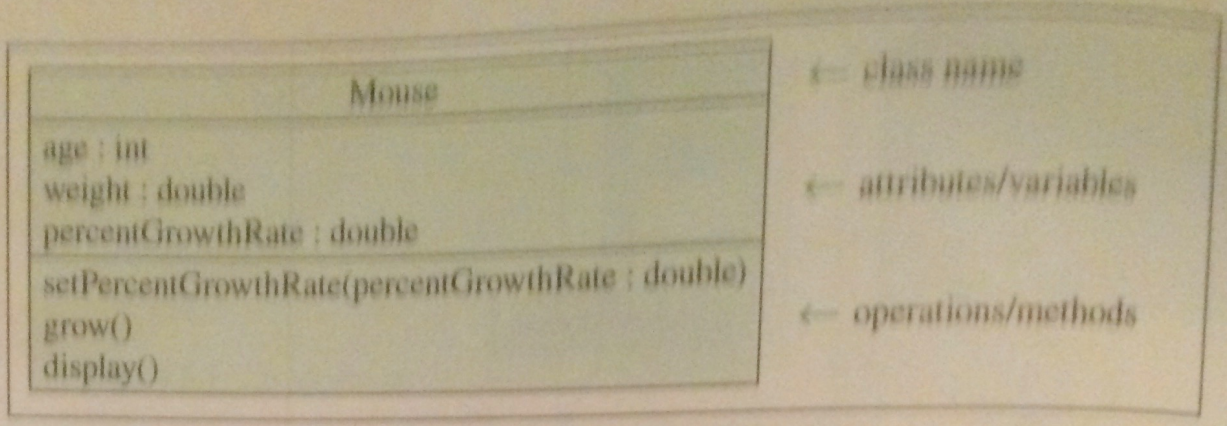

Figure 6.3 Abbreviated UML class diagram for a `Mouse` class

UML Class Diagram

See Figure 6.3. It contains an abbreviated UML class diagram for a `Mouse` class. A UML class diagram b is divided into three parts—class name at the top, *attributes* in the middle, and *operations* at the bottom. Wi Java programs, attributes equate to variables and operations equate to methods. Henceforth, we'll use the Jav terms, *variables* and *methods*, rather than the formal UML terms, *attributes* and *operations*. Collectively, w refer to a class's variables and methods as the class's *members*. Let's now describe each `Mouse` member.

The `Mouse` class has three instance variables—`age`, `weight`, and `percentGrowthRate`. The `age` instance variable keeps track of how old a `Mouse` object is, in days. The `weight` instance variable keeps track of a `Mouse` object's weight, in grams. The `percentGrowthRate` instance variable is the percentage of its current weight that gets added to its weight each day. If the `percentGrowthRate` is 10 percent and the mouse's current weight is 10 grams, then the mouse gains 1 gram by the next day.

The `Mouse` class has three instance methods—`setPercentGrowthRate`, `grow`, and `display`. The `setPercentGrowthRate` method assigns a specified value to the `percentGrowthRate` instance variable. The `grow` method simulates one day of weight gain for a mouse. The `display` method prints a mouse's age and weight.

Referring to Figure 6.3, note how we specify variable types in a class diagram. The type appears at the right of the variable (e.g., age : int). That's opposite from Java declarations, where we write the type at the left of the variable (e.g., `int age;`)

Start documenting early.

Some programmers use UML class diagrams as a means to document programs after they've already been written. That's OK, but it's not how class diagrams were originally intended to be used. We encourage you to start drawing class diagrams as a first step in your solution implementation. The class diagram details provide an outline for your program. Depending on the complexity of the program and your affinity for pseudocode, you may want to code the methods directly with Java or you may want to code the methods first, with pseudocode as an intermediate step. For our `Mouse` example, the `Mouse` class's methods are straightforward, so we'll code them directly with Java. Let's now take a look at the `Mouse` class's Java source code.

Mouse Class Source Code

Figure 6.4 shows the `Mouse` class implemented with Java. Note the `Mouse` class's three instance variable declarations for `age`, `weight`, and `percentGrowthRate`. Instance variables must be declared outside all methods, and to make your code more self-documenting, you should declare them all at the beginning of the class definition. Instance variable declarations are very similar to variable declarations you've seen in the past: The variable's type goes at the left of the variable, and you can optionally assign an initial value to the variable. Do you remember what it's called when you assign a value to a variable as part of a declaration?

```
/*******************************************************************
* Mouse.java
* Dean & Dean
*
* This class models a mouse for a growth simulation program.
*******************************************************************/

public class Mouse
{
  private int age = 0;                    // age of mouse in days
  private double weight = 1.0;            // mouse weight in grams
  private double percentGrowthRate;       // increase per day

  //*****************************************************************

  // This method assigns the mouse's percent growth rate.

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //*****************************************************************

  // This method simulates one day of growth for the mouse.

  public void grow()
  {
    this.weight +=
      (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  } // end grow

  //*****************************************************************

  // This method prints the mouse's age and weight.

  public void display()
  {
    System.out.printf("Age = %d, weight = %.3f\n",
      this.age, this.weight);
  } // end display
} // end class Mouse
```

instance variable declarations

parameter

To access instance variables, use this dot.

method body

Figure 6.4 Mouse class

```
/*********************************************
* MouseDriver.java
* Dean & Dean
*
* This is a driver for the Mouse class.
*********************************************/

import java.util.Scanner;

public class MouseDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double growthRate;
    Mouse gus = new Mouse();
    Mouse jaq = new Mouse();

    System.out.print("Enter % growth rate: ");
    growthRate = stdIn.nextDouble();
    gus.setPercentGrowthRate(growthRate);
    jaq.setPercentGrowthRate(growthRate);
    gus.grow();
    jaq.grow();
    gus.grow();
    gus.display();
    jaq.display();
  } // end main
} // end class MouseDriver
```

creation of two Mouse objects

Figure 6.5 `MouseDriver` class that drives the `Mouse` class in Figure 6.4

boxes immediately to the right of `gus` and `jaq` represent addresses. So `gus`'s little box holds the address of the first object.

Industry OOP Vernacular

Most Java programmers in industry don't use the term reference variable. Instead, they just use the term object. This blurs the distinction between reference variables and objects. For example, in the `MouseDriver` class in Figure 6.5, this statement initializes the `gus` reference variable:

```
Mouse gus = new Mouse();
```

Even though it's a reference variable, most industry Java programmers would refer to `gus` as an object. Despite the common practice of using the word "object" as a substitute for "reference variable," it's important to know the difference—an object holds a group of data, and a reference variable holds the location where that group of data is stored in memory. Understanding the difference between an object and a reference variable will help you to understand the behavior of Java code.

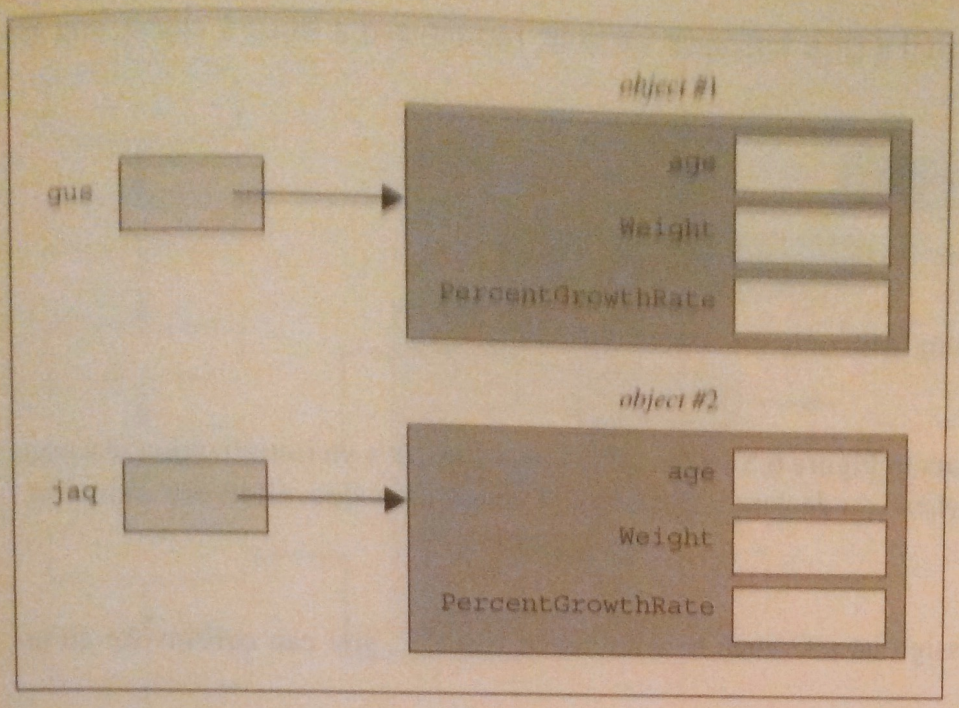

Figure 6.6 Reference variables and objects for the Mouse program in Figures 6.4 and 6.5
The two reference variables on the left, `gus` and `jaq`, contain references that point to the two objects on the right.

Declaring a Reference Variable

You must always declare a variable before you can use it. For example, in order to use an `int` variable named `count`, you must first declare `count` like this:

```
int count;
```

Likewise, in order to use a `gus` reference variable, you must first declare `gus` like this:

```
Mouse gus;
```

As you can see, the process for declaring reference variables mirrors the process for declaring primitive variables. The only difference is that instead of writing a primitive type on the left (e.g., `int`), for reference variables, you write a class name on the left (e.g., `Mouse`).

Instantiation and Assigning a Value to a Reference Variable

As you know, the point of a reference variable is to store a reference to an object. But before you can store a reference to an object, you have to have an object. So let's look at object creation.

To create an object, use the `new` operator. For example, to create a `Mouse` object, specify `new Mouse()`. The `new` operator should make sense when you realize that `new Mouse()` creates a new object. The formal term for creating an object is *instantiating* an object. So `new Mouse()` instantiates an object. The term *instantiate* is a verbalized form of the noun "instance." It is computer jargon for "make an instance of a class" or "create an object."

After instantiating an object, you'll normally assign it to a reference variable. For example, to assign a `Mouse` object to the `gus` reference variable, do this:

```
gus = new Mouse();
```

After the assignment, `gus` holds a reference to the newly created `Mouse` object.

Are you bothered by the lack of parentheses around the `return` statement's returned expression? With statements that use a condition (`if` statement, `while` statement, etc.), the condition must be surrounded by parentheses. With the `return` statement's returned expression, the parentheses are optional. You'll see it both ways in industry—sometimes parentheses are included and sometimes they're omitted.

Here's how the `isAdolescent` method could be used in a calling module:

```
Mouse pinky = new Mouse();
. . .
if (pinky.isAdolescent() == false)
{
  System.out.println("The mouse's growth is no longer" +
    " being simulated - too old.");
}
```

Do you know how the above `if` statement can be shortened? Here's a functionally equivalent `if` statement with an improved condition:

```
if (!pinky.isAdolescent())
{
  System.out.println("The mouse's growth is no longer" +
    " being simulated - too old.");
}
```

The goal is to print the warning message if `pinky` is old (not an adolescent). If `isAdolescent` returns `false` (indicating an old Pinky), then the `if` statement's condition is `true` (`!false` evaluates to `true`) and the program prints the warning message. On the other hand, if `isAdolescent` returns `true` (indicating a young Pinky), then the `if` statement's condition is `false` (`!true` evaluates to `false`) and the program skips the warning message.

Although the shortened-version `if` statement might be harder to understand initially, experienced programmers would prefer it. Following that lead, we encourage you to use `!` rather than `== false` for similar situations.

6.13 Problem Solving with Simulation (Optional)

In our previous mouse examples, to keep the focus on OOP concepts rather than mouse growth details, we used a simplistic growth formula. In this section, we show you how to simulate growth in a way that is much closer to the kind of growth that occurs in the real world. Then we show you a simple trick that can be applied to many simulation problems to improve the program's speed and accuracy greatly.

Previously, we modeled growth by assuming that added weight is proportional to weight, like this:

$$addedWeight = fractionGrowthRate \times weight$$

where

$$fractionGrowthRate = .01 \times percentGrowthRate$$

This kind of growth makes weight increase exponentially and continue to curve upward in time, as indicated by Figure 6.16. This is a good approximation for a young plant or animal, where most of the ingested energy goes into new growth.

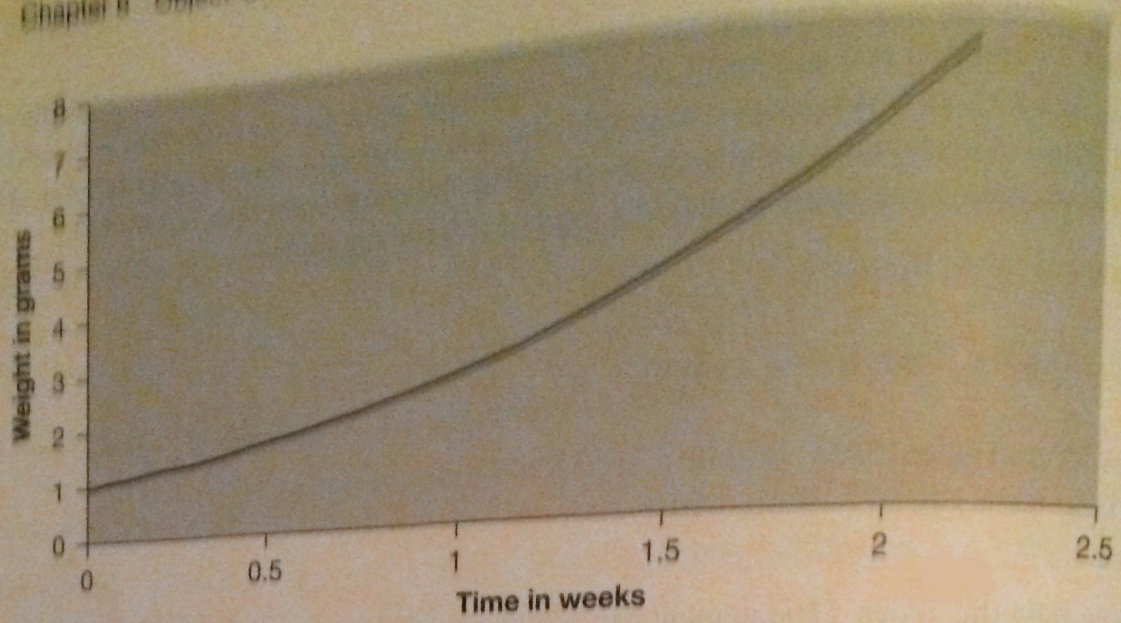

Figure 6.16 Exponential growth

Maturation

But there's a problem with the exponential growth model. Nothing keeps growing forever! After a while old tissue starts to die, and some of the ingested nutrients must be used to replace the old tissue instead of just adding to it. This slows the growth. As a larger fraction of ingested nutrients go into replacement, the growth curve straightens out, begins to bend the other way, and approaches a maximum. The easiest way to modify the basic exponential growth formula to make it describe maturation is to multiply by another factor to obtain what's called the *logistic equation:*

$$\texttt{addedWeight} = \texttt{fractionGrowthRate} \times \texttt{weight} \times \left(1.0 - \frac{\texttt{weight}}{\texttt{maxWeight}}\right)$$

A quick inspection of this improved growth formula shows that as `weight` approaches `maxWeight`, the quantity in parentheses on the right approaches zero, and therefore the added weight on the left approaches zero. At that point, there's no more growth. This provides a reasonable description of an organism reaching maturity.

Computer simulations rely on approximate mathematical models, like the model provided by the above logistic equation. Such simulation models are sometimes good, sometimes not so good, and it's difficult to know how good they are without comparing them to actual live data. But for the current weight gain problem, we have the luxury of being able to compare the simulation model with an exact mathematical model. Here is a closed-form exact mathematical solution that determines the weight of any given time:

$$\texttt{weight} = \frac{1.0}{\frac{1.0}{\texttt{maxWeight}} + e^{-(\texttt{fractionGrowthRate} \times \texttt{time} + g_0)}}$$

This formula contains a growth constant, g_0, which is

$$g_0 = \log_e\left(\frac{\texttt{minWeight}}{1.0 - \frac{\texttt{minWeight}}{\texttt{maxWeight}}}\right)$$

You can find g_0 by plugging *minWeight* and *maxWeight* values into the second formula. Then find *weight* by plugging g_0 into the first formula.

Simulation

Usually an exact solution is not available, and the only way to solve a problem is with a simulation. But for this weight gain problem, we have both. Let's look at a program that

If you can describe it, you can simulate it.

```
/**************************************************************************
* Growth.java
* Dean & Dean
*
* This provides different ways to calculate growth.
**************************************************************************/

public class Growth
{
  private double startSize;                  // initial size
  private double endSize;                    // maximum size
  private double fractionGrowthRate;         // per unit time

  //************************************************************************

  public void initialize(double start, double end, double factor)
  {
    this.startSize = start;
    this.endSize = end;
    this.fractionGrowthRate = factor;
  } // end initialize

  //************************************************************************

  public double getSize(double time)
  {
    double g0 = Math.log(startSize / (1.0 - startSize / endSize));

    return 1.0 / (1.0 / endSize +
      Math.exp(-(fractionGrowthRate * time + g0)));
  } // end getSize

  //************************************************************************

  public double getSizeIncrement(double size, double timeStep)
  {
    return fractionGrowthRate *
      size * (1.0 - size / endSize) * timeStep;
  } // end getSizeIncrement
} // end class Growth
```

Figure 8.17 Growth class, which implements different ways to evaluate growth

displays time, the exact solution, and the simulated solution together. See the program's output in Figure 6.17.

The `Growth` class has three instance variables, `startSize`, `endSize`, and `fractionRate`, and three methods. The `initialize` method initializes the three instance variables. The `getSize` method uses the closed-form mathematical solution formula provided earlier. It returns the size (e.g., mouse weight) at the given time. Notice that this method's name starts with "get," so it looks like an accessor method, and it returns a `double` value just as our previous `getWeight` method does. But this class does not have any instance variable named "size." So here's an example of a method that's really an accessor like the accessors described in Section 6.12, even though its name makes it look like an accessor. The point is this: any method can return a value, not just an accessor method, and any method can have any name that seems appropriate—`getSize` is simply the most appropriate name we could think of for this method that computes and returns a size.

The `getSizeIncrement` method implements one simulation step. It returns the change in size between the current time and the next time. Notice that the `getSize` and `getSizeIncrement` methods do different things. The first one gives the answer directly. The second one gives an incremental value that must be added to a previous answer to get the next answer.

If you are writing your own class and you want to model the growth of one of your class's entities, you could copy and paste the `Growth` class's variables and methods into your class. Alternatively, you can delegate the work to a `Growth` class object just as you delegate work to `Scanner` class objects. To do so, use `new` to instantiate a `Growth` object, initialize it with the growth-related data in your object. Then ask the `Growth` object to solve the growth problem for you by calling its `getSize` or `getSizeIncrement` method. In your program, you could use code like that in the `main` method of the `GrowthDriver` class in Figure 6.18.

This driver class may seem imposing, but it's not difficult. We start by declaring and initializing variables, and this includes instantiating and initializing a `Growth` object. Then we ask the user to provide a time increment and the total number of time increments. Finally, we use a `for` loop to print time, the exact solution, and the simulated solution for each time step. If you run the program composed of the classes in Figures 6.17 and 6.18, you'll get this result:

Sample session:

```
Enter time increment: 1
Enter total time units to simulate: 15
        exact  simulated
time     size    size
 0.0      1.0     1.0
 1.0      2.6     2.0
 2.0      6.4     3.9
 3.0     13.6     7.3
 4.0     23.3    13.3
 5.0     31.7    22.2
 6.0     36.5    32.1
 7.0     38.6    38.4
 8.0     39.5    39.9
 9.0     39.8    40.0
10.0     39.9    40.0
```

```
11.0    40.0    40.0
12.0    40.0    40.0
13.0    40.0    40.0
14.0    40.0    40.0
15.0    40.0    40.0
```

```
/*****************************************************************************
* GrowthDriver.java
* Dean & Dean
*
* This compares exact and simulated solutions for growth.
*****************************************************************************/

import java.util.Scanner;

public class GrowthDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double timeStep;
    double timeMax;
    Growth entity = new Growth();
    double startSize = 1.0;              // weight in grams
    double endSize = 40.0;               // weight in grams
    double fractionGrowthRate = 1.0;     // per unit time
    double size = startSize;

    entity.initialize(startSize, endSize, fractionGrowthRate);
    System.out.print("Enter time increment: ");
    timeStep = stdIn.nextDouble();
    System.out.print("Enter total time units to simulate: ");
    timeMax = stdIn.nextDouble();
    System.out.println("       exact  simulated");
    System.out.println("time    size    size");

    for (double time=0.0; time<=timeMax; time+=timeStep)
    {
      System.out.printf("%4.1f%8.1f%8.1f\n",
        time, entity.getSize(time), size);
      size += entity.getSizeIncrement(size, timeStep);
    } // end for
  } // end main
} // end class GrowthDriver
```

Instantiate Growth object.

Initialize Growth object.

Figure 6.18 GrowthDriver class that demonstrates the Growth class in Figure 6.17