Chapter 9. Introduction to Classes

In preceding chapters, we have modeled increasingly complex phenomena. We started with simple geometric figures and moved on to more complex objects like moving balls, functions and statistical data. While the basic types and control structures covered in the previous chapters are sufficient to support models of a wide variety of object types, the complexity of our programs is starting to grow beyond our ability to keep track of everything. Increasingly realistic programs become very complex very quickly.

One abstraction that programmers use to help address this complexity is the concept of a *class*. This chapter introduces the design and implementation of classes in Processing and Java.

9.1. Example: Pinball Game

Video games commonly involve interacting objects of various kinds. Even a simple game like Pac-Man has an agent, ghosts, power pellets and bonus items. The programming task of representing, presenting and maintaining all of these objects is daunting, and its difficulty increases as the number and complexity of the objects increases. Programmers can address this complexity by adopting an object-oriented perspective in which they design and implement models of sets of objects that encapsulate shared *states* and *behaviors*.

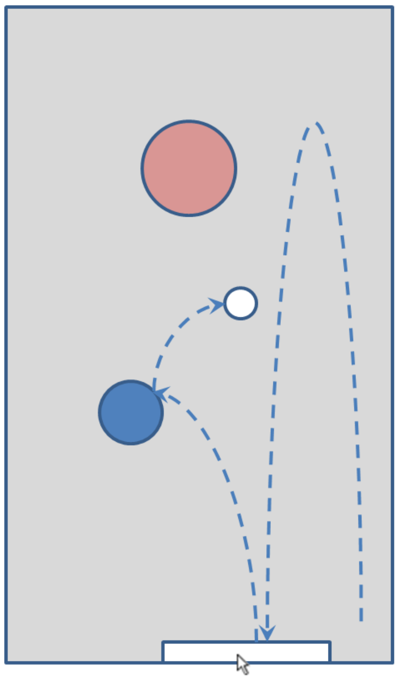


Figure 9-1. A pinball game sketch

In this chapter, our vision is to build a simple pinball video game like the one shown in Figure 9-1. This game expects the player to move the paddle, shown on the bottom, from side to side using the mouse pointer. A single white ball shoots up from the bottom right of the display window and bounces off the walls, the stationary, colored targets and the paddle. If it falls to the bottom without hitting the paddle, it falls into oblivion. We'd like to have a single paddle and multiple targets. Implementing a game such as this requires that the application model all the elements of the following objects:

- The moving ball;
- The stationary target balls;
- The movable paddle.

Each of these types of objects must encapsulate its own state and behaviors:

- The ball’s state is characterized by its a current location, velocity, acceleration, size and color; its behaviors must include movement and collision (with the targets, the paddle, the walls and ceiling);
- The targets are similar to the moving ball except that they have no velocity or acceleration;
- The paddle’s state is characterized by its current location, size and color;

We could implement all of these aspects of the game objects using techniques discussed in previous chapters, but object-oriented programming languages like Processing and Java provide a class mechanism that helps to address the complexity of these systems.

9.2. Class Design

Classes are the central element of a paradigm of computing known as *object-oriented programming*. They allow programmers to encapsulate all the elements related to a particular type of object within a single programming construct called a class. Programs that model complex phenomena, for example the interactions between the objects in a video game such as the one shown in Figure 9-1, can sub-divide their work into simpler pieces by implementing one class for each type of object.

9.2.1. Working with Objects

The world is filled with *objects*. The set of all objects includes a wide variety of individual objects where each object has a certain state and set of behaviors that make it unique. We can observe, however, that there are sub-sets of these objects that share certain common properties. These sub-sets are often called “classes.” Examples include the following:

- We divide students into classes. For example, the “freshman”, “sophomore”, “junior” and “senior” classes characterize students at different levels of schooling. Individual students in each class share certain traits, such as a maximum number of total credit hours, but also have individual characteristics that make them unique, such as their individual name and student id number;

- The U.S. Navy describes ships as belonging to certain classes. For example, the “Skipjack”, “Thresher”, and “Sturgeon” classes characterize different kinds of submarines. Individual submarines in each class share certain traits, such as a common weight displacement, but also have unique individual characteristics, such as their name and current location.

Used in this way, the word *class* is a synonym for the word *type*, because it provides a name for a set of related objects.

Because programs model the world, they must represent objects. Programmers, therefore, face the same issue of complexity that people face in real life, and they address this complexity in the same manner as people do, by dividing the objects being modeled into sub-sets. Programming languages provide types to represent these sub-sets. If the model only includes simple objects like strings or numbers, then primitive types are sufficient to build the program. However, most programs model more complex objects that cannot be directly represented with primitive types. For example, even a simple point object requires both an x and a y coordinate, and a more complex video game agent like the Pac-Man ghost requires much more (i.e., a name, color, location, shape, etc.).

Object-oriented programming languages allow programmers to define new types for the programming language by designing and building classes. Classes can be viewed as blueprints for constructing objects. These blueprints specify how to represent both the *state* of the object and the *behavior* of that object. Programmers use classes to extend the type system of the language in order to support more complex objects. This section discusses the design of classes in Java. Later sections discuss the implementation of classes.

9.2.2. Designing Class Attributes and Behaviors

Designing a class requires that we identify:

- class *attributes* – the data that must be stored in order to characterize the state of a class object;
- class *behavior* – the operations that a class object can perform.

The behavior of a class is usually identified first, because often the attributes of the class are not obvious and because identifying the class behavior can sometimes clarify the nature of the attributes. Also, if the behaviors are identified before any of the attribute details are established, they will remain independent of the details of how the attributes are implemented. This independence from implementation details is an important principle of good class design. To encourage this independence, it is helpful to distinguish two perspectives when designing classes:

- When we used pre-existing types in previous chapters, we tended to adopt an *external* perspective with respect to the types. For example, when we built and used arrays, we didn't much care how Java represented the array data structure so long as it supported the state memory and behaviors that we expect from arrays, e.g., the storage of multiple values of a common type, the length variable, the constructor (i.e., `new`) method and the subscript operator. When designing a new class, it is good to start by thinking from an external perspective about what state and behaviors the objects of that class should support.

- When we design our own types using classes, we must adopt an *internal* perspective. The internal perspective requires that we, as designers of the class, carefully consider not only the features that our class must provide but also precisely how we will implement those features. When the external perspective of the class is determined, we begin thinking from an internal perspective about how to design the internals of the class definition. It is frequently helpful to visualize ourselves as an object of the class we are defining and ask ourselves "What would I, as an object of this class, need to implement my required state and behavior?"

Consider the example of designing a class that implements a temperature. Scientific applications often represent and work with many different temperatures, all of which must be implemented consistently in a manner that allows them to be set to particular values, to specify particular scales (e.g., Celsius, Fahrenheit and Kelvin), to be translated from one scale to another and to be compared with one another. Java provides predefined types for representing floating-point values for the degrees of the temperature, but it doesn't provide any means for bundling the degrees, the scale and the behavior of even this rather unassuming temperature object. This bundling is important. For example, determining whether 40° is hot or cold depends very much on the scale in which it is specified; 40° Fahrenheit is relatively cold; 40° Celsius (104° Fahrenheit) is hot; 40° Kelvin (-387.6° Fahrenheit) is unspeakably cold.

From the external perspective, a programmer would expect a temperature object to support the following behaviors:

- Construct – Create a new temperature object;
- Change – Raise or lower the value of the temperature object;
- Convert – Convert a temperature in one scale to an equivalent temperature in another scale;
- Compare – Compare the value of one temperature object with another.

This external perspective defines the programmer's interface to the class, often called the *Application Programmer's Interface* (API). It doesn't say anything about how to implement these behaviors; programmers creating and using temperature objects should not have to know about these things any more than they should know how a **`String`** or **`PImage`** object implements its behaviors internally.

Because we are designing this temperature class ourselves, we must now switch to the internal perspective. From this perspective, I, as a temperature object, need to include all those attributes and methods required to implement the state and behaviors required by my API. Thus, I must implement the following attributes:

- Degrees – my magnitude;
- Scale – the scale in which my magnitude is specified.

The need for other attributes may become evident later in the implementation of the temperature class, but these attributes are good place to start.

9.2.3. Example Revisited

Consider the example of designing a class that implements a stationary ball object that renders itself on the display window. This is a simplified version of the stationary target balls shown in our original pinball game sketch shown in Figure 9-1. We would like our user to see each ball as a colored circle, as shown in Figure Figure 9-2. From the external perspective, a ball object should support the following behaviors:

- Constructor – Create a new ball object;
- Render – Draw the object as an ellipse based on its current state.

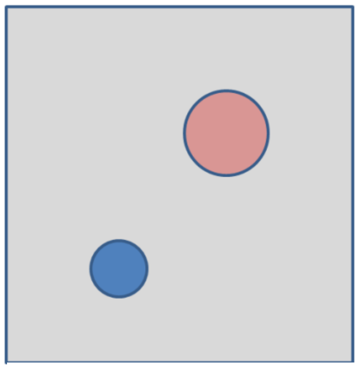


Figure 9-2. A sketch of two stationary ball objects

From the internal perspective. a ball object should implement the following attributes:

- Coordinates – my x and y coordinates;
- Dimensions – the diameter of my circle;
- Color – my color.

The class invariant that must be explicitly enforced for this application is simply that the diameter of the circle must be non-negative.

9.3. Class Implementation

In Java, classes are implemented using the following pattern.

Class Definition Pattern

```
class ClassName {
   attributeDeclarations
   methodDefinitions
}
```

- *ClassName* is the name of the new reference type being defined;
- *attributeDeclarations* declare the variables (and constants) required to store the state of the objects defined by the class, using standard variable (and constant) declaration patterns;
- *methodDefinitions* define the methods required to implement the behavior of the objects defined by the class, using the standard method definition pattern.

Class definitions are said to *encapsulate* attributes and methods in that they combine all the attributes and the methods into a single programming construct, the class definition. Classes can be viewed as blueprints for class objects that programmers can use to construct multiple objects of that same type as required by their application.

In Java, new classes are implemented in separate files where the filename matches the name of class. For example, a new **`Temperature`** class would be stored in the file **`Temperature.java`**. In Processing, new classes are implemented by adding a new .pde file for each class. The Processing IDE allows you to add a new class by adding a new editing tab to the IDE and it stores the new file in the same sketch directory as the main sketch pde file.

9.3.1. Implementing Class Attributes

Given a class design, our first task is to implement the class's attributes. For attributes that change over time or that differ from one object to another, we must define variables. Attributes that are the same for all objects and do not change can be represented as constants (or as methods that return a literal). The values of the attributes determine the state of a class object at a particular time.

For our temperature class, the attributes, degrees and scale, can differ from one temperature object to another and should therefore be stored in variables:

```
double myDegrees;
char myScale;
```

These will become the attributes — also called instance variables — of our temperature class. We give the names of attribute variables the prefix **`my`** to help indicate that they are attributes of the class and to reflect our internal perspective. As with all identifiers, the name of an attribute variable or constant should be self-documenting, describing the attribute being stored.

Encapsulating Attributes

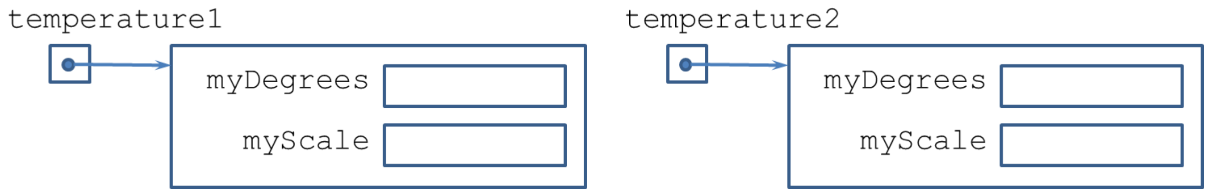
Once we have defined fields to represent the attributes of our class, we can actually create the class by wrapping these objects in a class declaration:

```
class Temperature {
   double myDegrees;
   char myScale;
}
```

This declaration creates a new type named `Temperature`. Because classes are reference types, we declare their objects using the `new` operator followed by a `Temperature` constructor. Given the following declarations:

```
Temperature temperature1 = new Temperature(),
            temperature2 = new Temperature();
```

`temperature1` and `temperature2` refer to distinct **`Temperature`** objects where each object contains two attributes: a `double` variable named `myDegrees` and a `char` variable named `myScale`. We can picture these newly-constructed objects as follows:



Wrapping the attribute variables in a class declaration and then using that class as a type to declare an object makes it possible for an object to store values of different types. In the vocabulary of programming languages, we say that class `Temperature` *encapsulates* the attribute variables `myDegrees` and `myScale`. Such encapsulation allows a single object to store multiple values of potentially different types.

Supporting Information Hiding

As the attribute variables of a `Temperature` object are currently declared, they can be accessed directly, as is also the case with an array object's length value (e.g., `myArray.length`). For example, the `main()` method of the application can check the value of `tempuerature1`'s degrees:

```
if (temperature1.myScale == 'C') // ...
```

This is problematic, however, because there is nothing protecting an attribute from being accidentally modified. For example, a programmer might assign an inappropriate value to the diameter,

```
temperature1.myScale = 'X';
```

The result would be an invalid `Temperature` object.

Such problems can be prevented by declaring the attribute variables to be `private`:

```
class Temperature {
      private double myDegrees;
      private char myScale;
}
```

This prevents direct access to the attribute variables from outside the class declaration. The default protection for attributes is `public`.

This fundamental concept of class design is called *information hiding*. By preventing a program from directly accessing the private data members of a class, we hide that information, thus removing the temptation to access the data members directly and preventing potential problems. *It is a good practice to hide all attribute variables of a class by making them* ***private****.*

Once we have the attribute variables encapsulated and hidden, we are almost ready to begin implementing the class operations.

Enforcing Class Invariants

Before defining class operations, we should identify any restrictions on the values of the attributes of our class. For example, we might stipulate that the only valid values for attribute variables `myScale` are `'C'`, `'F'` or `'K'`. Similarly, we might stipulate that the combination of values of `myDegrees` and `myScale` should never be below absolute zero (i.e., 0° Kelvin). If we identify such restrictions at the outset, then we can implement the various class operations in a way that ensures that these restrictions are enforced.

Once we have such a condition, we want to make certain that none of the code we write for this class violates it — that is, we want the condition to be true, both before and after each call of a class operation. Because this condition must be true throughout the life of all class objects, it is called a *class invariant*. When such an invariant is defined, it is good practice to record it in the documentation at the beginning of the class file and to implement measures ensuring that it is maintained.

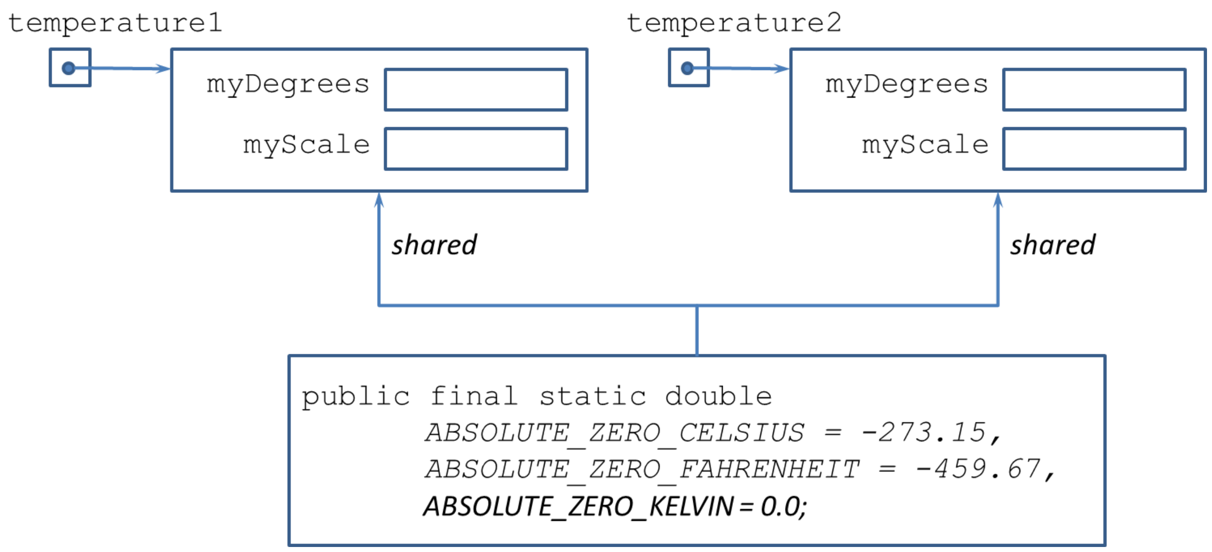
```
/**
 * This class implements a standard temperature object.
 *
 * Temperature class invariant:
 * myScale == 'C' && myDegrees >= ABSOLUTE_ZERO_CELSIUS ||
 * myScale == 'F' && myDegrees >= ABSOLUTE_ZERO_FAHRENHEIT ||
 * myScale == 'K' && myDegrees >= ABSOLUTE_ZERO_KELVIN
 *
 * @author kvlinden
 * @version Spring, 2012 (based on the original ANN implementation)
 */
class Temperature {
      public final static double ABSOLUTE_ZERO_CELSIUS = -273.15,
                                  ABSOLUTE_ZERO_FAHRENHEIT = -459.67,
                                ABSOLUTE_ZERO_KELVIN = 0.0;
      private double myDegrees;
      private char myScale;
}
```

We will enforce this invariant in our implementation.

Since our class invariant relies on the values of absolute zero in each of the different scales, we define constants for all three. Because these constants are likely to be generally useful to temperature-related programs, we define them as **`public`** constants so they are accessible both inside and outside the class declaration.

Static vs. Instance Constants

Note also that since the values of these constants will be the same for all temperature objects, we use the static modifier to define them as *class constants* (or *static constants*) rather than *instance constants*. This means that all temperature objects share these class constants rather than carrying around their own copies of them, as would be the case if they were instance constants. This can be visualized as shown in this diagram:


temperature1
myDegrees
myScale
temperature2
myDegrees
myScale
shared
shared
public final static double
ABSOLUTE_ZERO_CELSIUS = -273.15,
ABSOLUTE_ZERO_FAHRENHEIT = -459.67,
ABSOLUTE_ZERO_KELVIN = 0.0;

Here, the three constant values are represented once and shared by all temperature objects. Given that these absolute zero values are constants, it makes little sense for each temperature object to represent its own duplicate copies. For this reason, *most constants should be declared as static constants*.

Class constants are accessed through the class rather than through an instance of that class (i.e., an object); for example,

```
    Temperature.ABSOLUTE_ZERO_CELSIUS
```

retrieves the value –273.15, where the identifier on the left-hand side of the dot is the temperature class rather than any of the individual temperature objects (e.g., `temperature1`). Useful constants are often provided by libraries of static constants, e.g., the `Math` library provides `Math.PI` and `Math.E`

Variables can also be declared as `static` and thus shared by all class objects, but this is far less common than the use of static constants. Most variables store values that are unique to each individual class object, e.g., `myDegrees` and `myScale`, and thus should be declared as instance variables rather than class variables.

9.3.2. Implementing Class Behaviors

Once the class attributes have been implemented, we are ready to begin writing methods that implement the class behaviors. Class methods generally fall into one of the following categories:

- *Constructors* – Methods that initialize attribute variables
- *Accessors* – Methods that retrieve but do not change attribute values
- *Mutators* – Methods that change attribute variable values
- *Converters* – Methods that provide a representation of an object in a different type
- *Other Utilities* – Methods used by other methods to simplify or avoid redundant coding

In this section, we will see examples of each of these categories of methods.

The `toString()` Method

It is good practice to define methods for output early in the implementation process, because being able to view an object's value can help with checking the correctness of the other operations. For this reason we will begin by building an output method for our Temperature class.

In Java, the `print()` and `println()` methods can be used to display any value whose type is `Object` or any class that extends `Object`. These methods work by asking an object to convert itself to a String using its `toString()` method and then displaying the string returned by that object. Various other operations (e.g., the string concatenation operator +) also use `toString()` messages to convert objects to strings so they can be processed. For these reasons, most classes should provide a `toString()` method. The existing `print()` and `println()` methods can then be used to display objects of that class without further work.

As its name suggests, the `toString()` method is a converter method. From an external perspective, an expression such as

```
temperature1.toString()
```

should return a string containing the degrees and scale attributes of the temperature object to which `temperature1` refers.

From an internal perspective, when I (a temperature object) receive a call to my `toString()` method, I should respond by returning a String containing the values stored in `myDegrees` and `myScale`, with a space separating them. Thus, we can implement the behavior of `toString()` as follows:

```
public String toString() {
      return myDegrees + " " + myScale;
}
```

Note that although a method from a different class is not permitted to access the private attribute variables of a class, a method defined within the same class can access them freely.

Because the `print()` and `println()` methods send the `toString()` message to any object they are asked to display. A statement like

```
System.out.print(temperature1);
```

therefore, will (behind the scenes) send temp1 the `toString()` message to obtain a `String` representation of `temperature1` and display the result. For the `toString()` method just shown, the output produced by the above statement will be value of `temperature1.myDegrees` followed by a space and the value of `temperature1.myScale`. Similarly,

```
System.out.print(temperature2);
```

will display the value of `temperature2.myDegrees`, a space, and the value of `temperature2.myScale`.

Before this method is useful, however, the attribute variables `myDegrees` and `myScale` must have values. We therefore turn our attention to methods that enable attribute variables to be initialized.

Default Constructors

Java allows a class to define methods whose purpose is to initialize the attribute variables of its objects. These special methods are called *constructors*. A constructor that initializes the attribute variables to default values is called a default-value constructor, while a constructor that initializes the attribute variables to user-supplied values is called an explicit-value constructor.

As a temperature object, I must be able to initialize my attribute variables when I am created. The following example shows the definition of a default-value constructor for the `Temperature` class.

```
public Temperature() {
      myDegrees = 0;
      myScale = 'C';
}
```

Though this constructor looks very much like a normal instance method, we see some unusual features:

- There is no return type between public and the method's name. This is because constructor methods have no return type, not even `void`. As an initialization method, a constructor never returns anything to its caller. Its sole purpose is to initialize an object's attribute variables.
- The constructor has the same name as the class's name. This allows Java to determine that this method defines a constructor.
- Though not so unusual, the constructors must always be marked as `public` because programmers wanting to construct new `Temperature` objects must be able to access the constructor.

Note that because we can control the values that are assigned to `myDegrees` and `myScale`, we can ensure that they match the class invariant specified above.

Given this definition, a programmer can now write the following console application to test this much of the class:

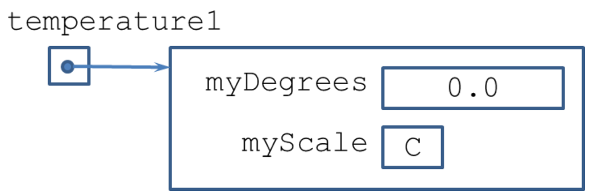
Code:

```
public static void main(String[] args) {
        Temperature temperature1 = new Temperature();
        System.out.println(temperature1);
}
```

Output:

```
0.0 C
```

When the first statement of this program is executed, the temperature class constructor will construct a new `Temperature` object called `temperature1`, which can be viewed as follows:



This object is produced because whenever Java encounters the expression

```
new Temperature()
```

it finds the `Temperature` class and searches for a `Temperature()` constructor that it can use to initialize the newly-created object. Upon finding one, it invokes that constructor method to initialize the object.[1] A constructor is called by the compiler whenever a class object is defined using the `new` operator. For this reason, always provide one or more constructors when building a class to initialize its attribute variables.[2]

Explicit-Value Constructors

The constructor method we just defined only allows us to initialize a `Temperature` object that has default values for its degrees and scale. It would also be useful to allow initializations of other non-default `Temperature` objects. This can be accomplished by defining a second `Temperature` constructor that receives the initial values via its parameters. The algorithm for this method is as follows:

Algorithm:

```
1. Receive degrees, a double, and scale, a char.
2. If degrees and scale specify a temperature that satisfies the class invariant, then:
    a. Set myDegrees = degrees.
    b. Set myScale = scale.
   otherwise:
    a. Print an error message and terminate the program.
```

Note that this algorithm has to be careful to check the values of the parameters supplied by the calling program to ensure that they specify a legal temperature. The scale must specify a legal system, i.e., ‘f’, ‘F’, ‘c’, ‘C’, ‘k’, or ‘K’ and the degrees must specify a temperature above absolute zero in the given scale.

[1] If Java does not find such a method and the class has no other constructors, the compiler will generate a default constructor and use it to initialize the attribute variables with the Java-specified default values.
[2] Static libraries like the `Math` library discussed in the previous section are not instantiated using `new`.

The following example shows the definition of an explicit-value constructor for the `Temperature` class.

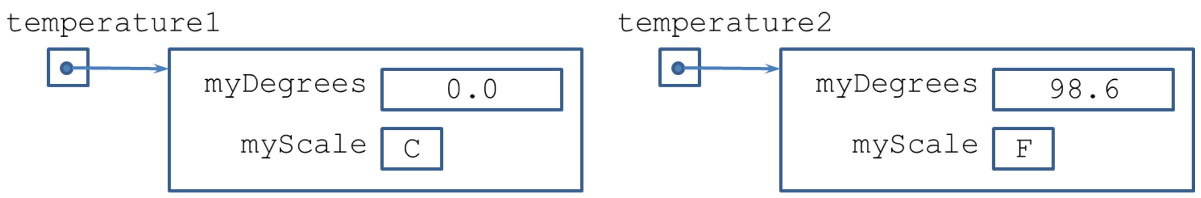
```
public Temperature(double degrees, char scale) {
      if (isValidTemperature(degrees, scale)) {
            myDegrees = degrees;
            myScale = Character.toUpperCase(scale);
      } else {
            System.err.println("Invalid Temperature: (" + degrees + ","
                                                        + scale + ")");
            System.exit(-1);
      }
}
```

This code uses a new method `isValidTemperature()` to validate the supplied degrees and scale before initializing the local instance variables. This protects temperature objects from getting bad data. This new method is discussed in detail in a later section. It also automatically converts lower cases values for 'c', 'f' and 'k' into uppercase values; this helps to ensure conformity to the class invariant, which only allows upper case values. If the degrees and scale do not specify a legal temperature, then this method prints an error message to `System.err`, the output stream designated for printing error messages and terminates the program using `System.exit()`.

Given this definition, a programmer can write a console application to test this much of the class:

Code:
```
public static void main(String[] args) {
      Temperature temperature1 = new Temperature();
      System.out.println(temperature1);
      Temperature temperature2 = new Temperature(98.6, 'f');
      System.out.println(temperature2);
}
```
Output:
```
0.0 C
98.6 F
```

This program constructs two new temperature objects. The first, `temperature1`, is constructed by the temperature class's default constructor and the second, `temperature2`, is constructed by the temperature class's explicit value constructor. The result can be viewed as follows:



Overloading Method Names

Having two different methods with the same name is known as *overloading*. Given the expression

```
new Temperature()
```

Java searches the `Temperature` class for a constructor method named `Temperature` that has no parameters. On finding one, it uses that constructor to perform the initialization.

By contrast, when Java encounters the expression

```
new Temperature(98.6, 'f')
```

it searches the class for a method named `Temperature` whose *signature* (i.e., name and ordered list of parameter types) matches the ordered list of argument types. On finding it, Java uses that constructor to perform the initialization.

More generally, a declaration statement of the form

```
ClassName variableName = new ClassName(argumentList);
```

causes the compiler to search the definition of class *`ClassName`* for a constructor whose parameter types match the types of the arguments in *`argumentList`*. If it finds such a constructor, it uses it to perform the initialization. If it finds no constructors, it automatically generates a default constructor.

Utility Methods

Classes often implement utility methods, which provide useful functions deployed throughout the class definition. The `isValidTemperature()` method introduced above is such a method. It must determine if a given degrees/scale pair specifies a temperature that satisfies the class invariant, that is that the scale be Celsius, Fahrenheit or Kelvin and that the degrees value be at or above absolute zero.

Algorithm:

```
1. Receive degrees, a double, and scale, a char.
2. If scale specifies Celsius, then:
       a. Return the value of (myDegrees >= absolute zero) in Celsius.
   otherwise if scale specifies Fahrenheit, then:
       a. Return the value of (myDegrees >= absolute zero) in Fahrenheit.
   otherwise if scale specifies Kelvin, then:
       a. Return the value of (myDegrees >= absolute zero) in Kelvin.
   otherwise
       a. Return false.
```

To ensure that the degrees and scale values received from the calling program satisfy the class invariant, this algorithm must verify that the scale is legal (Celsius, Fahrenheit or Kelvin) and carefully distinguish the degrees values of absolute zero for Celsius (-273.15°), Fahrenheit (-459.67°) and Kelvin (0.0°). This algorithm can be implemented as follows:

```
public static boolean isValidTemperature(double degrees, char scale) {
      if (Character.toUpperCase(scale) == 'C') {
            return degrees >= ABSOLUTE_ZERO_CELSIUS;
      } else if (Character.toUpperCase(scale) == 'F') {
            return degrees >= ABSOLUTE_ZERO_FAHRENHEIT;
      } else if (Character.toUpperCase(scale) == 'K') {
            return degrees >= ABSOLUTE_ZERO_KELVIN;
      } else {
            return false;
      }
}
```

This implementation automatically converts the scale value received from the calling program into upper case using the static `Character.toUpperCase()` method provided by the `Character` class.[3] The nature of static methods is discussed in the next section.

Static vs. Instance Methods

As discussed in the previous section on static and instance constants, methods can also be declared as either static or instance methods. As with static constants, static method are shared by all class objects and behave in exactly the same way for each object. A consequence of this rule is that a static method may not access instance constants, variables or methods; it may only access static constants, variables and methods.

As an example, consider the `toString()` method defined early in this section. This method returns a potentially different string for each temperature object because its return value is based on the `myDegrees` and `myScale` instance variables, which are potentially different for each temperature object. Thus, `toString()` must be defined as an instance method.

As a second example, consider the `isValidTemperature()` method just implemented. This method does not access any instance variables or methods. Instead, it receives the values it is to test via its parameters and accesses the absolute-zero class constants we defined earlier. Since it uses only class values and parameters, we define it as a class method (i.e., using `static`) rather than as an instance method. This allows its definition to be shared by all temperature objects.

Accessor Methods

An *accessor* is a method that allows a programmer to retrieve, but not modify, some attribute of a class. From an external perspective, a programmer should be able to call the class method

```
temperature1.getDegrees()
```

to retrieve the `myDegrees` value of the temperature object `temperature1` and call the class method

```
temperature1.getScale()
```

to retrieve the `myScale` value of `temperature1`. As the names suggest, it is common practice for the name of an accessor method to begin with the prefix `get` followed by the attribute it is accessing.

An autonomous temperature object will know its degrees value and its scale. From an internal perspective, when I (a temperature object) receive a call to my `getDegrees()` method, I should return the value of my `myDegrees` instance variable. We thus have this simple algorithm for `getDegrees()`.

```
Algorithm:
    1. Return myDegrees.
```

The code for the accessor methods for the `Temperature` class attributes is as follows.

[3] Given that scale is an integer-compatible type, the `switch` would likely be a more efficient choice. See the discussion of this specialized selection statement in Chapter 5

```
public double getDegrees() {
      return myDegrees;
}
public char getScale() {
      return myScale;
}
```

We mark these accessor methods as `public` to allow external programmers to access them.

To illustrate how this works from an external perspective, consider the following code segment:

Code:

```
Temperature temperature1 = new Temperature();
System.out.println(temperature1.getDegrees());
Temperature temperature2 = new Temperature(98.6, 'f');
System.out.println(temperature2.getDegrees());
```

Output:

```
0.0
98.6
```

Note how each temperature object "knows" or keeps track of its own degrees value.

Mutator Methods: Change Degrees

Methods that change the values of an object's instance variables change the internal state of the object and are thus called *mutators*. One example of a mutator method for the temperature class is `change()`, a method that changes the value of the temperature by a given amount of degrees. From an external perspective, the calling program must be able to tell a temperature object to change its degrees value. From an internal perspective, my `change()` method must contain the instructions that I (as an autonomous temperature object) must follow to change my degrees value (safely!). The algorithm for this mutator is:

Algorithm:

```
1. Receive an amount of degrees to add to myDegrees.
2. If incrementing myDegrees by amount is still a valid temperature then:
   a. Increment myDegrees by amount.
   otherwise:
   a. Print an error message and terminate the program.
```

In step 2, I must be certain that the new degrees value satisfies the class invariant before changing `myDegrees`. The scale will remain unchanged and therefore will still be ok, and we can always add to `myDegrees` without the risk of going below absolute zero, but the calling program could pass a negative value for `amount`, which might break the class invariant. The following code implements this algorithm:

```
public void change(double amount) {
      double newDegrees = myDegrees + amount;
      if (isValidTemperature(newDegrees, myScale)) {
            myDegrees = newDegrees;
      } else {
            System.err.println("Invalid temperature change: " + amount);
            System.exit(-1);
      }
}
```

As discussed above, this code enforces the class invariant using the `isValidTemperature()` method. We mark this mutator method as `public` to allow external programmers to access it. Given this method, a programmer can now write

Code:

```
      Temperature temperature1 = new Temperature(32.0, 'F');
      temperature1.change(20.0);
      System.out.println(temperature1);
```

Output:

```
      52.0 F
```

Here, `temperature1` is changed from 32.0 F to 52.0 F.

Mutator Methods: Set Scale

A second example of a mutator method is `setScale()`, a method that changes the scale of the temperature object. From an external perspective, the calling program must be able to tell a temperature object to change its scale and expect the degrees value to change appropriately. That is, changing a 0.0C temperature to Fahrenheit should automatically change the degrees to 32.0. From an internal perspective, my `setScale()` method must contain the instructions that I (as an autonomous temperature object) must follow to change my scale and to convert my degrees value. The algorithm for this mutator is:

Algorithm:

```
1. Receive a scale value, scale.
2. If scale is Celsius then:
   a. Convert myDegrees to the appropriate equivalent in Celsius.
   b. Set myScale to Celsius.
   Else if scale is Fahrenheit then:
   a. Convert myDegrees to the appropriate equivalent in Fahrenheit.
   b. Set myScale to Fahrenheit.
   Else if scale is Kelvin then:
   a. Convert myDegrees to the appropriate equivalent in Kelvin.
   b. Set myScale to Kelvin.
   Otherwise:
   a. Print an error message and terminate the program.
```

This algorithm ensures that new scale is a legal scale, and makes the appropriate conversion. Note that if the current temperature satisfies the class invariant and the new scale is valid, then the new temperature will automatically satisfy the class invariant.The following code implements this algorithm for Celsius conversions:

```
	public void setScale(char scale) {
		if (Character.toUpperCase(scale) == 'C') {
			convertToCelsius();
		} else if (Character.toUpperCase(scale) == 'F') {
			convertToFahrenheit();
		} else if (Character.toUpperCase(scale) == 'K') {
			convertToKelvin();
		} else {
			System.err.println("Invalid Temperature scale: (" + scale + ")");
			System.exit(-1);
		}
	}

	private void convertToCelsius() {
		if (myScale == 'F') {
			myDegrees = 5.0 / 9.0 * (myDegrees - 32.0);
		} else if (myScale == 'K') {
			myDegrees = myDegrees - 273.15;
		}
		myScale = 'C';
	}
```

This implementation makes use of the utility method `convertToCelsius()` to do the actual conversion of the degrees and the setting of the scale. The `setScale()` method is public and the `convertToCelsius()` method is private because external programs wishing to change the scale of a temperature should call the `setScale()` method.

Given these methods, a programmer can now write

Code:

```
	Temperature temperature2 = new Temperature(32.0, 'F');
	Temperature2.setScale('C');
	System.out.println(temperature2);
```

Output:

```
	0.0 C
```

Here, `temperature2` is set to 32.0 F and then converted to the equivalent Celsius value, 0.0°.

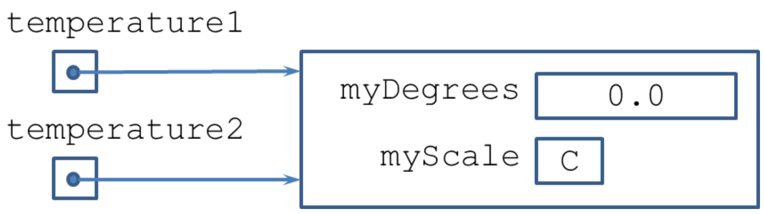
The private utility methods `convertToFahrenheit()` and `convertToKelvin()` are similar and are left as exercises.

Cloning Temperatures

As we saw with arrays in Chapter 8, making copies of reference types like arrays is not a trivial matter of using an assignment statement. Classes define new reference types and exhibit the same behavior. The following code segment does not make a copy of the `temperature1` as one might expect:

```
		Temperature temperature1 = new Temperature();
		Temperature temperature2 = temperature1;
```

The result of this code segment can be visualized as follows:



As with arrays, the assignment statement simply copies the handle of the reference type, that is, its reference, rather than actually copying the actual value. Using this approach to copying leads to the perhaps surprising consequence that changes made to `temperature1` will also manifest themselves in `temperature2`, as shown in the following example code segment:

Code:

```
Temperature temperature1 = new Temperature();
Temperature temperature2 = temperature1;
temperature1.change(10.0);
System.out.println("temperature1: " + temperature1);
System.out.println("temperature2: " + temperature2);
```

Output:

```
temperature1: 10.0 C
temperature2: 10.0 C
```

Here `temperature1` and `temperature2` actually refer to the same `Temperature` object as shown above and, thus, adding 10 degrees to `temperature1` changes the value of the object referred to by `temperature2`.

To remedy this problem, a class can provide a `copy()` method. From the external perspective, a programmer would like to call the `copy()` method on one temperature object and have the method return a new, separate but equal copy of the original temperature object.[4] The code for this method is as follows:

```
public Temperature copy() {
      return new Temperature(myDegrees, myScale);
}
```

This method simply uses the explicit value constructor to construct a new, equivalent temperature object. Using this new method, the unfortunate behavior shown in the example above can be fixed as shown here:

Code:

```
Temperature temperature1 = new Temperature();
Temperature temperature2 = temperature1.copy();
temperature1.change(10.0);
System.out.println("temperature1: " + temperature1);
System.out.println("temperature2: " + temperature2);
```

Output:

```
temperature1: 10.0 C
temperature2: 0.0 C
```

Here, the use of our new `copy()` method creates a new object for `temperature2` which is not affected by changes to `temperature1`.

[4] The Java `Object` class provides a `clone()` method that performs this copying function, but this method is protected and can only be exposed by specifying that the class implement `Clonable`.

9.3.3. Example Revisited

As we saw in Section 9.2.3, Processing, as a specialized version of Java, also supports classes, and their use can be just as effective in graphical applications. The chapter example shown in Figure 9-1 calls for a pinball game with, among other things, stationary ball objects, a bouncing pinball and a movable paddle. This section will use the techniques developed in this section to implement classes in Processing for these elements of the example.

The `SimpleBall` Class

The instance variables, invariant and basic behaviors for the simple ball class are described in Section 9.2.3. With these in hand, we can design the required methods using the following algorithm:

Given:

- `myX` and `myY` are floats that represent my x and y coordinates (we use floats because Processing supports floating point coordinates and to support some useful floating point methods that we will need);
- `myDiameter` is an integer that has a positive value;
- `myColor` is a legal color value.

Algorithm (explicit-value constructor for a stationary ball**):**

1. Receive `x` and `y` components and a `diameter` from my calling program;
2. Set `myX` = `x`;
3. Set `myY` = `y`;
4. Set `myDiameter` = `diameter` (or 1 if diameter ≤ 0);
5. Set `myColor` to a legal, random color;

Algorithm (`render()`**):**

1. Set my stroke and fill colors;
2. Draw a circle with my location with my size.

Here, the constructor method creates a colored, stationary ball with the given coordinates and diameter; for the ball to be visible, the display window must show the ball with the given state. In step 5, the constructor ensures that the diameter is positive. The render method draws the ellipse in the right place.

The following code implements these elements of the simple ball class, plus a useful set of accessors and mutators:

```
/**
 * This class implements a non-moving ball obstacle for the PinBall
 * application.
 *
 * @author kvlinden
 * @version Summer, 2009
 */
class SimpleBall {

  private float myX, myY;
  private int myDiameter; // This value must be non-negative
  private int myColor;
  // default constructor
  public SimpleBall() {
    myX = 50.0;
    myY = 50.0;
    myDiameter = 25;
    myColor = color(255);
  }

  // explicit-value constructor
  public SimpleBall(float x, float y, int diameter) {
    myX = x;
    myY = y;
    myDiameter = constrain(diameter, 1, MAX_INT);
    myColor = color(random(255), random(255), random(255));
  }

  // Accessor methods
  public float getX() {
    return myX;
  }
  public float getY() {
    return myY;
  }
  public int getDiameter() {
    return myDiameter;
  }
  public int getColor() {
    return myColor;
  }

  // mutator method
  public void setDiameter(int diameter) {
    myDiameter = constrain(diameter, 1, MAX_INT);
  }

  // display utility method
  public void render() {
    stroke(0);
    fill(myColor);
    ellipse(myX, myY, myDiameter, myDiameter);
  }
}
```

Note the API utility method `render()`. From an external perspective, the calling program must be able to display the ball whenever necessary (e.g., each time through the `draw()` method). From an internal perspective, my `render()` method must contain the instructions that I (as an autonomous `SimpleBall` object) must follow to display myself on the output pane.

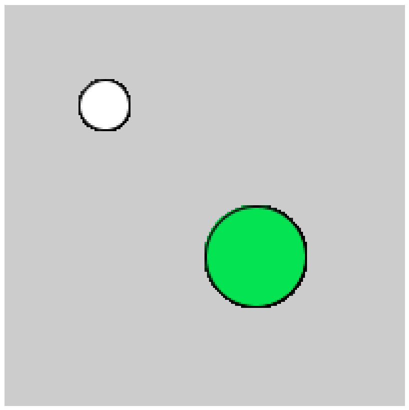
In the algorithm for this method, specified in Section 9.2.2, I can assume that my state variables are legal because I've been careful to enforce my class invariants in every case where I initialize or change the variables. The following code implements this algorithm:

```
  public void render() {
    stroke(0);
    fill(myColor);
    ellipse(myX, myY, myDiameter, myDiameter);
  }
```

As a ball object, I can use this code to draw myself according the values of my attributes. We mark these accessor methods as `public` to allow external programmers to access them.

Given this class definition, we can run the following program to produce the given output. `ball1` renders itself on the upper left and `ball2` renders itself on the lower right.

```
SimpleBall ball1, ball2;

void setup() {
  size(200, 200);
  smooth();
  ball1 = new SimpleBall();
  ball2 = new SimpleBall(125, 125, 50);
  noLoop();
}

void draw() {
  ball1.render();
  ball2.render();
}
```



The `Paddle` Class

The previous sections have detailed how to build a stationary ball class. This section shows how to upgrade the ball class to support bouncing and to add a paddle class, as shown in Figure 9-3. In this simplified version of the game, the ball drops from the top of the display window and bounces off the walls and the floor. The user can move the paddle from left to right using the mouse, but in this iteration the ball won't bounce exclusively off the paddle. We'll add this more desirable behavior in the next iteration.

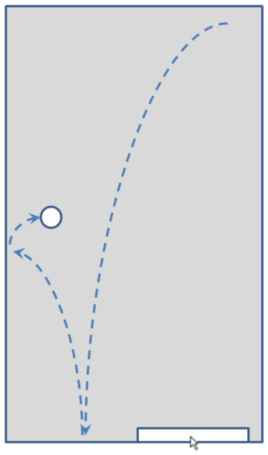


Figure 9-3. A simplified bounce-ball game sketch

We start by implementing the paddle class. The game player sees the paddle as a flat rectangle at the bottom of the display window, but while Processing provides a rectangle drawing method (`rect()`), we need to implement a paddle class to encapsulate the full range of the paddle's behavior. From the external perspective, a programmer would expect to be able to do the following:

- Constructor – Create a new paddle object;
- Render – Draw the object as a rectangle based on its current state.
- Move – Tell the paddle where to position itself, based, for example, on the current location of the mouse or a default starting location as defined by the game rules, and the paddle object should remember this as part of its state;

From the internal perspective, I, as a Paddle object, need to include all those attributes and methods required to implement the state and behaviors required by my API. The attributes must include the following:

- Coordinates – I must represent my x coordinate (a `float`). Note that because I always appear at the bottom of the screen, I don't need to store a y-coordinate;
- Dimensions – I must represent the width and thickness of my rectangle (`int`s).

With these attributes in hand, I must implement methods for each of the behaviors specified above. I can do this using the following algorithms:

Given:

- `myX` is a float that represents the horizontal center of the paddle and has a value such that 0 ≤ `myX` ≤ screen width;
- `myWidth` and `myHeight` represent the width and height of my rectangle respectively.
- `myScreenWidth` and `myScreenHeight` represent the width and height of the output display respectively.[5]

Algorithm (constructor**):**

1. Receive an `x` coordinate, desired paddle `paddleWidth` and the `screenWidth` and `screenHeight.`
2. Set `myX` = x (must be between 0 and screenWidth);
3. Set `myWidth` = `paddleWidth` (must be between 0 and `screenWidth`);
4. Set `myHeight` = 5;
5. Set `myScreenWidth` and `myScreenHeight` equal to `screenWidth` and `screenHeight` respectively.

Algorithm (`render()`**):**

1. Set my stroke and fill colors;
2. Draw a rectangle at the bottom of the display window with my location with my size.

Algorithm (`move()`**):**

1. Receive from my calling program `x`, an x coordinate;
2. Set `myX` = `x` (must be between 0 and `screenWidth`).

Here, the constructor method creates a new paddle object and sets the attributes to the given values. We include the width of the display window so that the paddle object knows how far it can slide to the right and we keep the height so that the paddle knows how to draw itself at the bottom of the display window. The render method draws the right kind of rectangle in the right place. The move method, in step 2, must ensure that `myX`'s value is between 0 and the width of the display window.

[5] Processing treats new classes as inner classes, which would allow the `Paddle` class to access the `width` and `height` variables globally from the main sketch. This is not a standard approach in Java programming and can lead to problems of scalability. Thus, this text adopts the practice of passing parameters rather than making global references.

The implementation of this class is as follows.

```
class Paddle {

  private final int HEIGHT = 5;

  private float myX;
  private int myWidth, myHeight, myScreenWidth, myScreenHeight;

  // explicit-value constructor
  public Paddle(float x, int paddleWidth,
                int screenWidth, int screenHeight) {
    myX = constrain(x, 0, screenWidth);
    myWidth = constrain(paddleWidth, 0, screenWidth);
    myHeight = HEIGHT;
    myScreenWidth = screenWidth;
    myScreenHeight = screenHeight;
  }

  // accessor methods
  public float getX() {
    return myX;
  }
  public int getWidth() {
    return myWidth;
  }
  // mutator method
  public void move(float x) {
    myX = constrain(x, 0, myScreenWidth);
  }
  // utility method
  public void render() {
    stroke(0);
    fill(255);
    rect(myX – (myWidth / 2), myScreenHeight - myHeight,
         myWidth, myHeight);
  }
}
```

For the most part, this implementation uses the basic techniques discussed in this section. As discussed above, the paddle class stores the width and height of the display window for use in maintaining the class invariants. The code implements only those accessor methods that are needed in this prototype.

Adding Movement to the `SimpleBall` Class

We now move to the bouncing ball class, which we will implement as an extended version of the stationary ball class discussed above. The ball should support the bouncing behavior sketched in Figure 9-3. From an external perspective, we must add a movement behavior to the API:

- Constructor – Create a new ball object;
- Render – Draw the object as a ellipse based on its current state.

- Move – Ask the ball to move itself according to its internal state from a legal position in one animation frame to an appropriate position in the next frame. The ball should bounce off the walls and the floor.

From an internal perspective, I, as a bouncing ball object, must add attributes for velocity and acceleration to handle the new API:

- Coordinates – my x and y coordinates (`float`);
- Velocity – my x and y velocities (`float`);
- Acceleration – my x and y accelerations (`float`);
- Dimensions – the diameter of my circle (`int`);
- Color – my color (`int`).

With these attributes in hand, I must make the following modifications to my stationary ball class algorithms:

Given:

- `myX` and `myY` are floats that represent my x and y coordinates;
- `myDiameter` is an integer that has a positive value;
- `myVelocityX` and `myVelocityY` are floats that represent my x and y coordinates;
- `myAccelerationX` and `myAccelerationY` are floats that represent my x and y acceleration;
- `myColor` is a legal color value.

Algorithm (constructor**):**

```
1. Receive x and y coordinates, diameter, xVelocity and yVelocity and
   xAcceleration and yAcceleration from my calling program;
2. Set myX and myY to x and y;
3. Set myDiameter = diameter (or 1 if diameter ≤ 0);
4. Set myVelocityX and myVelocityY to xVelocity and yVelocity;
5. Set myAccelerationX and myAccelerationY to xAcceleration and
   yAccleration;
6. Set myColor to a legal, random color;
```

Algorithm (`render()`**):**

```
1. Set my stroke and fill colors;
2. Draw a circle with my location with my size.
```

Algorithm (`move()`**):**

```
1. Increment my x and y positions by my x and y velocity values respectively;
2. If I hit the side:
       a. Invert my x velocity and acceleration;
3. If I hit the bottom:
       a. Invert my y velocity;
4. Increment my x and y velocities by my x and y acceleration values respectively.
```

Note that you can see that in the `move()` algorithm, step 2.a inverts both velocity and acceleration whereas step 3.a only inverts the velocity. This is because the horizontal acceleration is (mostly) reflected by a bounce off the wall whereas the downward-directed gravitational constant is, well, constant.

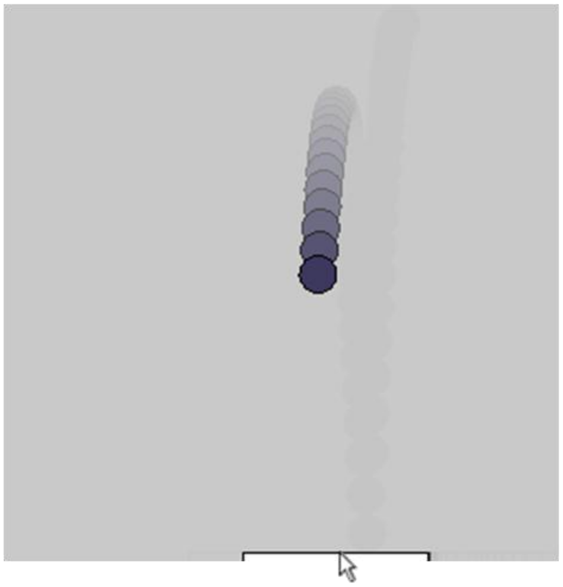
We can implement this upgraded algorithm as follows:

```
class Ball {

  private float myX, myY, myDiameter,
                myVelocityX, myVelocityY,
                myAccelerationX, myAccelerationY;
  private int myColor;


  public Ball(float x, float y, float diameter,
              float velocityX, float velocityY,
              float accelerationX, float accelerationY) {
    myX = x;
    myY = y;
    myDiameter = constrain(diameter, 1, MAX_INT);
    myVelocityX = velocityX;
    myVelocityY = velocityY;
    myAccelerationX = accelerationX;
    myAccelerationY = accelerationY;
    myColor = color(random(255), random(255), random(255));
  }

  public void render() {
    stroke(0);
    fill(myColor);
    ellipse(myX, myY, myDiameter, myDiameter);
  }

  public void move() {
    myX += myVelocityX;
    myY += myVelocityY;
    if (hitsSide()) {
      myVelocityX *= -1;
      myAccelerationX *= -1;
    }
    if (hitsBottom()) {
      myY = height - myDiameter/2 - paddle.getHeight();
      myVelocityY *= -1;
    }
    if (Math.abs(myVelocityX) < 0.01) {
      myVelocityX = 0.0;
    }
    myVelocityX += myAccelerationX;
    myVelocityY += myAccelerationY;
  }
  // determines if this ball has hit the side of the display
  private boolean hitsSide() {
    return (myX - myDiameter/2 <= 0) || (myX + myDiameter/2 >= width);
  }
  // determines if this the ball hits the bottom of the display
  private boolean hitsBottom() {
    return Math.abs(myY + myDiameter/2 + paddle.getHeight() - height)
           <= myVelocityY;
  }
}
```

The `move()` method implementation has some additional, private utility methods that it uses to determine if the ball hits the sides of the display, `hitsSide()`, or the bottom of the display, `hitsBottom()`. These methods are marked as `private` because they are not part of the external API for the `Ball` class and should thus only be used internally. The methods are relatively simple utility methods that return Boolean values used in the `move()` method.

With the Paddle and Ball classes in place, we can now using the following code to implement our bouncing ball game.

```
final int BALL_RADIUS = 20, PADDLE_WIDTH = BALL_RADIUS * 5;
Ball ball;
Paddle paddle;
void setup() {
  size(300, 300);
  background(255);
  ball = new Ball(width - 
BALL_RADIUS*2, BALL_RADIUS, 
BALL_RADIUS,
                    random(-2.0, 2.0), -
0.0, 0.0017, 1.0);
  paddle = new Paddle(width / 2, 
PADDLE_WIDTH, width, height);
}
void draw() {
  // Whitewash the previous frames.
  noStroke();
  fill(200, 50);
  rect(0, 0, width, height);
  ball.render();
  ball.move();
  paddle.render();
}
void mouseMoved() {
  paddle.move(mouseX);
}
```



The output of this game program is shown on the right. The program constructs a ball and a paddle and repeatedly moves the ball and renders the ball and the paddle. The paddle movement is implemented by the `mouseMoved()` method and the background is slowly whitewashed away in the animation using the technique described in Chapter 4.

9.4. Object Interaction

Objects in the real world interact with one another. Therefore, we would like to model these interactions in our programs. In the case of temperature objects, we would like to be able to compare one temperature object with another. For example, we might want to know if one temperature object represents a temperature that is lower than the temperature represented by another temperature object.

9.4.1. Comparing Temperatures

Because Java does not allow us to overload operators, we cannot simply use the expression `temperature1 < temperature2`. Instead, we can implement our own comparison methods. From an external perspective, a programmer could compare two temperatures as follows:

Code:
```
	Temperature oldTemperature = new Temperature(10.0, 'C');
	Temperature newTemperature = new Temperature(20.0, 'C');
	if (oldTemperature.lessThan(newTemperature)) {
		System.out.println("It's getting warmer!");
	}
```

Output:
```
	It's getting warmer!
```

Note how `lessThan()` is called on `oldTemperature`, passing `newTemperature` as an argument. This is the same structure we use to compare string values, as in `oldString.equals(newString)`. This may be somewhat less intuitive than the operator overloading approach, but it is a very object-oriented view.

Note that if `temperature1` refers to 0° C and `temperature2` refers to the 32° F, then the expression

```
   temperature1.lessThan(temperature2)
```

should return `false`, since these two temperatures are in fact equal. Implementing `lessThan()` is thus not just a simple matter of comparing the degrees values of the two temperatures; we must consider the scale as well. We can implement `lessThan()` as follows:

```
	public boolean lessThan(Temperature otherTemperature) {
		Temperature temp = otherTemperature.copy();
		temp.setScale(myScale);
		return myDegrees < temp.getDegrees();
	}
```

This implementation resolves the problem of different scales by converting the other temperature received as a parameter value to the appropriate scale before comparing degrees. Once we have two temperatures in the same scale, we can simply compare their `myDegrees` values using the normal less-than operator. From the internal perspective, I, as a temperature object, am being asked to compare myself with another temperature. To do this, I convert that other temperature to my scale so that I can directly compare my degrees with its degrees.

Note that `lessThan()` starts by making a copy of the temperature parameter value before working with it. This is important because any change made to `otherTemperature`, a reference type, will manifest itself in the calling program as well. Thus, it's safer to work with a temporary copy of the other temperature so as not to risk modifying the arguments passed to this method.

As we consider implementing the other comparison methods, it becomes clear that we'd be repeating much of the same code. For example, the `lessThanOrEqualTo()` method would be identical to the `lessThan()` method except that the return statement would use <= rather than <. Much the same would be true for the other comparison methods. When we face the prospect of duplicating code in this manner over and over again, it is wise to consider whether there is a common function shared by all the comparison operators that can be factored out and implemented separately. In this case, we can adopt the approach used in Java's `String`, `BigInteger`, and various other classes, and implement a general `compareTo()` method that "factors out" the code that these comparison methods have in common. More precisely, in keeping with these other classes, we can specify that our `compareTo()` method should behave as follows

```
Algorithm:
    1. Receive otherTemperature, another temperature object.
    2. Declare temp as a copy of otherTemperature.
    3. Set the scale of temp to be myScale.
    4. If myDegrees < temp's degrees then
       a. Return -1.
       Else if myDegrees > temp's degrees then
       b. Return +1.
       Else
       a. Return 0.
```

This algorithm uses elements of functionality from `lessThan()` that would likely have to be reimplemented in `lessThanOrEqualTo()`, but returns 0, –1, or +1 depending on whether `myDegrees` is equal to, less than, or greater than `temp`'s degrees. This algorithm can be implemented as follows:

```java
private int compareTo(Temperature otherTemperature) {
      Temperature temp = otherTemperature.copy();
      temp.setScale(myScale);
      if (myDegrees < temp.getDegrees()) {
            return -1;
      } else if (myDegrees > temp.getDegrees()) {
            return 1;
      } else {
            return 0;
      }
}
```

Given this method, the relational operators can now be implemented with almost no duplicate code. For example, the `equals()` and `lessThan()` methods can be defined using `compareTo()` as follows:

```
public boolean lessThan(Temperature otherTemperature) {
      return compareTo(otherTemperature) < 0;
}

public boolean equals(Temperature otherTemperature) {
      return compareTo(otherTemperature) == 0;
}
```

Thus, by "factoring out" the code common to all the comparison methods, we were able to write methods that are simpler and, therefore easier to debug and to maintain. The remaining relational operators can be defined in a similar fashion and are left as exercises.

9.4.2. Example Revisited

In our running example, we would like our simple ball objects to bounce, not just off of the walls and the floor, but also off of each other and off of the paddle. We'd have to detect when two balls "hit" each other and then program a "collision" algorithm that reflects their velocities in the appropriate directions.

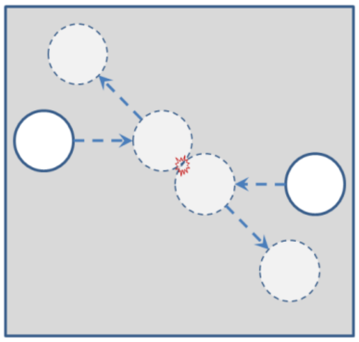


Figure 9-4. A colliding balls sketch

Handling Inter-ball Collisions

We would like to model collisions such as those shown in the sketch in Figure 9-4. In this sketch, two ball objects are moving toward each other, one from the left and one from the right. When they hit each other in the middle of the display window, they should collide and move off in new directions based on the precise point of contact.

To implement this, our `draw()` animation loop must not only ask balls to move and render themselves for each frame, but it will also need to check if the balls hit each other and if so, implement the collision. We could write methods to check for the hit and perform the collision directly in the main sketch, but in the object-oriented paradigm, we think instead of *responsibility*. What object is responsible for knowing about collisions? What object has access to the position and velocity values required to perform these calculations? In our case, this object is the ball itself. The ball object knows its position and its velocity and we've already asked it to move and render itself. Why not ask it to determine if it is hitting another ball and if so, to reflect its velocities in the correct way?

We've encountered this object-centered thinking already in this text. Recall that when we want to determine if two string objects are equal (in the sense that they have the same characters in the same order), we use one of the following expressions:

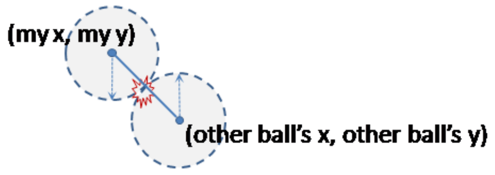
```
myString.equals(myOtherString)
myString.equalsIgnoreCase(myOtherString)
```

These expressions ask `myString` to check and see if it is equal to a given argument (i.e., `myOtherString`). This seems an odd way to think about checking string equivalence, but this is the way we work with reference types in Processing and Java.[6] We need to implement hit and collide methods that operate in a similar, object-oriented way and add them to our `Ball` class API.

To upgrade the Ball class to support collisions, we retain its current set of implemented attributes and methods, and add the following new methods:

Algorithm (`hits()`):

1. Receive a SimpleBall object, `otherBall`, from the calling program;
2. Return true if the distance between my x-y coordinates and those of `otherBall` is less than the sum of my radius and `otherBall`'s radius.



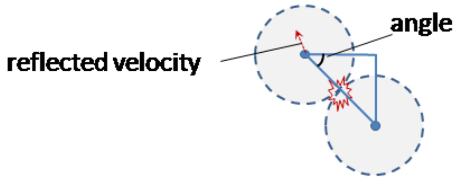
This collision algorithm returns true if the distance between my center point and the other ball's center point is less than the sum of our radii. This is shown in the figure to the right in which I am the ball to the upper left, and `otherBall` is the ball on the lower right.

Note how the algorithm refers to "my" x coordinate (to be implemented as `myX`) and the "the other ball's" x coordinate (to be implemented as `otherBall.getX()`). As a SimpleBall object, I have access to my own x coordinate directly, but not to the other ball's x coordinate. For that coordinate, I must use the other ball's accessor method. This is the manner in which one object can interact with another.

Algorithm (`collide()`):

1. Receive a SimpleBall object, `otherBall`, from the calling program;
2. Set `angle` = atan2(`otherBall`'s y – `myY`, `otherBall`'s x – `myX`);
3. Set `otherBallX` = `myX` + cos(`angle`) * (`myDiameter`/2 + `otherBall`.getDiameter()/2);
4. Set `otherBallY` = `myY` + sin(`angle`) * (`myDiameter`/2 + `otherBall`.getDiameter()/2);
5. Set `ax` = (`otherBallX` - `otherBall`'s x);
6. Set `ay` = (`otherBallY` - `otherBall`'s y);
7. Set `myVelocityX` = (`myVelocityX` - `ax`);
8. Set `myVelocityY` = (`myVelocityY` - `ay`);

This collision algorithm modifies my x and y velocities based on the angle of intersection between my center point and `otherBall`'s center point. This angle is computed using the two-argument arc-tangent function (atan2), which computes the angle shown in the figure to the right. The algorithm then uses



[6] With primitive types in Processing, like `int`, we can simply execute an operation, say addition, using the expression: `1 + 1`. In more fully object-oriented languages, however, there are no primitive types and everything, even integer addition, must be done in an object-oriented way: `myInteger.add(myOtherInteger)`.

this angle to modify the velocities of this ball appropriately.[7] This method interacts with the other ball object in the same manner as the `hits()` method interacts with the other ball.

Handling Ball-Paddle Interaction

We are now prepared to implement the interaction between the bouncing ball and the paddle. In particular, we would like to modify the sketch shown in Figure 9-3 so that the ball only bounces up when it hits the paddle, and falls into oblivion if it otherwise hits the floor. We will also take this opportunity to add an array of falling balls, just to make the user's job challenging. A sketch of this behavior is shown in Figure 9-5.

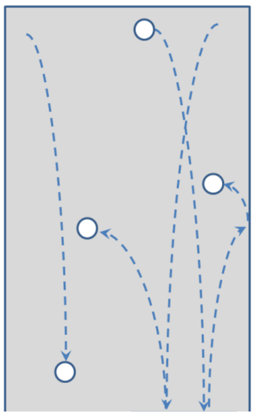


Figure 9-5. Multiple balls that bounce off the paddle

Modifying the ball class to implement bouncing only off the paddle requires the following small change to its move() method.

Algorithm (`move()`):

```
1. Receive paddle, the game's paddle object;
2. Increment my x and y positions by my x and y velocity values
   respectively;
3. If I hit the side:
     a. Invert my x velocity and acceleration;
4. If I hit the bottom and the paddle:
     a. Invert my y velocity;
5. Increment my x and y velocities by my x and y acceleration values respectively.
```

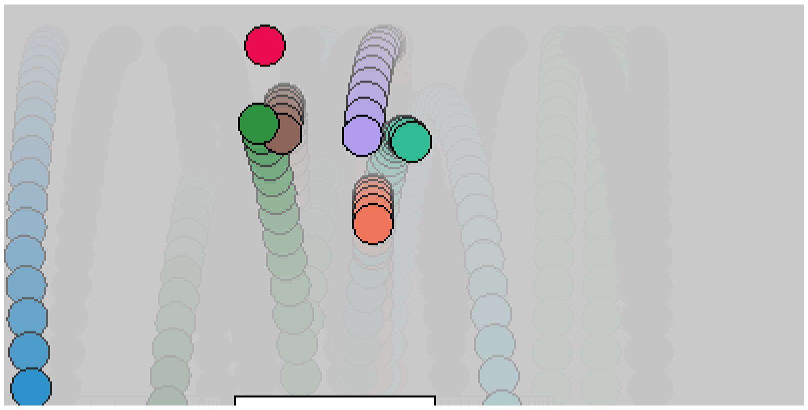
We can implement this modified algorithm as follows.

```
public void move(Paddle paddle) {
  myX += myVelocityX;
  myY += myVelocityY;
  if (hitsSide()) {
    myVelocityX *= -1;
    myAccelerationX *= -1;
  }
  if (hitsBottom() && hitsPaddle(paddle)) {
    myY = height - myDiameter/2 - paddle.getHeight();
    myVelocityY *= -1;
  }
  if (Math.abs(myVelocityX) < 0.01) {
    myVelocityX = 0.0;
  }
  myVelocityX += myAccelerationX;
  myVelocityY += myAccelerationY;
}

private boolean hitsPaddle(Paddle paddle) {
  return Math.abs(myX - paddle.getX()) < (paddle.getWidth() / 2);
}
```

[7] This algorithm is based on the collision algorithm presented in Algorithms for Visual Design, K. Terzidis, Wiley, 2009. For a more detailed discussion of how this algorithm works, see that text.

This modified code uses interacting objects, in this case, the Ball class using the Paddle class, to implement the paddle bouncing behavior that we desire.

To implement multiple balls, we modify the main program sketch, which is responsible for the basic game animation. This responsibility includes the determination of how many balls and how many paddles there will be as well as if and when the various objects are expected to interact with one another. The modified program is as follows, with the output shown first.



```
/**
 * BounceBall implements a simple paddleball game with multiple balls.
 *
 * @author kvlinden
 * @version Fall, 2009
 */

final int BALL_RADIUS = 20,
          PADDLE_WIDTH = BALL_RADIUS * 5,
          BALL_COUNT = 20;

Ball[] balls;
Paddle paddle;
float phaseIn;

void setup() {
  size(400, 200);
  background(255);
  balls = new Ball[BALL_COUNT];
  for (int i = 0; i < balls.length; i++) {
    balls[i] = new Ball(random(width - BALL_RADIUS * 2) + BALL_RADIUS,
                        BALL_RADIUS, BALL_RADIUS, random(-2.0, 2.0),
                        -0.0, 0.0017, 1.0);
  }
  paddle = new Paddle(width / 2, PADDLE_WIDTH, width, height);
  phaseIn = 0.0;
}
```

```
void draw() {
  // Slowly whitewash the previous frames.
  noStroke();
  fill(200, 50);
  rect(0, 0, width, height);
  for (int i = 0; i < balls.length && i < phaseIn; i++ ) {
    balls[i].render();
    balls[i].move(paddle);
  }
  paddle.render();
  phaseIn += 0.1;
}
void mouseMoved() {
  paddle.move(mouseX);
}
```

To implement the paddle bouncing behavior, this program simply uses the new `move()` method from the `Ball` class and passes the paddle as an argument. The ball's `move()` method does the rest, with the help of course from the `Paddle` class.

To implement the multiple balls, we have chosen to phase the additional balls in slowly rather than dumping them all at the very beginning of the game. We implement this using the `phaseIn` variable, which we initialize to 0.0 and then increment by 0.1 on each animation frame and then use as a limit on the `for` loop in the animation. The for condition expression is the unusually complex condition:

```
i < balls.length && i < phaseIn
```

The first part of the condition (`i < balls.length`) limits the loop to the total number of balls we have initialized for this animation; this is the sort of condition we have used before in our counting loops. However, we add a second condition (`i < phaseIn`) which has the effect of stopping the for loop before it has processed all the balls in the ball array. As the value of `phaseIn` grows, the number of balls that are actually rendered and moved grows; by the end of the animation, all the balls have been released.

9.5. Revisiting the Example

As a final prototype for the chapter example, this section modifies the bounceBall prototype from the last section to produce a pinball game in which only one ball moves and a number of other balls are fixed at random locations in space as targets (i.e., bumpers in a normal pinball game). The moving ball bounces off the targets if it collides with them, and on each bounce a bit of extra velocity is added to model the springs on real pinball games. The ball shoots up from the bottom right at the beginning of the game.

We implement the bounce boost factor by making the following small change to the Ball's `collide()` method:

```
public final float BOOST_FACTOR = 1.01;
public void collide(Ball target) {
  float angle = atan2(target.getY() - myY, target.getX() - myX);
  float targetX = myX + cos(angle) *
                      (myDiameter/2 + target.getDiameter()/2);
  float targetY = myY + sin(angle) *
                      (myDiameter/2 + target.getDiameter()/2);
  float ax = (targetX - target.getX());
  float ay = (targetY - target.getY());
  myVelocityX = (myVelocityX - ax) * BOOST_FACTOR;
  myVelocityY = (myVelocityY - ay) * BOOST_FACTOR;
}
```

We model the stationary target bumpers and the moving ball as Ball objects. The game engine simply sets the velocity of the target bumpers to zero and sets the moving ball's velocity appropriately. The paddle operates as it did before. The key change to the game engine algorithm is the addition of the following steps that handle the situations when the moving ball bounces off the stationary target bumpers.

Algorithm (`draw()`):

```
1. Whitewash the previous frames away;
2. Display and move the pinball;
3. Repeat for each of the stationary target bumpers:
    a. Draw the current target bumper (again, so that it doesn't get whitewashed away);
    b. If the moving ball hits the current target bumper:
        i. Ask the pinball to collide itself with the current target bumper;
4. Draw the paddle.
```

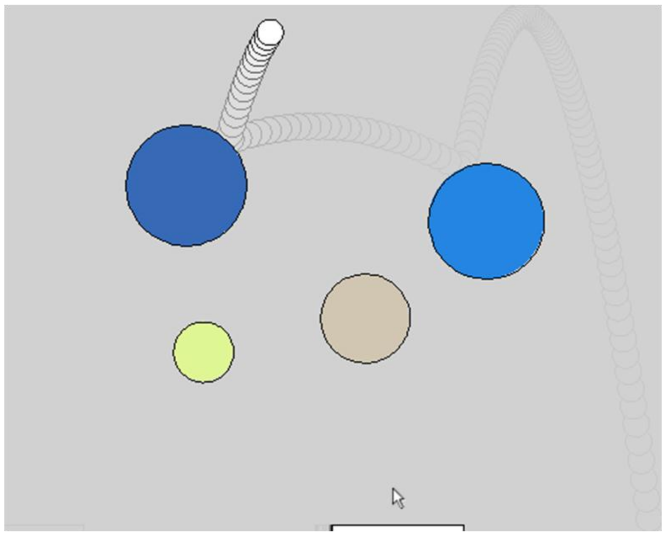
The key change in this algorithm is the repetition statement that implements the bouncing. The moving ball can, at any frame in the animation, come into contact with any of the target bumpers, so this algorithm visits each target bumper and checks to see if the pinball is hitting that target. If it is, the program asks the pinball to run its `collide()` method with respect to the given target bumper. Because the target bumpers don't move, this algorithm never needs to ask them to run their collision method.

The code that implements this final iteration is shown here with the output shown below.

```
/**
 * PinBall extends bounceBall to a simple pinball game with
 * some random balls with fixed positions serving as targets.
 * The main ball bounced off the walls, paddle and targets.
 *
 * @author kvlinden
 * @version Fall, 2009
 */

final int TARGET_COUNT = 4, BALL_RADIUS = 10,
          PADDLE_WIDTH = BALL_RADIUS * 10;
final int WIDTH=500, HEIGHT=400;

Ball[] targets;
```

```
Paddle paddle;

void setup() {
  size(WIDTH, HEIGHT);
  background(255);
  targets = new Ball[TARGET_COUNT];
  int targetRadius;
  for (int i = 0; i < targets.length; i++) {
    targetRadius = (int)random(BALL_RADIUS * 2, BALL_RADIUS * 5);
    targets[i] = new Ball(random(targetRadius, WIDTH - targetRadius),
                          random(targetRadius, HEIGHT - targetRadius * 2),
                          targetRadius);
  }
  paddle = new Paddle(WIDTH / 2, PADDLE_WIDTH);
}

void draw() {
  // Slowly whitewash the previous frames.
  noStroke();
  fill(200, 25);
  rect(0, 0, WIDTH, HEIGHT);

  // Process the moving pinball.
  ball.render();
  ball.move(paddle);

  // Process the target bumpers.
  for (int i = 0; i < targets.length; i++) {
    targets[i].render();
    // Bounce the ball off the target if appropriate.
    if (ball.hitsTarget(targets[i])) {
      ball.collide(targets[i]);
    }
  }

  // Draw the paddle.
  paddle.render();
}

void mouseMoved() {
  paddle.move(mouseX);
}
```

The implementation of `ball.hitsTarget()` method is left as an exercise.



9.6. Object-Oriented Programming

This section has demonstrated the power provided by the object-oriented principle of encapsulation. While it would certainly have been possible to implement the games shown in this chapter without the use of classes, it is clear that abstraction provided by encapsulation drastically simplified the complexity of implementing multiple interacting objects.

Consider, for example, the relative simplicity of the game engine from our final prototype. The Ball and Paddle classes are responsible for implementing the complex behaviors of the objects themselves, which makes the main game loop relatively simple because it is only responsible for implementing the rules of the game and running the animation.

We will defer to a later section the discussion of the additional power gained through the use of the other two object-oriented principles: inheritance and polymorphism.