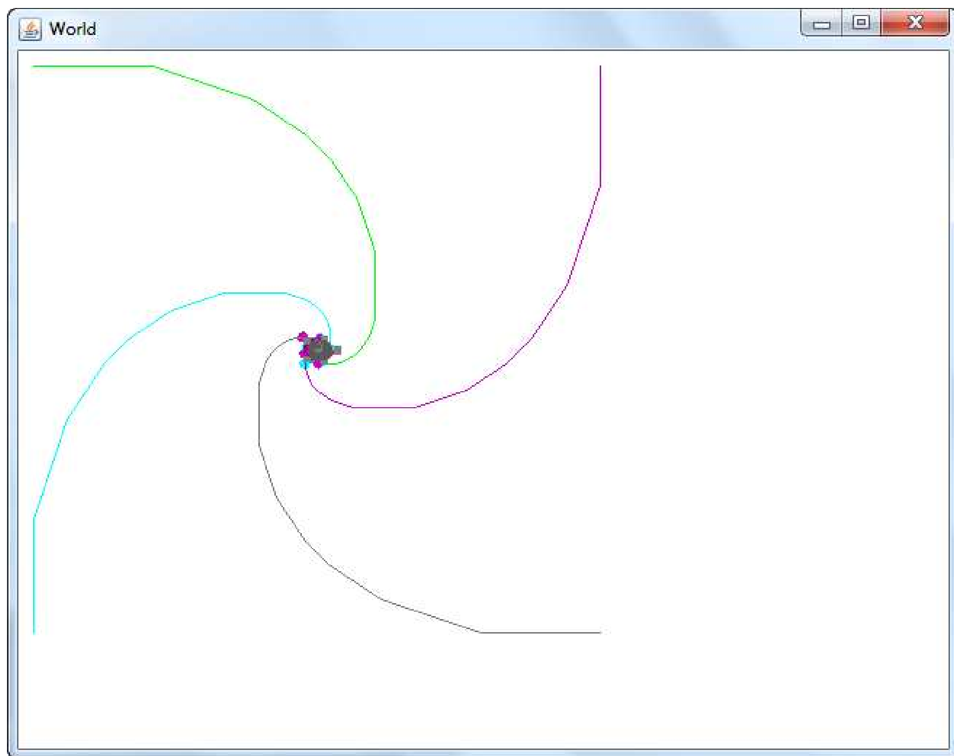

FIGURE 16.9
Four turtles chasing each other.

Program 150: Chase Turtles

```
def chase():
  # Set up the four turtles
  earth = World()
  al = Turtle(earth)
  bo = Turtle(earth)
  cy = Turtle(earth)
  di = Turtle(earth)
  al.penUp()
  al.moveTo(10,10)
  al.penDown()
  bo.penUp()
  bo.moveTo(10,400)
  bo.penDown()
  cy.penUp()
  cy.moveTo(400,10)
  cy.penDown()
  di.penUp()
  di.moveTo(400,400)
```

```
  di.penDown()
  # Now, chase for 300 steps
  for i in range(0,300):
    chaseTurtle(al,cy)
    chaseTurtle(cy,di)
    chaseTurtle(di,bo)
    chaseTurtle(bo,al)

def chaseTurtle(t1,t2):
  t1.turnToFace(t2)
  t1.forward(4)
```

How It Works

The main function here is `chase()`. The first few lines create a world and the four turtles, and place each of them in the corners (10, 10), (10, 400), (400, 400), and (400, 10). For 300 steps (a relatively arbitrary number), each turtle is told to "chase" (`chaseTurtle`) the one next to it, clockwise. So, the turtle that starts at (10, 10) (`al`) is told to chase the turtle that starts at (10, 400) (`cy`). To chase means that the first turtle turns to face the second turtle, then moves forward four steps. (Try different values—we liked the visual effect of 4 the most.) Eventually, the turtles spiral in to the center.

These functions are valuable for creating *simulations*. Imagine that we had brown turtles to act as deer, and gray turtles to act as wolves. Wolves would `turnToFace` deer when they saw them, and chase them. To run away, deer might `turnToFace` an oncoming wolf, then turn 180 and run away. Simulations are among the most powerful and insight-providing uses of computers.

Computer Science Idea: Parameters Work a Bit Differently with Objects

When you call a function and pass in a number as an input, the parameter variable (the local variable that accepts the input) essentially gets a *copy* of the number. Changing the local variable does not change the input variable.

Look at the function `chaseTurtle`. When we call the function with `chaseTurtle(al,cy)`, we *do* change the position and heading of the turtle whose name is `al`. Why is it so different? It isn't really. The variable `al` doesn't actually hold a turtle—it holds a *reference* to a turtle. Think of it as an address (in memory) of where the turtle object can be found. If you make a copy of an address, the address still references the same place. The same turtle is being manipulated inside and outside the function. We still can't make `al` reference a new object from within a function like `chaseTurtle.` We can only change the object that `al` references.

Turtles also know how to `drop` pictures. When a turtle drops a picture, the turtle stays at the upper-left-hand corner of the picture—at whatever heading the turtle is facing. (See Figure 16.10).

```
>>> # I chose Barbara.jpg for this
>>> p=makePicture(pickAFile())
>>> # Notice that we make the World and Turtle here
>>> earth=World()
```

```
>>> turtle=Turtle(earth)
>>> turtle.drop(p)
```

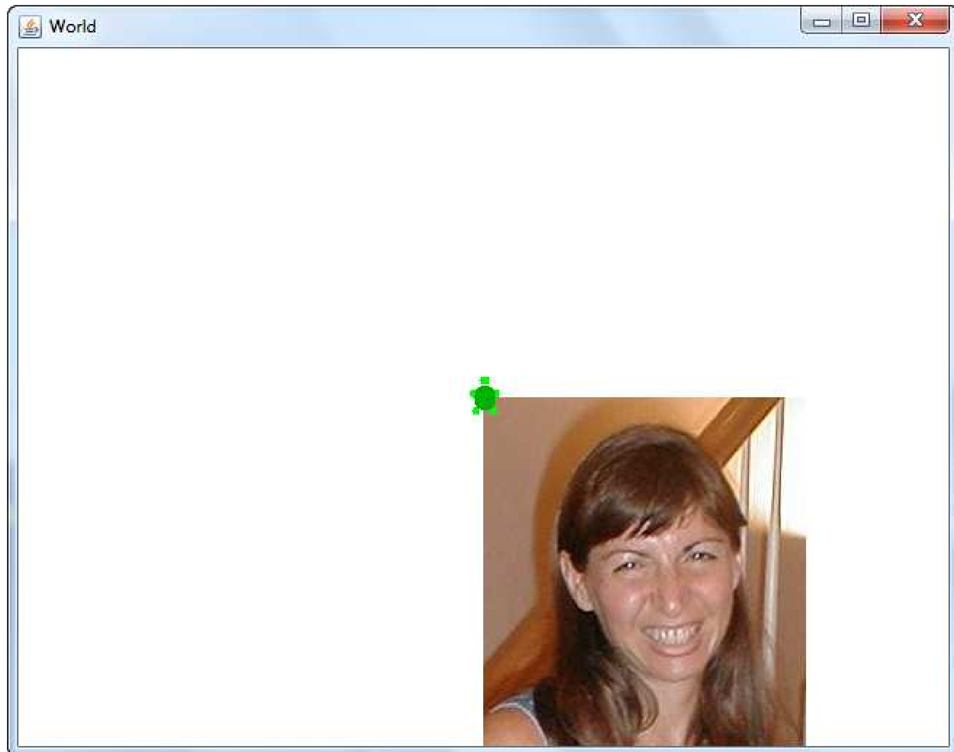

FIGURE 16.10
Dropping a picture on a world.

Turtles can also be placed on pictures, as well as World instances. When you put a turtle on a picture, its body doesn't show up by default (though you can make it visible) so that it doesn't mess up the picture. The pen is down, and you can still draw. Putting a turtle on a picture means that we can create interesting graphics on top of existing pictures or use existing pictures in your turtle manipulations.

One of our favorite techniques is spinning a picture: Have the turtle move a little, turn a little, drop a copy of the picture, then keep going. Here's an example Figure 16.11. Below is the code that made the picture. We called it with the same picture of Barb from the previous example, `show(spinAPicture(p))`.

Program 151: Spinning a Picture by Dropping It from a Turning Turtle

```
def spinAPicture(apic):
  canvas = makeEmptyPicture(640,480)
  ted = Turtle(canvas)
  for i in range(0,360):
```

```
        ted.drop(apic)
        ted.forward(10)
        ted.turn(20)
    return canvas
```

FIGURE 16.11
Dropping a picture on a picture, while moving and turning.

16.4 AN OBJECT-ORIENTED SLIDE SHOW

Let's use object-oriented techniques to build a slide show. Let's say that we want to show a picture, then play a corresponding sound and wait until the sound is done before going on to the next picture. We'll use the function (mentioned many chapters ago) *blockingPlay()*, which plays a sound and waits for it to finish before executing the next statement.

Program 152: Slide Show As One Big Function

```
def playSlideShow():
    pic = makePicture(getMediaPath("barbara.jpg"))
    sound = makeSound(getMediaPath("bassoon-c4.wav"))
```

```
show(pic)
blockingPlay(sound)
pic = makePicture(getMediaPath("beach.jpg"))
sound = makeSound(getMediaPath("bassoon-e4.wav"))
show(pic)
blockingPlay(sound)
pic = makePicture(getMediaPath("church.jpg"))
sound = makeSound(getMediaPath("bassoon-g4.wav"))
show(pic)
blockingPlay(sound)
pic = makePicture(getMediaPath("jungle2.jpg"))
sound = makeSound(getMediaPath("bassoon-c4.wav"))
show(pic)
blockingPlay(sound)
```

This isn't a very good program from any perspective. From a procedural programming perspective, there's an awful lot of duplicated code here. It would be nice to get rid of it. From an object-oriented programming perspective, we should have slide objects.

As we mentioned, objects have two parts. Objects *know* things—these become *instance variables*. Objects can *do* things—these become *methods*. We're going to access both of these using dot notation.

So what does a slide know? It knows its *picture* and its *sound*. What can a slide do? It can *show* itself, by showing its picture and playing its sound.

To define a slide object in Python (and many other object-oriented programming languages, including Java and C++), we must define a `Slide` **class**. We have already seen a couple of class definitions. Let's go through it again, slowly, building a class from scratch.

As we have already seen, a class defines the instance variables and methods for a set of objects—that is, what each object of that class knows and can do. Each object of the class is an *instance* of the class. We'll make multiple slides by making multiple instances of the `Slide` class. This is aggregation: collections of objects, just as our bodies might make multiple kidney cells or multiple heart cells, each of which knows how to do certain kinds of tasks.

To create a class in Python, we start with:

```
class Slide:
```

What comes after this, indented, are the methods for creating new slides and playing slides. Let's add a *show()* method to our `Slide` class.

```
class Slide:
  def show(self):
      show(self.picture)
      blockingPlay(self.sound)
```

To create new instances, we call the class name like a function. We can define new instance variables by simply assigning them. So here is how to create a slide and give it a picture and sound.

```
>>> slide1=Slide()
>>> slide1.picture = makePicture(getMediaPath("barbara.jpg"))
>>> slide1.sound = makeSound(getMediaPath("bassoon-c4.wav"))
>>> slide1.show()
```

The `slide1.show()` function shows the picture and plays the sound. What is this `self` stuff? When we execute `object.method()`, Python finds the method in the object's class, then calls it, using the instance object as an *input*. It's Python style to name this input variable `self` (because it is the object itself). Since we have the object in the variable `self`, we can then access its picture and sound by saying `self.picture` and `self.sound`.

But this is still pretty hard to use if we have to set up all the variables from the Command Area. How could we make it easier? What if we could pass in the sound and picture for the slides as *inputs* to the `Slide` class, as if the class were a real function? We can do this by defining something called a **constructor**.

To create new instances with some inputs, we must define a function named `__init__`. That's "underscore-underscore-i-n-i-t-underscore-underscore." It's the predefined name in Python for a method that *initializes* new objects. Our `__init__` method needs three inputs: the instance itself (because all methods get that), a picture, and a sound.

Program 153: A `Slide` Class

```
class Slide:
  def __init__(self, pictureFile,soundFile):
      self.picture = makePicture(pictureFile)
      self.sound = makeSound(soundFile)

  def show(self):
      show(self.picture)
      blockingPlay(self.sound)
```

We can use our `Slide` class to define a slide show like this.

Program 154: Playing a Slide Show, Using Our `Slide` Class

```
def playSlideShow2():
    pictF = getMediaPath("barbara.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide1 = Slide(pictF,soundF)
    pictF = getMediaPath("beach.jpg")
    soundF = getMediaPath("bassoon-e4.wav")
    slide2 = Slide(pictF,soundF)
    pictF = getMediaPath("church.jpg")
    soundF = getMediaPath("bassoon-g4.wav")
    slide3 = Slide(pictF,soundF)
    pictF = getMediaPath("jungle2.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
```

```
    slide4 = Slide(pictF,soundF)
    slide1.show()
    slide2.show()
    slide3.show()
    slide4.show()
```

One of the features of Python that make it so powerful is that we can mix object-oriented and functional programming styles. Slides are now objects that can easily be stored in lists, like any other kind of Python object. Here's an example of the same slide show where we use `map` to show the slide show.

Program 155: Slide Show, In Objects and Functions

```
def showSlide(aSlide):
  aSlide.show()

def playSlideShow3():
    pictF = getMediaPath("barbara.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide1 = Slide(pictF,soundF )
    pictF = getMediaPath("beach.jpg")
    soundF = getMediaPath("bassoon-e4.wav")
    slide2 = Slide(pictF,soundF)
    pictF = getMediaPath("church.jpg")
    soundF = getMediaPath("bassoon-g4.wav")
    slide3 = Slide(pictF,soundF)
    pictF = getMediaPath("jungle2.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide4 = Slide(pictF,soundF)

    map(showSlide,[slide1,slide2,slide3,slide4])
```

Is the object-oriented version of the slide show easier to write? It certainly has less replication of code. It features **encapsulation** in that the data and behavior of the object are defined in one and only one place, so that any change to one is easily changed in the other. Being able to use lots of objects (like lists of objects) is called **aggregation**. This is a powerful idea. We don't always have to define new classes—we can often use the powerful structures we know, like lists with existing objects, to great impact.

16.4.1 Making the `Slide` Class More Object-Oriented

What happens if we need to change the picture or sound of some class? We can. We can simply change the `picture` or `sound` instance variables. But if you think about it, you realize that that's not very safe. What if someone else used the slide show and decided to store movies in the `picture` variable? It could easily be made to work, but now we have two different uses for the same variable.

What you really want is to have a method that handles getting or setting a variable. And if it becomes an issue that the wrong data is being stored in the variable, the set-the-variable method can be changed to check the value, to make sure it's the right type and valid, before setting the variable. In order for this to work, *everyone* that uses the class has to agree to use the methods for getting and setting the instance variables, and *not* directly mess with the instance variables. In languages such as Java, one can ask the compiler to keep instance variables `private` and do not allow any uses that directly touch the instance variables. In Python, the best we can do is to create the setting-and-getting methods and encourage their use only.

We call those methods (simply enough) *setters and getters*. Here is a version of the class where we define setters and getters for the two instance variables—as you can see, they are quite simple. Notice how we change the `show` and even the `__init__` methods so that, as much as possible, we use the setters and getters instead of direct access of the instance variables. This is the style of programming that Adele Goldberg meant when she talked about, "Ask, don't touch."

Program 156: Class `Slide` with Getters and Setters

```
class Slide:
  def __init__(self, pictureFile,soundFile):
      self.setPicture(makePicture(pictureFile))
      self.setSound(makeSound(soundFile))

  def getPicture(self):
    return self.picture
  def getSound(self):
    return self.sound

  def setPicture(self,newPicture):
    self.picture = newPicture
  def setSound(self,newSound):
    self.sound = newSound

  def show(self):
      show(self.getPicture())
      blockingPlay(self.getSound())
```

Here is something cool about our revised class. We don't have to change *anything* in our `playSlideShow3` function. It just works, still, even though we made several changes to how the class `Slide` works. We say that the function `playSlideShow3` and the class `Slide` are *loosely coupled*. They work together, in well-defined ways, but the inner workings of either can change without impacting the other.

16.5 OBJECT-ORIENTED MEDIA

As we said, we have been using objects throughout this book. We have been creating `Picture` objects with the function `makePicture`. We can also create a picture using the normal Python constructor.

```
>>> pic=Picture(getMediaPath("barbara.jpg"))
>>> pic.show()
```

Here's how the *function* `show()` is defined. You can ignore `raise` and `__class__`. The key point is that the function is simply executing the existing picture method `show`.

```
def show(picture):
    if not picture.__class__ == Picture:
        print "show(picture): Input is not a picture"
        raise ValueError
    picture.show()
```

We could have other classes that also know how to `show`. Objects can have their own methods with names that other objects also use. Much more powerful is that each of these methods with the same name can achieve the same *goal*, but in different ways. We defined a class for slides, and it knew how to `show`. For both slides and pictures, the method `show()` says, "Show the object." But what's really happening is different in each case: pictures just show themselves, but slides show their pictures and play their sounds.

Computer Science Idea: Polymorphism

When the same name can be used to invoke different methods that achieve the same goal, we call that **polymorphism**. It's very powerful for the programmer. You simply tell an object show()—you don't have to care exactly what method is being executed and you don't even have to know exactly what object it is that you're telling the object to show. You the programmer simply specify your *goal*, to show the object. The object-oriented program handles the rest.

There are several examples of polymorphism built into the methods that we're using in JES.[1] For example, both pixels and colors understand the methods `setRed`, `getRed`, `setBlue`, `getBlue`, `setGreen`, and `getGreen`. This allows us to manipulate the colors of the pixels without pulling out the color objects separately. We could have defined the functions to take both kinds of inputs or to provide different functions for each kind of input, but both of those options get confusing. It's easy to do with methods.

```
>>> pic=Picture(getMediaPath("barbara.jpg"))
>>> pic.show()
>>> pixel = pic.getPixel(100,200)
>>> print pixel.getRed()
73
```

[1]Recall that JES is an environment for programming in Jython, which is a specific kind of Python. The media supports are part of what JES provides—they're not part of the core of Python.

```
>>> color = pixel.getColor()
>>> print color.getRed()
73
```

Another example is the method `writeTo()`. The method `writeTo(filename)` is defined for both pictures and sounds. Did you ever confuse `writePictureTo()` and `writeSoundTo()`? Isn't it easier to just always write `writeTo(filename)`? That's why that method is named the same in both classes, and why polymorphism is so powerful. (You may be wondering why we didn't introduce this in the first place. Were you ready in Chapter 2 to talk about dot notation and polymorphic methods?)

Overall, there are actually many more methods defined in JES than functions. More specifically, there are a bunch of methods for drawing on pictures that aren't available as functions.

- As you would expect, pictures understand `pic.addRect(color,x,y,width,height)`, `pic.addRectFilled(color,x,y,width,height)`, `pic.addOval(color,x,y,width,height)`, and `pic.addOvalFilled(color,x,y,width,height)`.

  See Figure 16.12 for examples of rectangle methods drawn from the following example.

  ```
  >>> pic=Picture (getMediaPath("640x480.jpg"))
  >>> pic.addRectFilled (orange,10,10,100,100)
  >>> pic.addRect (blue,200,200,50,50)
  >>> pic.show()
  >>> pic.writeTo("newrects.jpg")
  ```

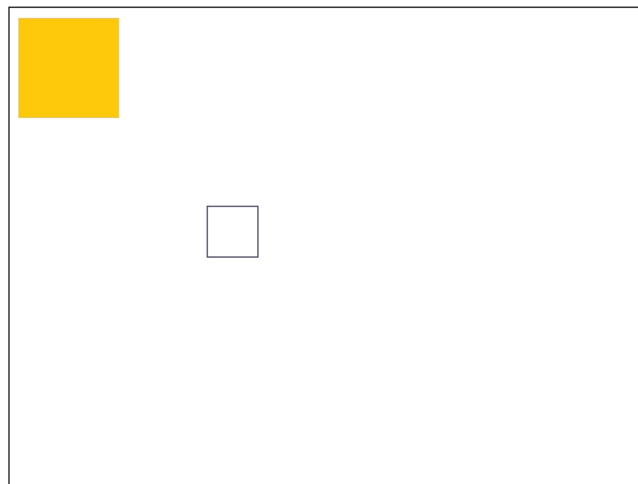

FIGURE 16.12
Examples of rectangle methods.

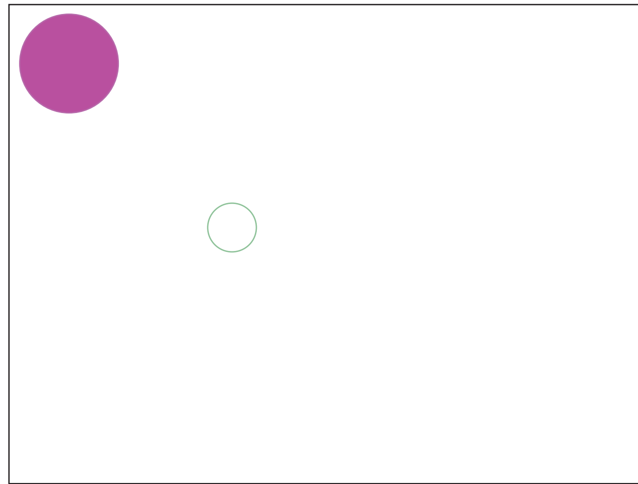

FIGURE 16.13
Examples of oval methods.

See Figure 16.13 for examples of ovals drawn from the following example.

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addOval (green,200,200,50,50)
>>> pic.addOvalFilled (magenta,10,10,100,100)
>>> pic.show()
>>> pic.writeTo("ovals.jpg")
```

- Pictures also understand *arcs*. Arcs are literally parts of a circle. The two methods are `pic.addArc(color,x,y,width,height,startAngle,arcAngle)` and `pic.addArcFilled(color,x,y,width,height,startAngle,arcAngle)`. They draw arcs for `arcAngle` degrees, where `startAngle` is the starting point. 0 degrees is at 3 o'clock on the clock face. A positive arc is counter clockwise and negative is clockwise. The center of the circle is the middle of the rectangle defined by (x, y) with the given `width` and `height`.
- We can also now draw colored lines, using `pic.addLine(color,x1,y1,x2,y2)`.

  See Figure 16.14 for examples of arcs and lines drawn from the following example.

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addArc(red,10,10,100,100,5,45)
>>> pic.show()
>>> pic.addArcFilled (green,200,100,200,100,1,90)
>>> pic.repaint()
>>> pic.addLine(blue,400,400,600,400)
>>> pic.repaint()
>>> pic.writeTo("arcs-lines.jpg")
```

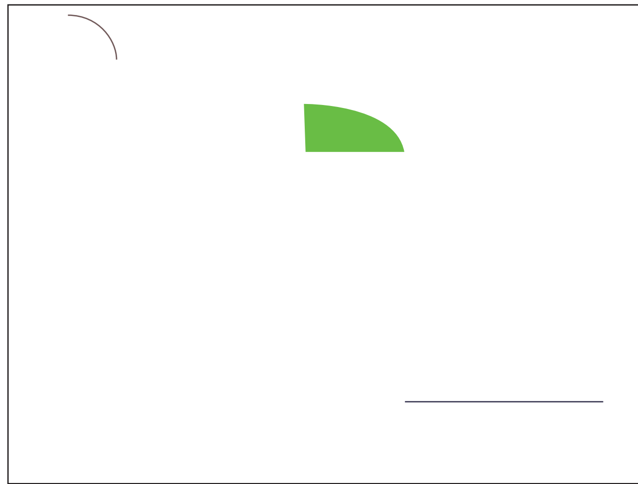

FIGURE 16.14
Examples of arc methods.

- Text in Java can have styles, but these are limited to make sure that all platforms can replicate them. `pic.addText(color,x,y,string)` is the one we would expect to see. There is also `pic.addTextWithStyle(color,x,y,string,style)`, which takes a style created from `makeStyle(font,emphasis,size)`. The `font` is `sansSerif`, `serif`, or `mono`. The `emphasis` is `italic`, `bold`, or `plain`, or sum them to get combinations (e.g., `italic+bold`.`size` is a point size).

  See Figure 16.15 for examples of text drawn from the following example.

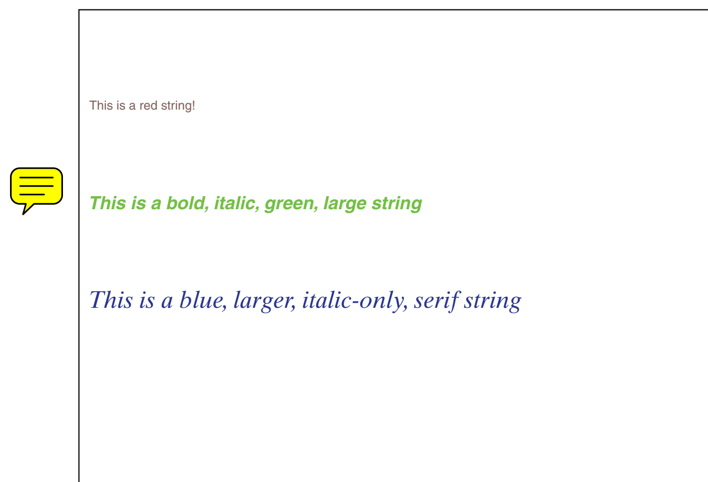

FIGURE 16.15
Examples of text methods.

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addText(red,10,100,"This is a red
    string!")
>>> pic.addTextWithStyle (green,10,200,"This is a
    bold, italic, green, large string",
    makeStyle(sansSerif, bold+italic,18))
>>> pic.addTextWithStyle (blue,10,300,"This is a
    blue, larger, italic-only, serif string",
    makeStyle(serif, italic,24))
>>> pic.writeTo("text.jpg")
```

The older media functions that we wrote can be rewritten in method form. We will need to create a subclass of the Picture class and add the method to that class.

Program 157: Making a Sunset Using a Method

```
class MyPicture(Picture):
  def makeSunset(self):
    for p in getPixels(self):
      p.setBlue(int(p.getBlue()*0.7))
      p.setGreen(int (p.getGreen()*0.7))
```

This can be used like this.

```
>>> pict = MyPicture(getMediaPath("beach.jpg"))
>>> pict.explore()
>>> pict.makeSunset()
>>> pict.explore()
```

We can also create new subclasses of the Sound class and new methods to work on sound objects. The methods for accessing sound sample values are getSampleValue() and getSampleValueAt(index).

Program 158: Reverse a Sound with a Method

```
class MySound(Sound):
  def reverse(self):
    target = Sound(self.getLength())
    sourceIndex = self.getLength() - 1
    for targetIndex in range(0,target.getLength()):
      sourceValue = self.getSampleValueAt(sourceIndex)
      target.setSampleValueAt(targetIndex,sourceValue)
      sourceIndex = sourceIndex - 1
    return target
```

This can be used like this.

```
>>> sound = MySound(getMediaPath("always.wav"))
>>> sound.explore()
>>> target = sound.reverse()
>>> target.explore()
```

16.6 JOE THE BOX

The earliest example used to teach object-oriented programming was developed by Adele Goldberg and Alan Kay. It's called *Joe the Box*. There is nothing new in this example, but it does provide a *different* example from another perspective, so it's worth reviewing.

Imagine that you have a class `Box` like the one below:

```
class Box:
  def __init__(self):
    self.setDefaultColor()
    self.size=10
    self.position=(10,10)
  def setDefaultColor(self):
    self.color = red
  def draw(self,canvas):
    addRectFilled(canvas, self.position[0],self.
    position[1], self.size, self.size, self.color)
```

What will you see if you execute the following code?

```
>>> canvas = makeEmptyPicture(400,200)
>>> joe = Box()
>>> joe.draw(canvas)
>>> show(canvas)
```

Let's trace it out.

- Obviously, the first line just creates a white `canvas` that is 400 pixels wide and 200 pixels high.
- When we create `joe`, the `__init__` method is called. The method `setDefaultColor` is called on `joe`, so he gets a default color of red. When `self.color=red` is executed, the *instance variable* `color` is created for `joe` and gets a value of red. We return to `__init__`, where joe is given a size of 10 and a position of (10,10) (`size` and `position` both become new instance variables).
- When `joe` is asked to `draw` himself on the `canvas`, he's drawn as a red, filled rectangle (`addRectFilled`), at x position 10 and y position 10, with a size of 10 pixels on each side.

We could add a method to `Box` that allows us to make `joe` change his size.

```
class Box:
  def __init__(self):
    self.setDefaultColor()
    self.size=10
    self.position=(10,10)
  def setDefaultColor(self):
    self.color = red
  def draw(self,canvas):
    addRectFilled(canvas, self.position[0],self.
```

M01_GUZD3514_03_SE_C16 30/1/2012 19:27 Page 400

```
    position[1], self.size, self.size, self.color)
  def grow(self,size):
    self.size=self.size+size
```

Now we can tell joe to grow. A negative number like −2 will cause joe to shrink. A positive number will cause joe to grow—though we'd have to add a move method if we wanted him to grow much and still fit on the canvas.

Now consider the following code added to the same Program Area.

```
class SadBox(Box):
  def setDefaultColor(self):
    self.color=blue
```

Note that SadBox lists Box as a superclass (parent class). This means that SadBox *inherits* all the methods of Box. What will you see if you execute the code below?

```
>>> jane = SadBox()
>>> jane.draw(canvas)
>>> repaint(canvas)
```

Let's trace it out:

- When jane is created as a SadBox, the method __init__ is executed in class Box.
- The first thing that happens in __init__ is that we call setDefaultColor *on the input object* self. That object is now jane. So we call jane's setDefaultColor. We say that SadBox's setDefaultColor *overrides* Box's.
- The setDefaultColor for jane sets the color to blue.
- We then return to executing the rest of Box's __init__. We set jane's size to 10 and position to (10,10).
- When we tell jane to draw, she appears as a 10 × 10 blue square at position (10,10). If we haven't moved or grown joe, he will disappear as jane is drawn on top of him.

Note that joe and jane are each a different *kind* of Box. They have the same instance variables (but different *values* for the same variables) and mostly know the same things. Because both understand draw, for example, we say that draw is *polymorphic*. The word *polymorphic* just means many forms.

A SadBox (jane) is slightly different in how it behaves when it created, so it knows some things differently. Joe and Jane highlight some of the basic ideas of object-oriented programming: inheritance, specialization in subclasses, and shared instance variables while having different instance variable values.

16.7 WHY OBJECTS?

One role for objects is to reduce the number of names that you have to remember. Through polymorphism, you only have to remember the name and the goal, not all the various global functions.

More importantly, though, objects encapsulate data and behavior. Imagine that you wanted to change the name of an instance variable and then all the methods that use the variable. That's a lot to change. What if you miss one? Changing them all in one place, together, is useful.

Objects reduce the *coupling* between program components, that is, how dependent they are on each other. Imagine that you have several functions that all use the same global variable. If you change one function so that it stores something slightly different in that variable, all the other functions must also be updated or they won't work. That's called *tight* coupling. Objects that only use methods on each other (no direct access to instance variables) are more *loosely* coupled. The access is well-defined and easily changed in only one place. Changes in one object do not demand changes in other objects.

An advantage of loose coupling is ease in developing in team contexts. You can have different people working on different classes. As long as everyone agrees on how access will work through methods, nobody has to know how anybody else's methods work. Object-oriented programming can be particularly useful when working on teams.

Aggregation is also a significant benefit of object systems. You can have lots of objects doing useful things. Want more? Just create them!

Python's objects are similar to the objects of many languages. One significant difference is in access to instance variables, though. In Python, any object can access and manipulate any other object's instance variables. That's not true in languages like Java, C++, or Smalltalk. In these other languages, access to instance variables from other objects is limited and can even be eliminated entirely—then you can only access objects' instance variables through getter and setter methods.

Another big part of object systems is **inheritance**. As we saw with our turtle and box examples, we can declare one class (*parent class*) to be *inherited* by another class (*child class*) (also called superclass and subclass). Inheritance provides for instant polymorphism—the instances of the child automatically have all the data and behavior of the parent class. The child can then add more behavior and data to what the parent class had. This is called making the child a *specialization* of the parent class. For example, a 3-D rectangle instance might know and do everything that a rectangle instance does by saying `class Rectangle3D(Rectangle)`.

Inheritance gets a lot of press in the object-oriented world but it's a trade-off. It reduces even further the duplication of code, which is a good thing. In actual practice, inheritance isn't used as much as other advantages of object-oriented programming (like aggregation and encapsulation), and it can be confusing. Whose method is being executed when you type the below? It's invisible from here, and if it's *wrong*, it can be hard to figure out where it's wrong.

```
myBox = Rectangle3D()
myBox.draw()
```

So when should you use objects? You should define your own object classes when you have data and behavior that you want to define for all instances of the group (e.g., pictures and sounds). You should use existing objects *all the time*. They're very powerful. If you're not comfortable with dot notation and the ideas of objects, you

can stick with functions—they work just fine. Objects just give you a leg up on more complex systems.

PROGRAMMING SUMMARY

Some of the programming pieces that we met in this chapter.

OBJECT-ORIENTED PROGRAMMING

`class`	Lets you define a class. The keyword `class` takes a class name and an optional superclass in parentheses, ending with a colon. Methods for the class follow, indented within the class block.
`__init__`	The name of the method called on an object when it's first created. It's not required to have one.

GRAPHICS METHODS

`addRect`, `addRectFilled`	The methods in the `Picture` class for drawing rectangles and filled rectangles.
`addOval`, `addOvalFilled`	The methods in the `Picture` class for drawing ovals and filled ovals.
`addArc`, `addArcFilled`	The methods in the `Picture` class for drawing arcs and filled arcs.
`addText`, `addText-WithStyle`	The methods in the `Picture` class for drawing text and text with style elements (like boldface or sans serif).

`addLine`	The method in the `Picture` class for drawing a line.
`getRed`, `getGreen`, `getBlue`	The methods for both `Pixel` and `Color` objects for getting the red, green, and blue color components.
`setRed`, `setGreen`, `setBlue`	The methods for both `Pixel` and `Color` objects for setting the red, green, and blue color components.

PROBLEMS

16.1 Answer the following questions.

- What is the difference between an instance and a class?

- How are functions and methods different?
- How is object-oriented programming different from procedural programming?
- What is polymorphism?
- What is encapsulation?
- What is aggregation?
- What is a constructor?
- How did biological cells influence the development of the idea of objects?

16.2 Answer the following questions.

- What is inheritance?
- What is a superclass?
- What is a subclass?
- What methods does a child class inherit?
- What instance variables (fields) does a child class inherit?

16.3 Add a method to the `Turtle` class to draw an equilateral triangle.

16.4 Add a method to the `Turtle` class to draw a rectangle given a width and height.

16.5 Add a method to the `Turtle` class to draw a simple house. It can have a rectangle for the house and an equilateral triangle as the roof.

16.6 Add a method to the `Turtle` class to draw a street of houses.

16.7 Add a method to the `Turtle` class to draw a letter.

16.8 Add a method to the `Turtle` class to draw your initials.

16.9 Create a movie with several turtles moving in each frame.

16.10 Add another constructor to the `Slide` class that takes just a picture filename.

16.11 Create a `SlideShow` class that holds a list of slides and shows each slide one at a time.

16.12 Create a `CartoonPanel` class that takes an array of `Pictures` and displays the pictures from left to right. It should also have a title and author and display the title at the top left edge and the author at the top right edge.

16.13 Create a `Student` class. Each student should have a name and a picture. Add a method, `show`, that shows the picture for the student.

16.14 Add a field to the `SlideShow` class to hold the title and modify the `show` method to first show a blank picture with the title on it.

16.15 Create a `PlayList` class that takes a list of sounds and play them one at a time.

16.16 Use the methods in the `Picture` class to draw a smiling face.

16.17 Use the methods in the `Picture` class to draw a rainbow.

16.18 Rewrite the mirror functions as methods in the `MyPicture` class.

16.19 Make some modifications to Joe the Box.

- Add a method to `Box` named `setColor` that takes a color as input, then makes the input color the new color for the box. (Maybe `setDefaultColor` should call `setColor`?)
- Add a method to `Box` named `setSize` that takes a number as input, then makes the input number the new size for the box.
- Add a method to `Box` named `setPosition` that takes a list or tuple as a parameter, then makes that input the new position for the box.
- Change `__init__` so that it uses `setSize` and `setPosition` rather than simply setting the instance variables.

*16.20 Finish the Joe the Box example.

(a) Implement `grow` and `move`. The method `move` takes as input a relative distance like (−10,15) to move 10 pixels left (x position) and 15 pixels down (y position).

(b) Draw patterns by creating `joe` and `jane`, then move a little and draw, grow a little and draw, then repaint the new canvas.

16.21 Create a movie with boxes growing and shrinking in it.

TO DIG DEEPER

There is lots more to do with Python in exploring procedural, functional, and object-oriented programming styles. Mark recommends the books by Mark Lutz (especially [30]) and Richard Hightower [24] as nice introductions to the deeper realms of Python. You might also explore some of the tutorials at the Python Web site (`http://www.python.org`).