5 Selective Control and Advanced Animations

Introduction

An important feature of any application program is its ability to make decisions. For example, programmers expect their users to make mistakes. The reality is that to err is human nature, so software must be designed to recover from mistakes. Software, in general, is full of choices for users and is typically designed to allow a user to navigate a path based on these choices. For example, when computing wages for an employee, a decision needs to be made as to whether some of the hours worked will be eligible for an overtime rate. Decision making is a regular characteristic of software applications.

Selective control is a process in which some condition is checked and a decision is made about whether a certain segment of the program code will execute. For example, if an employee works overtime, the calculation to compute his pay may be different than if the same person was a salaried employee. Statements that permit a program to make decisions are called selective control structures. They provide much of the power and simplicity of a programming language and allow us to write meaningful programs. When combined with assignment statements, these selective control structures offer a potent assortment of programming constructions. This chapter looks at the two AS3 selective control structures: `if` statements and `switch` statements.

The foundation of selective control is the subject called Boolean logic. The name "Boolean" is bestowed in honor of the English mathematician George Boole, who pioneered the mathematical theory that now bears his name. A variable of the data type `Bool` can store a value of either true or false. In AS3, as with most programming languages, `true` and `false` are reserved keywords. In this chapter, we will examine relational operators and learn how to form Boolean expressions, also called Boolean test conditions, by using logical operators.

Boolean logic is important to selective control structures because test conditions are defined using Boolean expressions.

5.1 Boolean Logic

Simply speaking, Boolean logic is used to represent expressions that have two possible values: true or false. These expressions are often constructed with, but not limited to, relational operators, arithmetic operators, and logical operators.

5.1.1 Relational Operators

In arithmetic, number values can be compared using equalities (==) and inequalities (<, >, <=, >=, and so on). All programming languages provide for the comparison of number values. The operators used for comparison are called **relational operators**. Table 5-1 lists the six relational operators used by AS3.

TABLE 5-1 AS3 Relational Operators

Relational Operator	Meaning
`==`	Is equivalent to
`<`	Is less than
`>`	Is greater than
`<=`	Is less than or equal to
`>=`	Is greater than or equal to
`!=`	Is not equivalent to

When two numbers or variable values are compared using a single relational operator, the expression is referred to as a *simple Boolean expression*. Each simple Boolean expression has the Boolean value of true or false according to the validity of the expression. Data of the same general type can be compared; thus numbers can be compared with each other, and strings can be compared with strings. Strings and numbers, however, cannot be compared.

Here are two examples of simple Boolean expressions.

Example 1

Evaluate the value of a simple Boolean expression.

```
17 != 5
```

Evaluation

This Boolean statement expresses that the number 17 is not equivalent to the number 5. This is clearly true.

The value returned by this expression is true.

Example 2

```
4 < (3 + 2)
```

Evaluation

Arithmetic operators, such as +, -, and *, can also be used in simple Boolean expressions. The value returned by this example is true.

5.1.2 Priority Levels of Relational Operators and Arithmetic Operators

The evaluation of an expression that uses both arithmetic operators and relational operators necessitates recognition of priority level. Among arithmetic and relational operators, there are three levels of priority. The relational operators have the lowest priority and are always evaluated last.

Table 5-2 summarizes the priority of these operations. Among arithmetic and relational operators, the relational operators are always evaluated last. Operators of the same priority level are evaluated in order from left to right.

TABLE 5-2 Priority Levels of Relational Operators Compared with Arithmetic Operators

	Priority	Operator
Highest Priority	Level 1	* / %
	Level 2	+ -
Lowest Priority	Level 3	== > < <= >= !=

Example 1

```
4 * 5 != 17 + 3
```

Evaluation

In this example, the expressions on both sides of the relational operator are evaluated first. Because both sides evaluate to 20, this statement evaluates to false.

Example 2

```
14 + 3 * 5 <= 17 + 30 / 4 - 20
```

Evaluation

This example shows an expression that is difficult to evaluate. Although parentheses are not required, sometimes it is a good idea to use them to increase the readability of an expression. Parentheses may also help you avoid using an incorrect expression.

Using precedence priorities, the left side evaluates to 29, while the right side evaluates to 4. Thus this Boolean expression evaluates to false.

5.1.3 Logical Operators

Simple Boolean expressions, such as the ones shown in the previous examples, can be combined to form compound Boolean expressions. This is done by using logical connectives and negation. The logical connectives used by AS3 are `&&` (for AND) and `||` (for OR). Negation is represented by the symbol `!` (for NOT). These three reserved symbols (`&&`, `||`, and `!`) are called logical operators.

The && Operator (AND) To understand the `&&` operator, we first define two simple Boolean expressions `P` and `Q`. As shown here, `&&` is used to express the conjunction of `P` and `Q`. `P && Q` is true only when `P` is true and `Q` is true.

P	Q	P && Q
true	true	**true**
true	false	**false**
false	true	**false**
false	false	**false**

The || Operator (OR) The logical operator `||` is used to express the disjunction of two simple Boolean expressions in which the resulting compound expressions are true if either or both of the expressions are true. As shown here, `P || Q` is true when `P` is true, `Q` is true, or both `P` and `Q` are true.

P	Q	P || Q
true	true	**true**
true	false	**true**
false	true	**true**
false	false	**false**

The ! Operator (NOT) The NOT operator `!` produces the logical negation of an expression; thus not true is false and not false is true. As shown here, this operator is a unary operator and is not used to join simple Boolean expressions.

P	!P
true	**false**
false	**true**

5.1.4 Priority Levels for All Operators

The priority levels of all operators are listed in Table 5-3. Note that the logical operators do not share the same level of priority. As a unary operator, `!` has the highest possible priority among all operators, including the arithmetic and relational operators. It is also important to note that `&&` has a higher priority than `||`. As with arithmetic and relational operators, `&&` operators are evaluated from left to right, as are `||` operators.

TABLE 5-3 Priority Levels of All Operators

	Priority	Operator
Highest Priority	Level 1	!
	Level 2	*, /, %
	Level 3	+, -
	Level 4	== >, < <=, >= !=
	Level 5	&&
	Level 6	||
Lowest Priority	Level 7	= +=, -= /=, *=, %=

Priority levels are illustrated in the following six examples. For the first four examples, assume that `P`, `Q`, and `R` are Boolean expressions with the values true, true, and false, respectively.

Example 1

```
P && !Q
```

Evaluation

In this example, the logical negation of `Q` is performed first because `!` has the highest priority. The successive steps are shown here.

	P	&&	!Q
Step 1	T	&&	!T
Step 2	T	&&	F
Final value		false	

Note

P and Q are simple expressions. The complete expression `P && !Q` is referred to as a compound Boolean expression.

Example 2

```
P && Q || !R
```

Evaluation

In this expression, the logical negation of `R` is performed first. Next, the logical AND is performed linking `P` and `Q`. Finally, the results of these two operations are linked together with logical OR. The successive steps in the evaluation of the Boolean expression are shown here.

	P	&&	Q	||	!R
Step 1	T	&&	T	||	!F
Step 2	T	&&	T	||	T
Step 2		T		||	T
Final value		true			

Example 3

```
P && ! (Q || R)
```

Evaluation

In this example, a set of parentheses is used to alter the order of priority. The logical operation `||` is first used to link `Q` and `R`. This result is negated to produce a value of false and then linked with `P` using the logical operation `&&`. The successive steps are as follows.

	P	**&&**	**!(Q**	**||**	**R)**
Step 1	T	&&	!(T	||	F)
Step 2	T	&&		!T	
Step 2	T	&&		F	
Final value		false			

Example 4

```
P || Q && R
```

Evaluation

Because the `&&` operator has a higher priority than `||`, `Q && R` is evaluated first, followed by the `||` logical operation. The successive steps in the evaluation of the Boolean expression are as follows.

	P	**||**	**Q**	**&&**	**R**
Step 1	T	||	T	&&	F
Step 2	T	||	F		
Final value		false			

When logical operators are used with relational expressions, parentheses are not required but are often helpful. When complex compound expressions are being evaluated, the logical operators, arithmetic expressions, and relational operators are evaluated during successive passes through the expression. The next set of examples illustrates the evaluation, construction, and common errors found in compound Boolean expressions.

Example 5

Task

Write a Boolean statement expressing that the values stored in the `Number` variables named `n1` and `n2` are either both positive or both negative.

Solution

```
n1 > 0 && n2 > 0 || n1 < 0 && n2 < 0
```

Evaluation

The solution requires a compound Boolean statement. The first component expresses that `n1` and `n2` are both positive. The second component expresses that `n1` and `n2` are both negative. If the first or second component of the expression is true, then the entire Boolean expression is true. Due to the fact that `&&` is evaluated first, no parentheses are required.

Note

Only if the first component of this statement is false will the second component be tested.

Example 6

Task

Write a Boolean statement expressing that values stored in `n1` and `n2` are not both zero.

Solutions

a. `n1!= 0 && n2!= 0`

b. `n1 && n2`

Evaluation

There are actually more than two solutions to express this condition. The two shown here are the most clear-cut.

a. This Boolean statement uses `!=`, the "is not equivalent" operator, to express the condition that both `n1` and `n2` are not zero.

b. Of the two solutions, this one is the least complicated. It plainly exploits the condition that any number other than zero is considered to be true.

5.2 Introduction to `if` Statements

Boolean expressions are used to identify whether a condition is true. An `if` statement provides the mechanism for controlling the execution of program statements based on this condition. This control structure gives AS3 the capacity to make a decision. In other words, programmers can use `if` statements to examine the existence of a condition and then select the appropriate course of action best suited to that condition. Because it is used in this way, an `if` statement is referred to as a selection control mechanism.

We begin with the general form of the `if` statement.

```
if (Boolean expression) {
        statement 1;
        statement 2;
        .
        .
        .
        statement n;
}
```

An `if` statement begins with the keyword `if`, written in lowercase, followed by a Boolean expression, which must be enclosed in parentheses. The list of statements to be executed, contained within the {}, is referred to as the set of actions. AS3 requires the set of actions be enclosed in {}. To ensure better readability, the set of action statements should be indented. In the decision structure's simplest form, a specific action, or set of actions, is taken only when a specific condition exists. As shown in Figure 5-1, when the condition is found to be false, the set of actions statements is not performed, but instead is completely bypassed.

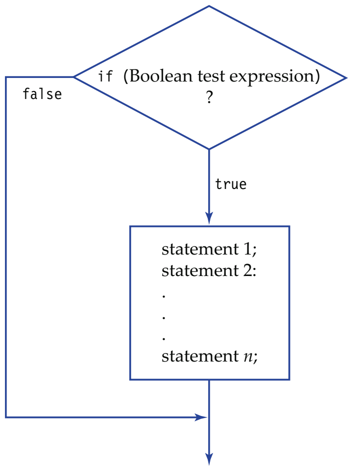


FIGURE 5-1 Flowchart visually depicting how the `if` statement works.

The following examples illustrate how the `if` statement works.

Example 1

```
if (day == WEEKEND){
   trace ( "Wear comfortable jeans.");
   trace ( "Put on sandals");
}
```

Evaluation

The Boolean expression on line 1 tests whether it is a weekend day. If this condition is true, then the set of actions consisting of "Wear comfortable jeans." and "Put on sandals." is executed. As shown in Figure 5-2, if it is not a weekend day, then the condition is false and the program flow follows another path and skips the set of actions.

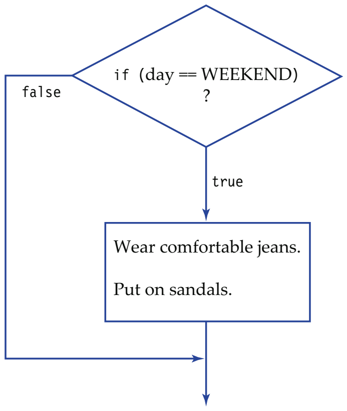


FIGURE 5-2 Diagram of the `if` statement flow of control.

Example 2

```
1  var age:int;
2  if (age < 35) {
3     trace ( "You are not old enough to be a U.S. president.");
4  }
5  trace ( "The youngest U.S. president was Kennedy at 42 years old.");
```

Evaluations for Different Values of ~~Age~~

a. `age = 25`

In this scenario, the conditional expression on line 2 (`age < 35`) is true and, therefore, the action statement on line 3 is executed. The statement on line 5 is not located inside the { } and, therefore, is an unconditional statement. Unconditional statements do not depend on a Boolean expression being true or false and will automatically be executed.

The output produced by this segment of code is

```
You are not old enough to be a U.S. president.
The youngest U.S. president was Kennedy at 42 years old.
```

b. `age = 36`

The condition expression for this given age is false. The action statement is not executed. The unconditional statement will still execute, however.

The output produced by this segment of code is

```
The youngest U.S. president was Kennedy at 42 years old.
```

Example 3

What is the problem in the following segment of code?

```
1  var  n1:int = 3;
2  var  n2:int = 7;
3  if (n1 = n2) {
4     trace("The value in n1 is  ", n1);
5  }
```

Evaluation

This example illustrates a common error when writing Boolean expressions: The operator used to express equality is incorrect. The relational operator is `==` and the assignment operator is `=`. As with most logic errors, this kind of typographical error can be difficult to identify because it does not hinder the application's ability to run. This error can be detected only during runtime testing of the application.

In this specific example, the Boolean expression on line 3 is an assignment statement. Recall that the number 7 is considered to be a true value because only zero is false. Because true is being assigned to `n1`, the result is a true expression.

The output produced by this segment of code is

```
The value in n1 is 7
```

Example 4

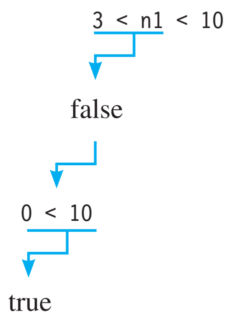
What is the problem in the following segment of code?

```
1  var n1:int = -5;
2  if (3 < n1 < 10) {
3     trace("Blue");
4  }
```

Evaluation

At first glance, the Boolean expression on line 2 appears to be written correctly—but a deceptive logical error is actually present. The goal is to express that the value in `n1` is within the range of 3 to 10. The problem is that this mathematical relationship cannot be represented in a programming language without using a logical operator. More to the point, this statement will always be true, no matter what value is stored in `n1`.

The first operation examines whether 3 is less than `n1`, which produces a value of false. Because false is represented by a zero, the value zero is used as the operand for the next operation. To show the illogical outcome of this expression, it is necessary to examine the sequence of steps.



The output produced by this segment of code is

```
Blue
```

The correct way to write this condition follows in Example 5.

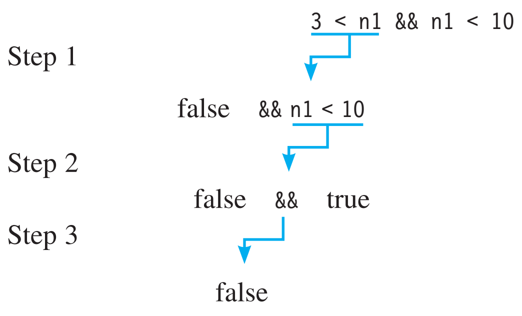
Example 5

```
1   var  n1:int = -5;
2   if (3 < n1  && n1 < 10){
3      trace("Blue");
4   }
```

Evaluation

Unlike the expression in Example 4, the Boolean expression on line 2 is written correctly. This mathematical relationship begins with an expression of what the lower bound of `n1` is. The logical operator `&&` is then used to link the lower bound logical definition with the upper bound definition.

Here is the sequence of steps.



No output is produced by this segment of code.

5.3 The `if–else` Statement

The `if` statements considered thus far involve selecting a single alternative. Another form of the `if` statement is one that contains an `else` clause and, therefore, offers the possibility of selecting one of two alternatives. The correct form and syntax for an `if-else` statement is

```
if (Boolean expression) {
        set of action statements 1;
}
else {
        set of action statements 2;
}
```

The flow of control when using an `if-else` statement is as follows:

1. The Boolean expression is evaluated.
2. If the Boolean expression is true, the first set of action statements following the expression is executed and control then exits the entire `if-else` statement structure.
3. If the Boolean expression is false, the set of action statements belonging to the `else` clause is executed and control then exits the `if-else` statement.

The next two examples illustrate how the `if-else` statement works.

Example 1

```
1  var rain:Boolean = false;
2  if (rain == true){
3     trace ( "Get an umbrella.");
4     trace ( "Wear boots.");
5  } else {
6     trace ( "Store the umbrella.");
7     trace ( "Wear sandals.");
8  }
```

Evaluation

As shown in Figure 5-3, the Boolean expression on line 2 tests whether it is raining. If this condition is true, the block of statements on lines 3 and 4 is executed. If the condition is false, the `else` clause on lines 6 and 7 will be executed and "Store the umbrella." and "Wear shoes" are output.

The Boolean expression is false. Thus the output produced by this segment of code is

```
Store the umbrella.
Wear sandals.
```

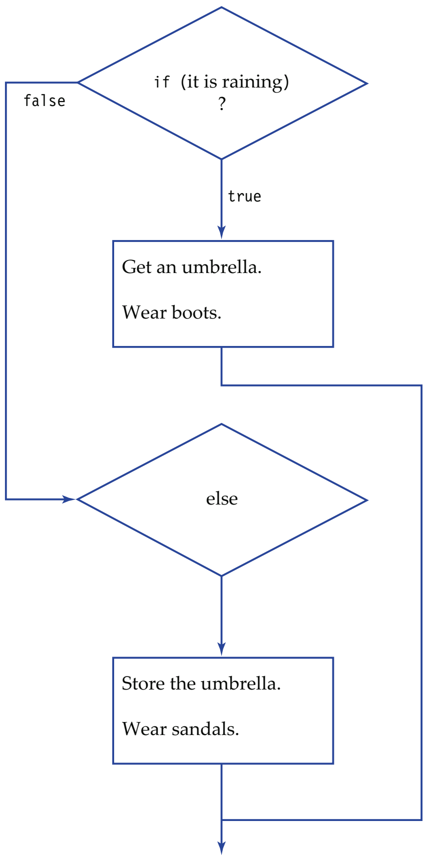


FIGURE 5-3 `if-else` statement flow of control.

Example 2

```
var Num1: Number = 2.2;
var Num2: Number = 0;
var value:Number;

if (Num2== 0){
   trace ( "Error.  Division by Zero ");
} else {
   value = Num1 / Num2;
   trace ( value);
}
trace("exit");
```

Evaluation

This segment of code employs the `if-else` statement to guard against division by zero. The Boolean expression on line 5 is true; thus the output produced by this segment of code is that shown below. The final statement on line 11 is not dependent on the Boolean expression. Because it is unconditional, it will always execute.

```
Error.  Division by Zero
exit
```

5.4 The `if–else if–else` Statement

Oftentimes, decisions need to be made that involve more than two alternatives. It is possible to use the `if–else` structure to formulate a selective statement exactly for this purpose. This type of selective statement will take the form `if–else if–else if–else`, with each clause selecting its own appropriate course of action. This is not a new kind of `if` statement, but rather a collection of `if-else` statements.

Here is a generic version of the `if` statement that contains five alternatives. In this format, statement 1 will execute only if Boolean expression 1 is found to be true, and statement 2 will execute only if Boolean expression 2 is true, and so on. Each `if` in an `else if` clause is actually a new `if` statement and, therefore, is executed only if the conditions defined in all the preceding Boolean expressions are false. Similarly, each `else` in an `else if` clause is actually associated with the `if` of the preceding `else if` clause (or the first `if`).

```
if (Boolean expression1) {
        statement 1;
}else if (Boolean expression2) {
        statement 2;
}else if (Boolean expression3) {
        statement 3;
}else if (Boolean expression4) {
        statement 4;
}else {
        statement 5;
}
```

The next three examples examine these kinds of selective statements.

Example 1

```
1  var n1: Number ;
2  if (n1 == 0){
3     trace ( "Zero");
4  } else  if (n1 % 2 == 1){
5     trace ( "Odd ");
6  } else  {
7     trace ( "Even");
8  }
9  trace ( "Number");
```

Evaluations for Different Values of `n1`

In this example, the value stored in `n1` can either be zero, an odd number, or an even number. The `if` statement uses an `if–else if–else` format to test each possibility. Each Boolean expression will be tested in sequential order. Once it reaches a condition that is true, it executes the action statement associated with that condition and then exits the structure. The statement `trace ("Number");` on line 9 is unconditional and not dependent on any `if` statement. It will always execute.

a. `n1 = 0`

   The output produced by this segment of code is

```
Zero
Number
```

b. `n1 = 44`

   The output produced by this segment of code is

```
Even
Number
```

c. `n1 = 57`

   The output produced by this segment of code is

```
Odd
Number
```

Example 2

Task

Given the following grading scale, write a segment of code to display the correct grade for an exam score. Assume the value stored in the variable holding the exam score is valid.

Grade	Exam Score
A	Greater than or equal to 90
B	80–89
C	70–79
D	60–69
F	Less than 60

Solution

By using an `if–else if–else if–else` structure, this task can be greatly simplified. A well-designed selective structure should avoid unnecessary test conditions. For example, the letter grade A is displayed when (`score >= 90`) is true, as shown in line 2. If this condition is false, it is unnecessary to test for the second part of (`score >= 80 && score < 90`), because `score < 90` is automatically implied. A well designed `if–else if–else if–else` is written as follows:

```
1  var score: Number ;
2  if (score>= 90){
3     trace ( "Grade: A ");
4  } else  if (score>= 80){
5     trace ( "Grade: B ");
6  } else  if (score>= 70){
7     trace ( "Grade: C");
8  } else  if (score>= 60){
9     trace ( "Grade: D");
10 } else  {
11    trace ( "Grade: F ");
12 }
```

Evaluations for Different Values of `score`

a. `score = 55`

The only Boolean condition that is true for this scenario is the default test, the `else` clause. All of the previous Boolean expressions are false.

The output produced by this segment of code is

```
Grade: F
```

b. `score = 75`

The program will examine each of the Boolean conditions in sequential order. Once it reaches a Boolean condition that is true, it executes the action statement associated with that condition and then exits the structure.

The sequential conditions are evaluated as follows:

Test 1	`if (score >=90)`	false	
Test 2	`if (score >=80)`	false	
Test 3	`if (score >=70)`	true	`trace ( "Grade: C");`

The output produced by this segment of code is

```
Grade: C
```

Example 3

```
1  var score: Number ;
2  if (score>= 70){
3     trace ( "Grade: C ");
4  } else  if (score>= 80){
5     trace ( "Grade: B ");
6  } else if (score>= 90){
7     trace ( "Grade: A");
8  }
```

Evaluation

The objective of this segment of code is similar to that of Example 2, which is to display the letter grade that corresponds with the numeric score. However, the order of these conditions creates a logic error such that all scores of 70 or higher are given a grade of ‘C’ and any score less than 70 will not receive a grade. Upon close examination, you will see that the condition on line 4 is evaluated only if `score` is less than 70.

There are several ways to solve this problem, two of which are identified here. The first option is of poor quality, however.

Option 1 utilizes independent `if` statements, even though the Boolean conditions are unmistakably related to each other. This option is inefficient because all test conditions will be evaluated regardless of whether a preceding condition is found to be true. This option should not be considered because of its poor quality.

Option 2 is not only correct, but also of good quality; it is concise and clear. The code and order of the conditions have been refined by eliminating all unnecessary Boolean expressions.

Option 1: Poor Quality

```
if (score>= 70 && score < 80){
   trace ( "Grade: C ");
}
if (score>= 80 && score < 90){
   trace ( "Grade: B ");
}
if (score>= 90){
   trace ( "Grade: A");
}
```

Option 2: Excellent Quality

```
if (score <= 90){
   trace ( "Grade: A ");
} else if (score<= 80){
   trace ( "Grade: B ");
} else if (score>= 70 ){
   trace ( "Grade: C");
}
```

5.5 Case Study 1: The Game of Pong and Collision Detection

In 1972, Atari released a simple video game that featured two elemental objects: a bouncing ball and a paddle to hit the ball with. This game was called Pong. Its enormous popularity eventually led to the start of the video game industry. In this case study we will create a similar game, as shown in Figure 5-4.

Objectives of this case study:

1. Explore decision making.
2. Work with concepts of animation.
3. Program basic collision detection.

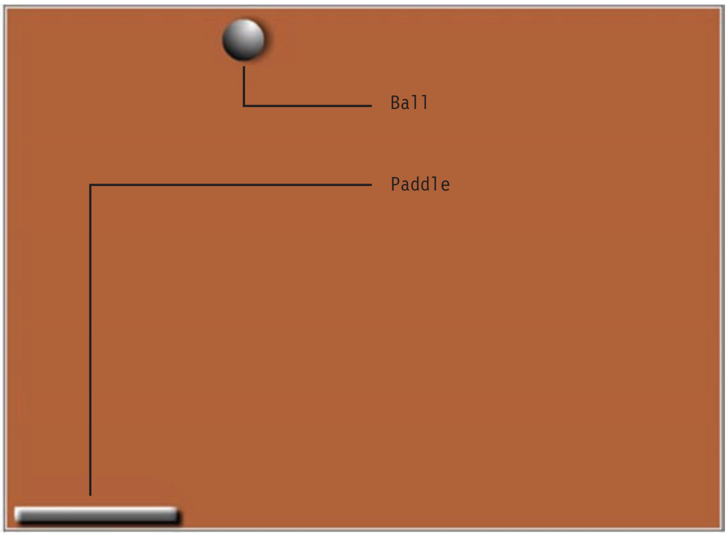


FIGURE 5-4 Pong application.

Game Plan

This game is a simple version of Pong. It is a single-player game that uses a single paddle and ball. At most, two display objects can be in motion at any given time—the ball and the paddle. The ball will move with a fixed velocity around the stage. When the ball collides with the top, left, or right wall of the stage, it will reverse its direction. The paddle will be controlled by the mouse and can move horizontally across the bottom of the stage. The player must stop the ball from going off the bottom of the stage by hitting it with the paddle.

Once the ball bounces below the bottom of the stage, the ball disappears and the game is over.

Visual Elements

At its most basic level, Pong is a highly instinctive game. This fact is significant for the visual design phase because it indicates that game-play instructions or even elaborate interface elements can be minimized or, as in this case study, skipped altogether. Our game of Pong will rely solely on a single screen during the entire game.

The two display objects, `Ball` and `Paddle` (shown in Figure 5-4), are both `MovieClip` instances. The main Timeline is organized using a single layer in a single frame. Both display objects, `Ball` and `Paddle`, are placed on the same layer. The frame rate for this animation is set to 30 frames per second, which is fast enough to produce smooth movements.

The display objects, `Ball` and `Paddle`, will utilize inherited properties such as `height`, `width`, `x`, and `y`. In addition, Ball will have two newly constructed properties, `xVelocity` and `yVelocity`. Table 5-4 lists all of the properties.

TABLE 5-4 Display Objects and Properties Used in the Pong Game

Display Object Name	Property
`Ball`	`height`: Height of `Ball`. `width`: Width of `Ball`. `x`: x-axis position on the Stage. `y`: y-axis position on the Stage. `xVelocity`: A newly constructed property governing the x-axis velocity. `yVelocity`: A newly constructed property governing the y-axis velocity.
Paddle	`x`: The `Paddle` `MovieClip` moves along the x-axis. Only the x-axis position on the Stage will be required.

Collision Detection

To design the algorithm, the details of the collisions need to be established. This game will use three boundary limits of the viewable screen: `TOP`, `LEFT`, and `RIGHT`. `TOP`, `LEFT`, and `RIGHT` represent the topmost, leftmost, and rightmost boundaries, espectively, of where the ball can move in the game area. `BOTTOM` will not be used in this case study, but will be explored further in an end-of-chapter problem.

To understand boundaries and collisions, we will assume the ball's registration point is located at its center. It is necessary to detect these collisions because the ball's velocity will be reversed once it collides with wall boundaries. As shown in Figure 5-5, the ball collides with `TOP` when its `y` position is less than or equal to zero (the topmost edge of the Stage) plus the ball's radius. In a similar fashion, the ball collides with the `RIGHT` boundary when its `x` location is equal to or exceeds the width of the Stage minus the ball's radius. Finally, the ball collides with the `LEFT` boundary when its `x` location is less than or equal to zero plus the ball's radius.

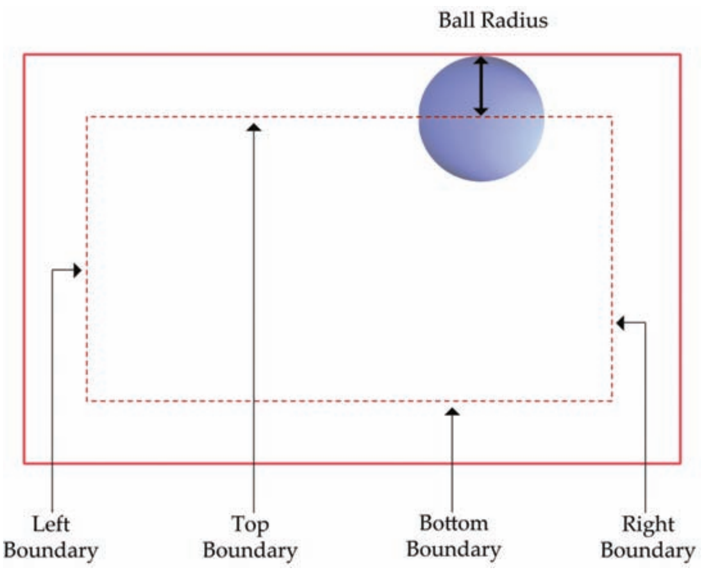


FIGURE 5-5 The `Ball` boundaries used for collision detection in the game of Pong.

Algorithm Design

The AS3 code for this application consists of the class constructor `pongApp()`, which initializes the game, and `moveBall()` and `movePaddle()`, which control the animation and drive the game. In addition, two auxiliary functions perform the tasks of detecting collisions. Table 5-5 lists all five main functions, and Figure 5-6 illustrates their relationship.

TABLE 5-5 Pong Application Program Functions

Function Name	Description
`pongApp()`	The class constructor. After launching, it initializes the `Ball` and `Paddle` properties and registers the events for interaction and animation.
`moveBall()`	Performs the task of animating the ball. This event handler controls the ball's movements by responding to the event `ENTER_FRAME` looping mechanism. It also guides the collision detection tasks by calling the appropriate functions.
`movePaddle()`	Responds to the event `MOUSE_MOVE`. This event handler performs the simple task of visually moving `Paddle`.
`checkBalltoWall()`	Checks whether `Ball` has collided with one of the walls and responds by reversing `Ball`'s velocity.
`checkBalltoPaddle()`	Checks whether `Ball` has been struck by `Paddle` and responds appropriately.

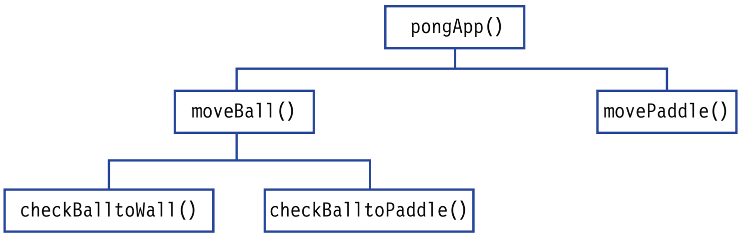


FIGURE 5-6 Relationship of the functions used by the Pong game.

The class `package` requires the `display` and `events` library classes. The last two lines contain closing curly brackets to conclude the `pongApp` class and the package.

```
package {
    import flash.display.*;
    import flash.events.*;

    public class pongApp extends MovieClip {

⋮

    }
}
```

Constants

The game of Pong uses constant values when identifying collisions and reversing the direction of the ball. These constants can be divided into three categories.

The first constant represents the radius of the ball, which is set to 16 pixels. This constant will be used to calculate the boundaries of the area where the ball can freely move.

The second category represents the concrete boundaries: `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`. As shown in Figure 5-5, the `TOP` boundary is not zero, but rather the point at which the ball strikes the ceiling, which is the radius of the ball. Lines 9–12 provide constant declarations for these boundaries.

The final constant is `REVERSE`, which is assigned a value of −1. When the ball's current velocity is multiplied by this constant, the velocity will be negated to reverse the direction.

```
6
7   const BALL_RADIUS:int=16;
8
9   const TOP:Number=BALL_RADIUS;
10  const BOTTOM:Number=stage.stageHeight;
11  const LEFT:Number=BALL_RADIUS;
12  const RIGHT:Number=stage.stageWidth - BALL_RADIUS;
13
14  const REVERSE:int=-1;
15
```

These six functions for this game are designed as follows.

Function `pongApp()` Design

The function `pongApp()` is immediately executed upon launching the game. Its objective is to simply initialize the elements of the game, which are organized into two tasks.

Task 1: This task performs the initialization of the two dynamic display objects, `Ball` and `Paddle`. Both use several inherited properties that will be set to initial values. In addition, `Ball` will have two newly constructed properties, `xVelocity` and `yVelocity`. Table 5-4 lists all of the properties.

Task 2: The second task of this function is to register two listener events. The first registered event listens for the simple movement of the mouse, which will be used to move the paddle. The second registered event is a frame loop that animates the ball by calling `moveBall()` at regular intervals.

```
function pongApp() {
   //TASK 1:  INITIALIZE BALL AND PADDLE
   Ball.height=RADIUS * 2;
   Ball.width=RADIUS * 2;
   Ball.x=40;
   Ball.y=40;
   Ball.xVelocity=3;
   Ball.yVelocity=3;
   Paddle.x=500;
   Paddle.y=500;

   //TASK 2: REGISTER EVENT LISTENERS TO MOVE BALL AND PADDLE
   addEventListener(Event.ENTER_FRAME, moveBall);
   stage.addEventListener(MouseEvent.MOUSE_MOVE, movePaddle);
}

```

Function `movePaddle()` Design

The function `movePaddle()` has one very simple objective—to move the paddle in a horizontal direction along the bottom of the Stage. The y position for `Paddle` has already been initialized. The `x` position will be altered as the user moves the mouse. Therefore, `Paddle` adheres to the position of the mouse along the *x*-axis.

```
function movePaddle(event:MouseEvent) {
   //TASK : MOVE PADDLE ALONG WITH THE MOUSE ON THE X-AXIS
   Paddle.x=stage.mouseX;
}

```

Function `moveBall()` Design

This function is the event handler for the `ENTER_FRAME` event registered by `pongApp()`. It is also the animation engine that takes the game from static mode to dynamic play by directing the movement of the ball (Task 1) and calling the functions that detect collisions at regular intervals.

In Task 1, `Ball` is incremented by its fixed velocity along the *x*- and *y*-axes. Thus, when the `xVelocity` property is positive, `Ball` moves from left to right; otherwise, it moves from right to left. When the `yVelocity` property is positive, the ball moves toward the bottom of the Stage.

Task 2 examines two types of collisions. The first collision occurs when the ball strikes a wall. The second collision takes place when the ball hits the paddle.

```
function moveBall(event:Event) {
   //TASK 1:  MOVE THE BALL ITS FIXED VELOCITY
   Ball.x += Ball.xVelocity;
```

```
    Ball.y += Ball.yVelocity;

    //TASK 2:  CHECK BALL COLLISIONS
    checkBalltoWall();
    checkBalltoPaddle();
}

```

Function checkBalltoWall() Design

This function checks for three possible boundary collisions. It is possible for a ball to collide with the top wall and the left wall at the same time, such as when the ball has struck the corner of the Stage. This rationale also applies to the top wall and the right wall. It is not possible, however, for a ball to strike the left and right walls simultaneously. The tasks for this function are to test these specific conditions.

If a ball moves beyond `TOP`, `LEFT`, or `RIGHT`, the ball is positioned at the boundary and the appropriate velocity is reversed. The tactic of positioning `Ball` at the boundary accomplishes two things:

- It tricks the eye into seeing the ball hit the boundary.
- It forces the ball into the correct location. Because the ball is not allowed to move beyond its boundary, its *x* and *y* coordinates are set to a corrected position.

Note that collision with the `BOTTOM` boundary is not being tested for. Once the ball has moved beyond the `BOTTOM` boundary of the Stage, the game is over because the ball is no longer available to hit. How would this be resolved? This issue is revisited in an end-of-chapter programming problem.

```
function checkBalltoWall() {
    //TASK 1:  IF BALL HAS HIT TOP OF THE STAGE, REVERSE ITS DIRECTION
    if (Ball.y<TOP) {
       Ball.y=TOP;
       Ball.yVelocity*=REVERSE;
    }

    //TASK 2:  IF BALL HAS HIT A SIDE WALL,  REVERSE ITS DIRECTION
    if (Ball.x<LEFT) {
       Ball.x=LEFT;
       Ball.xVelocity*=REVERSE;
    } else if (Ball.x > RIGHT) {
       Ball.x=RIGHT;
       Ball.xVelocity*=REVERSE;
    }
}

```

Function `checkBalltoPaddle()` Design

This function uses the display object inherited method called `hitTestObject()`. This method returns true if the referenced object collides with the argument object—in this case, `Paddle` and `Ball`.

```
function checkBalltoPaddle() {
   //TASK :  USE hitTestObject
   if (Paddle.hitTestObject(Ball)) {
      Ball.yVelocity*=REVERSE;
      }
   }
}
```

5.6 Case Study 2: Weight Loss Calculator with Error Detection

Problem

One pound of body weight is equivalent to 3500 calories, regardless of the person's gender or age. Thus, to lose one pound of weight, a person must create a deficit of 3500 calories. This can be done by burning more calories, by reducing the calorie intake, or by implementing a combination of both. The weight loss calculator created in this case study computes the time it will take to drop a given number of pounds using the method of reducing the daily calorie intake by a specific amount. An example run of this application is shown in Figure 5-7.

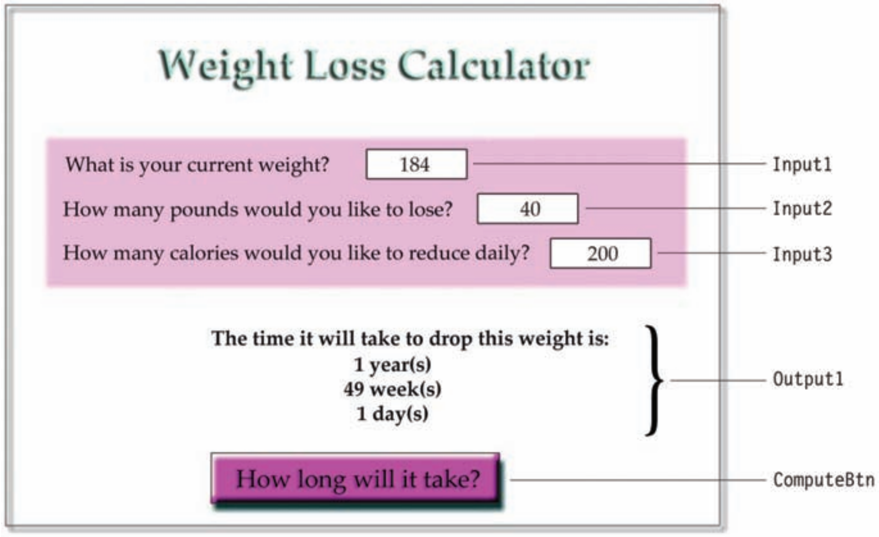

Weight Loss Calculator
What is your current weight?
184
How many pounds would you like to lose?
40
How many calories would you like to reduce daily?
200
The time it will take to drop this weight is:
1 year(s)
49 week(s)
1 day(s)
How long will it take?
Input1
Input2
Input3
Output1
ComputeBtn

FIGURE 5-7 Weight loss calculator application.

Problem Analysis

This interactive application provides input text fields for users to supply their current weight, the number of pounds they wish to drop, and the number of calories they will eliminate daily. Once it is determined that the input is valid, the application computes the amount of time it will take to lose the desired weight. This time will be displayed in terms of years, weeks, and days, as opposed to simply days. Examine the following two sentences. The first sentence is easier to assimilate.

Output Option 1: It will take 1 year, 49 weeks and 1 day to lose the weight.

Output Option 2: It will take 709 days to lose the weight.

This case study has two primary objectives:

1. Explore error detection in the user's input by using an `if–else if–else` statement.
2. Examine the use of the `appendText` method. Because the output for this application requires years, weeks, and days to obtain readable results, we will use this method to concatenate several text values.

Visual Design

The visual side of this application requires only one screen for the combined input of weight loss information and the output of the computed results. This interface screen uses a single frame in the Timeline.

In terms of design, it is important that the screen be readable, intuitive, visually appealing, and efficient. The text for this application is the primary source of information. To minimize the possibility of human error, the labels must be worded carefully so that they are easy to understand. The screen is organized from left to right and from top to bottom, and the input text fields are grouped together for effective and quick navigation. Finally, the single interactive button on the screen is labeled and graphically designed so that the user can reasonably understand how to use it.

The visual objects for this application consist of three input text boxes, a single dynamic text box for multipurpose output, and a button that initiates the process to calculate how long it will take to drop the given weight. Figure 5-7 shows the visual blueprint of the completed screen, along with the interactive objects. Table 5-6 lists the objects and describes their specific use.

TABLE 5-6 Visual Objects Used in the Input/Output Screen

Visual Object Name	Type of Object	Description of the Object
`Input1`	Input text box	Used to input the user's current weight.
`Input2`	Input text box	Used to input the number of pounds the user wishes to lose.
`Input3`	Input text box	Used to input the reduction in daily calories.
`Output1`	Dynamic text box	Used to display all errors that have been encountered during input and the amount of time it will take to lose the weight.
`ComputeBtn`	Button	Used to start the computation process. This interactive button is labeled "How long will it take?"

Program Design

The AS3 code for this application consists of three algorithms, each represented by a function. The class constructor is named `caloriesApp()`. The other two functions are `validateInput()` and `computeTime()`. Table 5-7 lists the functions used in the application and Figure 5-8 shows the relationship and flow between these functions.

TABLE 5-7 Weight Loss Program Functions

Function Name	Description
`caloriesApp()`	Controls interactivity. This function is the main function as well as the class constructor.
`validateInput()`	Identifies input errors. This function reads the input values and tests for a variety of errors.
`computeTime()`	Computes the time required to lose the weight. The time is then formatted and displayed on the screen.

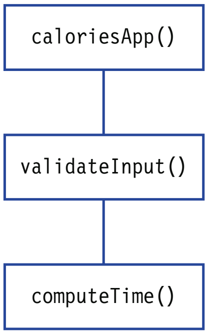


FIGURE 5-8 Relationship of functions used by the weight loss application.

The package for this program contains `flash.display.*` and `flash.events.*`. The application requires three constants for computing the amount of time for the weight loss. These constants (lines 7–9) define the number of days in a year, the number of days in a week, and the number of calories that constitute one pound.

```
1   package {
2      import flash.display.*;
3      import flash.events.*;
4
5      public class caloriesApp extends MovieClip {
6         //CONSTANTS USED FOR WEIGHT LOSS COMPUTATION
7         const DAYS_IN_YEAR:int=365;
8         const DAYS_IN_WEEK:int=7;
9         const CALORIES_PER_LB:int=3500;
10

⋮

60     }
61  }
```

The tasks of each function written for this application are outlined next.

Function `caloriesApp()` Design

The main algorithm, `caloriesApp`, is simply the class constructor function that will load immediately once the application is launched. The objective of this function is to construct a listener event that waits for the user to click `ComputeBtn`. Once this button is clicked, the sub-algorithm `validateInput()` is called. Prior to clicking the `ComputeBtn`, the user should have entered the current weight, the number of pounds the user wishes to lose, and the intended reduction in calories.

```
11  function caloriesApp() {
12     //TASK: REGISTER A MOUSE CLICK EVENT TO VALIDATE INPUT AND COMPUTE
13     ComputeBtn.addEventListener(MouseEvent.CLICK, validateInput);
14  }
15
```

Function `validateInput()` Design

The main goal of the `validateInput()` function is to validate the user's input. Once it is determined that the input does not contain any errors, the function `computeTime()` is called to process the weight loss information.

If an error is encountered in an input text box, an appropriate error message is displayed in the text field `output1`. When validating user input, the `isNaN` operator allows the application to test whether a variable or expression is **N**ot **a** **N**umber. For example, if the text box named `input1`, which gathers the user's current weight, contains something other than a number, `isNaN()` returns true.

Six obvious validation conditions are examined by the code, each one defined as a Boolean expression in an `if` statement:

Validation 1: The current weight must be a number.

Validation 2: The current weight must be greater than zero.

Validation 3: The number of pounds the user wishes to lose must be a number.

Validation 4: The number of pounds must be greater than zero and less than the current weight.

Validation 5: The reduced calories must be a number.

Validation 6: The reduced calories must be greater than zero.

If no errors are found in the input, the `else` default clause calls the function `computeTime()` to compute the amount of time it will take to lose the desired weight.

```
function validateInput(event:MouseEvent) {
   //TASK1: EXAMINE EVERY POSSIBLE INCORRECT INPUT FROM THE USER
   if (isNaN(Number(input1.text))) {
      output1.text="Your current weight must be a number.\n";
   } else if (Number(input1.text) <= 0) {
      output1.text="Your current weight must be greater than zero.\n";
   } else if (isNaN(Number(input2.text))) {
      output1.text="The amount of weight you wish to lose must be a number.\n";
   } else if (Number(input2.text) >= Number(input1.text) || Number(input2.text) <= 0) {
      output1.text="The pounds to lose must be less than your weight.\n" +
         "and greater than zero.";
   } else if (isNaN(Number(input3.text))) {
      output1.text="The reduced calories must be a number.\n";
   } else if (Number(input3.text)<= 0) {
      output1.text="The reduced calories must be greater than zero.\n";
   } else {
   //TASK: IF INPUT FROM THE USER IS CORRECT, COMPUTE THE TIME TO LOSE WEIGHT
      computeTime();
   }
}

```

Function `computeTime()` Design

The `computeTime()` function is called by `validateInput()` only after it has been established that the input is error free. This function, whose objective is to calculate the time it will take the user to lose the specified weight and display it in a readable fashion, performs four tasks:

Task 1: Read the text from dynamic text boxes for `Input1`, `Input2`, and `Input3` into the data objects `currentWeight`, `weightLoss`, and `reducedCals`.

Task 2: Compute the time it takes to lose the weight. The time will be in days.

Task 3: For easy readability, convert the total number of days into years, weeks, and days. The `%` operator is used in these computations.

Task 4: Construct the output display of the time it takes to lose the weight. The `output1` text box is used to display this output and a general message along with the six values in the remaining dynamic text boxes.

```
function computeTime () {
   //TASK 1:  READ VALUES FROM INPUT TEXTFIELDS
   var currentWeight:Number=Number(input1.text);
   var weightLoss:Number=Number(input2.text);
   var reducedCals:Number=Number(input3.text);

   //TASK 2:  COMPUTE THE DAYS REQUIRED TO LOSE THE WEIGHT
   var days:int=int((weightLoss * CALORIES_PER_LB)/reducedCals);

   //TASK 3:  TRANSLATE DAYS INTO YEARS, WEEKS, AND DAYS
   var years:int=days/DAYS_IN_YEAR;
   var weeks:int=days%DAYS_IN_YEAR/DAYS_IN_WEEK;
   days=days%DAYS_IN_YEAR%DAYS_IN_WEEK;

   //TASK 4:  DISPLAY YEARS, WEEKS, AND DAYS TO LOSE WEIGHT
   output1.text="The time it will take to drop this weight is:\n";
   if (years>0) {
      output1.appendText(years + " year(s)\n");
   }
   if (weeks>0) {
      output1.appendText(weeks + " weeks(s)\n");
   }
   if (days>0) {
      output1.appendText(days + " day(s)\n");
   }
}
```

5.7 The Nested `if` Statement

The previous section explored `if` statements that may also contain `if-else` and `else` clauses. These clauses provide the option of selecting multiple alternatives. An `if` statement contains a set of action statements, enclosed in `{}`, which may also be another `if` statement—hence the term `if` statement. The objective of a nested `if` statement is to improve the efficiency of the code by reducing or eliminating redundant Boolean tests. These refinements occur when compound Boolean expressions are pared down and assembled into a collection of simple Boolean expressions containing nested `if` statements. Oftentimes, use of such nested `if` statements can also boost readability.

The next two examples demonstrate nested `if` statements.

Example 1

For this first example, let us consider the childhood game called Rock, Scissors, Paper. In this two-player game, the user plays against the computer. The user selects one of the elements, while the computer is randomly assigned one. The user wins only if her choice dominates the computer's choice. The two players tie if the choices are the same. Otherwise, the user loses. Here are the domination rules:

Rule 1: Rock beats scissors because it can crush it.

Rule 2: Scissors beats paper because it can cut it.

Rule 3: Paper beats rock because it can cover it.

A good solution will display a detailed reason for a win, loss, or tie outcome for the user. There are a total of nine detailed game outcomes.

Two code solutions are provided here to showcase the efficiency of a nested `if` statement.

Rock, Scissors, Paper Game Solution 1

This first solution does not rely on nested statements. Upon close examination, it becomes clear that within each compound Boolean expression is a simple Boolean expression that appears in several compound Boolean expressions. For example, the simple Boolean expression user == ROCK occurs three times, on lines 1, 3, and 5.

```
1  if (user == ROCK && computer == PAPER) {
2     trace ("You lose because computer chose PAPER, which covers
ROCK.");
3  } else if (user == ROCK && computer == SCISSORS) {
4     trace ("ROCK wins because SCISSORS can be crushed by ROCK.  Player
wins.");
5  } else if (user == ROCK && computer == ROCK) {
6     trace ("You tied with the computer because you both chose ROCK. ");
7  } else if (user == PAPER && computer == SCISSORS) {
8     trace ("PAPER loses because SCISSORS cuts PAPER.  Player loses.");
9  } else if (user == PAPER && computer == ROCK) {
10    trace ("PAPER wins because PAPER covers ROCK.  Player wins.");
11 } else if (user == PAPER && computer == PAPER) {
12    trace ("You both chose PAPER.  Player ties with the computer.");
13 } else if (user == SCISSORS && computer == PAPER) {
14    trace ("SCISSORS wins because PAPER is vulnerable to cutting.  You
win.");
} else if (user == SCISSORS && computer == ROCK) {
   trace ("ROCK loses because PAPER covers ROCK.  Player loses.");
} else {
   trace ("You both chose SCISSORS.  Player ties with the computer.");
}
```

Rock, Scissors, Paper Game Solution 2

This second solution utilizes nested `if` statements to take advantage of the fact that the compound Boolean expressions from Solution 1 can be grouped into three main scenarios:

```
user == ROCK
user == PAPER
user == SCISSORS
```

Within each scenario is a set of subscenarios expressed as nested `if` statements. For example, when the user chooses Rock, as identified in line 1, there are three possible subscenarios identified on lines 2–8: `computer == PAPER`, `computer == SCISSORS`, and `computer == ROCK`.

The final solution is easier to read and of higher quality due to the elimination of redundant Boolean test expressions.

```
1  if (user == ROCK){ //SCENARIO 1: USER IS ROCK
2     if (computer == PAPER) {
3        trace ("You lose because the computer chose PAPER, which covers
ROCK.");
4     } else if (computer == SCISSORS) {
5        trace ("ROCK wins because SCISSORS can be crushed by ROCK.
Player wins.");
6     } else {
7        trace ("You tied with the computer because you both chose ROCK.
");
8     }
9  } else if (user == PAPER ) {   //SCENARIO 2: USER IS PAPER
10    if (computer == SCISSORS) {
11       trace ("PAPER loses because SCISSORS cuts PAPER.  Player los-
es.");
12    } else if (computer == ROCK) {
13       trace ("PAPER wins because PAPER covers ROCK.  Player wins.");
14    } else {
15       trace ("You both chose PAPER.  Player ties with the computer.");
16    }
17 } else if (user == SCISSORS) { //SCENARIO 3: USER IS SCISSORS
18    if (computer == PAPER) {
19       trace ("SCISSORS wins because PAPER is vulnerable to cutting.
Player wins.");
```

```
   } else if (computer == ROCK) {
      trace ("ROCK loses because PAPER covers ROCK.  Player loses.");
   } else {
      trace ("You both chose SCISSORS.  Player ties with the comput-
er.");
   }
}
```

Example 2

In this example, we examine a segment of AS3 code that issues facts about a user's age. Depending on the user's age, multiple conditions need to be examined and one or more age-appropriate pieces of information displayed based on the following facts.

Fact 1: Any person age 5 or younger is considered to be a child.

Fact 2: All persons older than age 5 and younger than age 13 are considered to be kids.

Fact 3: Kids younger than age 8 go to elementary school.

Fact 4: Kids age 8 and older go to middle school.

Fact 5: All persons from the age of 13 through 19 are called teenagers.

Fact 6: Teenagers younger than age 16 cannot drive a car.

Fact 7: Teenagers of age 16 and older may drive a car if they pass a driving test.

Fact 8: Any person who is not a child, kid, or teenager is considered to be an adult.

Two solutions are used to showcase the efficiency of a nested `if` statement.

Age-Appropriate Facts Solution 1

The first solution does not use nested statements. Because multiple statements can be true, this solution is split into two independent `if` statements. Suppose the user is 13 years old. He is a teenager and he cannot drive.

```
//ARE YOU A CHILD, KID, TEENAGER, OR ADULT?
if (age <= 5) {
   trace ("You are a child.");
} else if (age < 13) {
   trace ("You are a kid.");
} else if (age <= 19) {
   trace ("You are a teenager.");
} else {
   trace ("You are an adult.");
}

//WHAT YOU CAN (OR CANNOT) DO AT YOUR CURRENT AGE
if (age < 13 && age > 5) {
   trace ("You are in school.");
} else if (age <= 19 && age >= 16) {
   trace ("You can drive.");
} else if (age < 16)
   trace ("You cannot drive yet.");
} else {
   trace ("You are an adult.");
   trace ("You can run for president.");
}
```

Age-Appropriate Facts Solution 2

This solution uses a single nested `if` statement in which user is categorized into one of the following groups:

Child Kid Teenager Adult

By using nested `if` statements, multiple conditions can be examined. This code solution is concise, efficient, and easy to read.

```
if (age <= 5) {                              // A CHILD
   trace ("You are a child.");
} else if (age < 13) {                       // A KID
   trace ("You are a kid.");
   if (age > 5) {
      trace ("You should be in school.");
   }
} else if (age <= 19) {                      // A TEENAGER
   trace ("You are a teenager.");
   if (age <= 16) {
      trace ("You can drive.");
   } else {
      trace ("You cannot drive yet.");
   }
} else {                                     // AN ADULT
   trace ("You are an adult.");
}
```

5.8 Case Study 3: The Virtual Pet Fish

Problem

The virtual fish in this case study has a single requirement for survival—food. Our fish will never starve, however, because there is a constant source of food in the tank.

The fish exhibits three possible states: playing, hungry, and eating. While the fish is playing, it is in a constant state of motion, moving around the tank seeking out its toy, and burning off calories. Once it becomes hungry, the fish immediately moves toward the food source. When it locates its food, the fish eats until its stomach is full, at which point it again seeks out its toy to burn off calories until it is hungry again, and the cycle continues.

There are three objectives for this case study:

1. Use nested `if` statements to monitor and respond to the three possible conditions of the fish: `isHungry`, `isEating`, and `isAtPlay`.
2. Build custom properties to identify individual fish attributes.
3. Work with basic game mathematics to create realistic motion.

Problem Analysis and Visual Design

The primary goal of this application is to create a display object that behaves like a simple-minded pet fish in a virtual fish tank. This pet fish will need a minimal amount of artificial intelligence that allows it to evaluate its current condition as it plays, becomes hungry, or generally swims around the tank. Once it understands its current condition, it then responds with an appropriate change in behavior.

The display object on the stage that represents the pet fish is a `MovieClip` instance named `Fish`. In addition to `Fish`, two other MovieClip instances are found in this virtual environment: `Toy` and `Food`. These three display objects are shown in the fish tank environment in Figure 5-9.

`Fish`'s behavior will be controlled by basic artificial intelligence constructed as a set of nested `if–else if–else` statements, more appropriately called rules. These rules will utilize a set of properties, shown in Table 5-8, created specifically for this purpose.

The artificial intelligence rules for controlling `Fish` are as follows:

Rule 1: If `Fish` is playing with its toy (`isAtPlay == true`), rotate and move `Fish` toward `Toy`. Burn a calorie by reducing the amount of food in `Fish`'s stomach and check whether it is hungry.

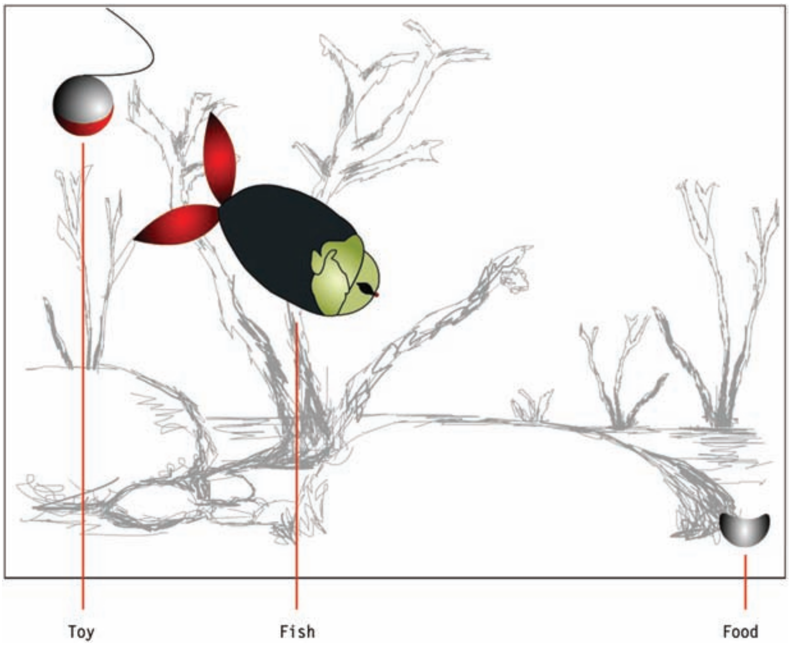


FIGURE 5-9 Virtual pet fish application.

Rule 2: If `Fish` is hungry (`isHungry == true`), rotate and move `Fish` toward `Food`. Check whether it has reached its food.

Rule 3: If `Fish` is eating (`isEating == true`), check whether its stomach has reached its limited capacity for food.

TABLE 5-8 Properties Constructed for `Fish`

`Fish` Property	Type of Property	Description of Property
`isHungry`	Boolean, dynamic value	True if `Fish` is hungry; false otherwise. If `Fish` is hungry, it is not eating or playing.
`isAtPlay`	Boolean, dynamic value	True if `Fish` is playing with, or seeking to play with, its toy.
`isEating`	Boolean, dynamic value	True if `Fish` is currently eating; false otherwise. If `Fish` is eating, it is not hungry and not playing.
`velocity`	Number, static value	`Fish`'s normal traveling velocity. This static value will not change throughout the program.
`capacity`	Number, static value	The amount of food `Fish` can eat before it is full.
`inStomach`	Number, static value	The current amount of food in `Fish`'s stomach.

Algorithmic Design

The program for this game consists of the five functions described in Table 5-9. Figure 5-10 shows the relationship between these functions.

TABLE 5-9 Program Functions Used by the Virtual Pet Fish Application

Function Name	Description
`virtualPetApp()`	The main function as well as the class constructor. It executes immediately when the application begins. This function is responsible for initializing the `Fish` properties and registering the main listener event `ENTER_FRAME`.
`makeItLive()`	The event handler for the listener event `ENTER_FRAME`. It is also the engine that drives the artificial intelligence of the pet fish by monitoring its three possible conditions—`isHungry`, `isAtPlay`, and `isEating`—and responds with a call to the appropriate function that carries out the required behavior.
`goPlay()`	Called when `Fish` is seeking out its toy and playing.
`findFood()`	Dictates how `Fish` will locate its food.
`eatFood()`	Called when `Fish` is eating its food. This function dictates how the pet eats and finishes eating.

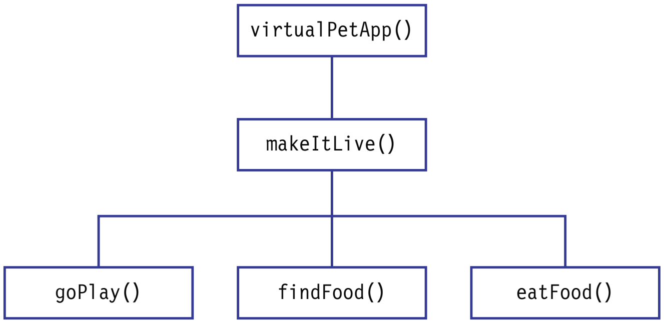


FIGURE 5-10 Relationship structure of functions used by the virtual pet fish application.

This program does not require any game constants. The package contains the libraries `flash.display.*` and `flash.events.*`.

```
package {
   import flash.display.*;
   import flash.events.*;

```

```
    public class virtualPetApp extends MovieClip {

⋮

    }
}
```

The tasks of each function written for this application are outlined next.

Function `virtualPetApp()` Design

The function `virtualPetApp()`, which is the class constructor, has the sole task of initializing `Fish` and registering the event listener `ENTER_FRAME` to enable a looping mechanism. Its instructions are organized into three tasks:

Task 1: Set `Fish`'s initial behavioral conditions. Initially `Fish` is playing with its toy. Because it is not currently hungry and not eating, both these conditions are set to false.

Task 2: Set `Fish`'s attributes that govern its velocity and the physical properties that will dictate hunger. Its normal traveling velocity is set to 10. Its capacity, which is the amount of food it can eat to satisfy its hunger, and the current amount of food in its stomach are both set to 50.

Task 3: Register the listener event `ENTER_FRAME` with the function `makeItLive()` as the event handler.

```
function virtualPetApp() {
    //TASK 1: SET THE INITIAL CONDITION OF THE FISH TO PLAY
    Fish.isHungry=false;
    Fish.isAtPlay=true;
    Fish.isEating=false;

    //TASK 2: SET THE INITIAL PROPERTIES OF THE PET FISH
    Fish.velocity=10;//ITS NORMAL TRAVELING VELOCITY.
    Fish.capacity=50;//HOW MUCH CAN IT EAT BEFORE IT'S FULL
    Fish.inStomach=50;//THE CURRENT AMOUNT OF FOOD IN FISH

    //TASK 3:  USE AN ENTER_FRAME LISTENER TO MAKE PET ALIVE
    addEventListener(Event.ENTER_FRAME,makeItLive);
}

```

Function `makeItLive()` Design

Function `makeItLive()` is the event handler that executes at regular intervals when called by the event listener for `ENTER_FRAME`. This function takes the static fish and makes it dynamic by responding with appropriate function calls based on the current condition of `Fish`.

By using an `if–else if–else if` structure, the function can identify when `Fish` is hungry, playing with its toy, or eating.

```
function makeItLive (event:Event) {
   //EXAMINE AND RESPOND THE CONDITIONS: isHungry, isAtPlay, isEating
   if (Fish.isAtPlay) {
      goPlay();
   } else if (Fish.isHungry) {
      findFood();
   } else if (Fish.isEating) {
      eatFood();
}

```

Function `goPlay()` Design

The `goPlay()` function directs the behavior of `Fish` while its current state is playing with its toy. This function is subdivided into five tasks:

Task 1: Burn a calorie by reducing the amount of food in the fish's stomach by decrementing this amount by one.

Task 2: Compute the distance `Fish` must travel to get to its toy. This distance must be computed along both the *x*-axis and the *y*-axis.

Task 3: Turn `Fish` in the direction of its toy. To accomplish this task, the angle must first be computed. This is done by using the arctangent function `Math.atan()`, which produces a result in radians. Because rotation requires a value in degrees, the conversion from radians to degrees is performed by multiplying the angle by `180 / Math.PI`.

Task 4: Move `Fish` toward `Toy`. To create the illusion of elegant dynamic movement, the fish must be able to gradually slow down as it nears its toy, as opposed to barreling into the toy and stopping abruptly. This can easily be done by dividing the distance to be traveled by the fixed velocity, which eases `Fish` to its final destination.

Task 5: Check whether `Fish` is hungry again. This code assumes `Fish` is hungry if it has less than 25 units of food in its stomach. If `Fish` is found to be hungry, the `isAtPlay` property is set to false and the `isHungry` property is set to true.

```
function goPlay() {
   //TASK 1:  BURN A CALORIE
   Fish.inStomach–;

   //TASK 2: COMPUTE THE DISTANCE TO TOY
   var xDistance:Number=Toy.x-Fish.x;
   var yDistance:Number=Toy.y-Fish.y;

```

```
    //TASK3: ANGLE THE FISH TOWARD ITS TOY
    var Angle:Number=Math.atan2(yDistance,xDistance);
    Fish.rotation=Angle*180/Math.PI;

    //TASK4: MOVE FISH CLOSER TO ITS TOY
    Fish.x+=xDistance/Fish.velocity;
    Fish.y+=yDistance/Fish.velocity;

    //TASK 5: CHECK IF THE FISH IS HUNGRY
    if (Fish.inStomach < 25) {
       //SUBTASK 1: SET THE FISH TO HUNGRY
       Fish.isAtPlay=false;
       Fish.isHungry=true;
    }
}

```

Function `findFood()` Design

The `findFood()` function is divided into four tasks:

Task 1: Compute the distance `Fish` must travel to locate `Food`.

Task 2: Turn `Fish` in the direction of `Food`.

Task 3: Move `Fish` toward `Food` by traveling distance divided by velocity. `Fish` will ease toward its final destination.

Task 4: Check whether `Fish` has located `Food`. This code assumes `Fish` has located `Food` once its horizontal distance is less than 5 and its vertical distance is less than 3. At this point, the `isEating` property is set to true and the `isHungry` property is set to false.

```
function findFood() {
    //TASK 1: CALCULATE DISTANCE TO ITS FOOD
    var xDistance:Number=Food.x-Fish.x;
    var yDistance:Number=Food.y-Fish.y;

    //TASK 2: TURN FISH TOWARD ITS FOOD
    var Angle:Number=Math.atan2(yDistance,xDistance);
    Fish.rotation=Angle*180/Math.PI;

    //TASK 3: MOVE THE FISH TOWARD ITS FOOD
    Fish.x+=xDistance/Fish.velocity;
    Fish.y+=yDistance/Fish.velocity;

    //TASK 4: CHECK IF THE FISH HAS LOCATED ITS FOOD
    if (xDistance < 5 && yDistance < 3) {
       //SUBTASK: SET THE STATE OF THE FISH TO EATING
       Fish.isHungry=false;
```

```
        Fish.isEating=true;
    }
}

```

function eatFood() Design

The eatFood() function is divided into two tasks:

Task 1: Increment the amount of food in Fish's stomach.

Task 2: Check whether Fish's stomach is full. This code assumes Fish is full when the amount of food in its stomach has reached capacity. At this point, the isEating property is set to false and the isAtPlay property is set to true.

```
function eatFood() {
    //TASK 1:  FISH CONSUMES A CALORIE OF FOOD
    Fish.inStomach++;

    //TASK 2: CHECK IF THE FISH IS FULL
    if (Fish.inStomach >= Fish.capacity) {
        //SUBTASK: SET THE STATE OF THE FISH TO PLAYING
        Fish.isEating=false;
        Fish.isAtPlay=true;
    }
}

```

5.9 The switch Statement

In addition to the if statement, there is another selective control structure available—the switch statement. The switch statement is not strictly necessary, but can in some cases make for more concise and readable code.

As seen in the previous sections, one of the most commonly seen patterns in programming is a series of if–else if–else statements that test a single value against a series of values. For example, the following code segment displays the string "Freshman," "Sophomore," "Junior," or "Senior," on depending on the value of year:

```
if (year == 1) {
   trace("Freshman");
}else  if (year == 2 {
   trace("Sophomore");
}else if (year == 3) {
   trace("Junior");
}else {
   trace("Senior");
}
```

This structure is important in programming because it provides us with a mechanism to solve all multiway selective-type problems. Because multiway selection statements can sometimes be difficult to follow, many languages provide an alternative method of handling this concept—the `switch` statement.

In AS3 3.0, `switch` statements are often used when several options depend on the value of a single variable or expression, as in the previous example. The typical form of the `switch` statement is shown next. The words `switch`, `case`, `break`, and `default` are reserved keywords.

```
switch (variable) {

case  value1 :        statement list1;
  break;

case  value2 :        statement list2;
  break;
.
.
.
case  valueN :        statement listN;
  break;


default : statement listDefault;
}
```

1. The value of `variable` is determined.
2. The first matching value with a `case` is found. The statements following the matching case are executed.

   Note: `break` and `default` statements are optional. If a `break` statement occurs, control is transferred to the first statement following the end of the `switch` statement; otherwise, the execution of statements continues. In general, a `case` should end with a `break` statement.
3. If no matching value is found, then the `default` statement list is executed.

The following code segment is a rewrite of the `if` statement that displays "Freshman," "Sophomore," "Junior," or "Senior," depending on the value of `year`. This segment illustrates the `switch` statement. Each of the statement lists in a `switch` statement usually ends with a `break` statement. The effect of the `break` in these statements causes a transfer of control to the end of the `switch` statement.

```
switch  (year) {
   case   1:
      trace("Freshman");
      break;
   case   2:
      trace("Sophomore");
      break;
   case   3:
      trace("Junior");
      break;
   default  :
      trace("Senior");
}
```

The next three examples illustrate how the `switch` statement works.

Example 1

```
1   var Num:Number;
2   switch (Num){
3      case 0 : trace ("ZERO");
4         break;
5      case 1 : trace ("ONE or");
6      case 2 : trace ("TWO");
7         break;
8      default :trace("NO
9   MATCHES!");
10  }
11  trace ("DONE");
```

Evaluations for Different Values of `Num`

a. `Num = 0`

In this scenario, the value of `Num` is matched with the first `case` in line 1. The `trace` statement executes, and a `break` is encountered on line 4, ending the `switch` statement. Line 11 is an unconditional statement that displays the text "DONE."

The output produced by this segment of code is

```
ZERO
DONE
```

b. `Num = 1`

The value of `Num` has found a match with the second `case` statement, 1, on line 5. The `trace` statement on the same line executes, but no `break` is encountered; hence, all instructions are executed until a `break` is found, or the `switch` statement terminates on its own. In this case, a `break` statement is found on line 7.

The output produced by this segment of code is

```
ONE or
TWO
DONE
```

c. `Num = 4`

No matching values for `Num` are found. The `default` statement on line 8 is executed.

The output produced by this segment of code is

```
NO MATCHES!
DONE
```

Example 2

```
1   var Digit:int;
2   switch (Digit){
3      case 0 : trace ("ZERO");
4         break;
5      case 1 :
6      case 3 :
7      case 5 :
8      case 9 : trace ("ODD");
9         break;
10     case 2 :
11     case 4 :
12     case 6 :
13     case 8 : trace ("EVEN");
14  }
```

Evaluation

The `switch` statement in this segment of code illustrates the capture of multiple matches.

a. `Digit = 3`

When `Digit` contains the value 3, a match is found in the third `case` statement on line 6. There are no statements to execute, but more importantly, there is no `break` statement to terminate the `switch` statement. This means any instructions from lines 6 through 8 are executed until a `break` is encountered on line 9.

The output produced by this segment of code is

```
ODD
```

b. `Digit = 8`

In this scenario, a match is found in the final `case` statement on line 13. There is no `break` statement following this final statement list because there are no statements to execute, so the `switch` statement will terminate automatically. As shown in this example, `default` statements are optional.

The output produced by this segment of code is

```
EVEN
```

Example 3

```
1  var P:int;
2  var Q:int;
3  switch (P){
4     case 0 : switch (Q){
5        case 3: P = P + Q; break;
6        default: P = P - Q;
7        }
8        break;
9     case 1 :
10    case 3:
11    case 5 : switch (Q){
12       case 2:
13       case 6: P = P * Q; break;
14       case -4:
15       case -6: P = Q * Q;
16       }
17 }
18 trace ("P = ", P );
19 trace ("Q = ", Q );
```

Evaluation

This segment of code illustrates a nested `switch` statement. Like `if` statements, `switch` statements may contain multiple layers of alternatives.

a. `P = 0, Q = 3`

A match is found for `P` in the first `case` statement on line 4. The nested `switch` statement `switch (Q)` is executed, and a match is found in case 3 on line 5. The statement `P = P + Q` is executed, followed by the `break` on line 8.

The output produced by this segment of code is

```
P = 3
Q = 3
```

b. `P = 1, Q = -3`

When `P` contains the value 1, a match is found in the second `case` statement on line 9. There is no `break` statement to terminate the `switch` statement, so all statements will be executed until a `break` is encountered. In this case, the nested `switch (Q)` statement in case 5 on line 11 will execute. No match is found within the nested `switch` statement, and the values in `P` and `Q` will not be altered.

The output produced by this segment of code is

```
P = 1
Q = -3
```

5.10 Case Study 4: Airship Flight Simulator

Problem

In this case study, we construct a simple top-view flight simulator for an airship. Airships were once luxury passenger aircrafts that experienced a period of grandeur in Europe during the 1920s and 1930s. These magnificent aircraft, which were designed to fly wealthy passengers across oceans and continents, often featured opulent lounges, fine accommodations, dining rooms, and even smoking rooms. The largest passenger airship in history, as well as the most disastrous, was the *Hindenburg*.

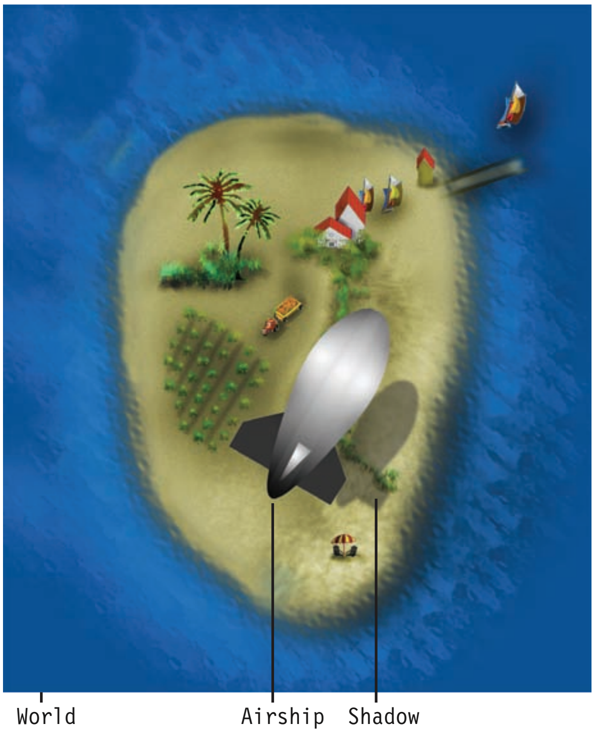


FIGURE 5-11 Airship flight simulator application.

Three objectives are addressed in this case study:

1. Use `switch` statements to respond to keyboard control of the airship.
2. Use a timer to create smooth animations as the airship flies to a higher or lower altitude over the world below.
3. Work with basic game math to reposition the world below as the airship changes direction and flies overhead.

Game Plan and Analysis

The flight simulator for this case study, shown in Figure 5-11, will be kept simple in terms of the rules of engagement and the game play parameters. For example, the airship will not be permitted to land or crash. Four essential characteristics govern how this simulator will perform:

1. The airship will be constrained to a static location at Stage center. The world below will move as the airship flies overhead. There are two reasons for the restricted movement of the airship. First, an anchored airship will not run the risk of flying off the Stage. It will be possible for the airship to change direction. Second, and most important, the user can expect a uniform perspective, thereby creating the illusion of being aboard, or at least linked to, the airship. This consistent perspective will make the navigation more user friendly.
2. The dynamic world below the airship will be stored as a separate `MovieClip` instance. Using an `ENTER_FRAME` event, this world will be continuously repositioned and scaled to reveal a new view from the moving airship.
3. The airship will have a fixed velocity. The user will not have access to this velocity.
4. The user can guide the airship by simply pressing the up, down, left, or right arrow keys. These keys will change the direction and elevation of the airship. The up arrow will be used to climb to a higher altitude. The down arrow key will be used to decrease elevation, but not beyond a set minimum altitude. The left and right arrow keys allow the airship to change direction by turning left or right a set number of degrees.

Visual Design

The flight simulator requires three `MovieClip` instances as shown in Figure 5-11: `Airship`, `Shadow`, and `World`. Both `Airship` and `Shadow` are simple vector drawings created in Flash, with `Shadow` representing the shadow the airship casts over the world below. For this example, the `MovieClip` instance named `World` was created in Adobe

Photoshop and then imported into Flash. This is not a requirement and could just as easily have been done in Flash using the vector drawing tools.

The simulator application relies solely on a single frame, representing a single screen. There are no instructions or buttons. The Timeline is organized using three layers. The bottom layer holds `World`, the middle layer holds `Shadow`, and the top layer contains `Airship`. This animation is set to a rate of 60 frames per second, which is fast enough to create the illusion of a smooth flight—a hallmark of these luxurious airships.

Program Design

The algorithm for this game application will use `Airship`, `Shadow`, and `World` to create the animations necessary for visualizing an aircraft in virtual flight. These animations will rely on a select set of inherited display object properties. In addition, the `MovieClip` instance `Airship` will utilize the constructed properties `Velocity` and `Altitude`. Table 5-10 provides a comprehensive explanatory list of all the display object properties required for the visualization of the flight simulator.

TABLE 5-10 Properties Constructed for the Airship Flight Simulator Application

Display Object Name	Property
`Airship`	`rotation`: `Airship`'s turn amount dictated by the left and right arrow keys. Note: The `x` and `y` position of `Airship` will remain static throughout execution; however, the ability to turn left or right will be permitted.
	`Velocity`: `Airship`'s traveling `Velocity`. This newly constructed property will be used to compute the visual adjustments made to the `World`. These adjustments will ultimately create the illusion of movement from the perspective of the airship looking below as it travels.
	`Altitude`: Elevation gains and losses of `Airship`. This property is used to make alterations to the velocity for the airship. The lower the airship, the faster it appears to fly because the `World` is magnified.
`Shadow`	`alpha`: The level of transparency for the airship's shadow. Creates a three-dimensional impression of an overhead airship.
	`scaleX`, `scaleY`: The scale of the shadow projected by the airship. Used to create the illusion of three-dimensionality. This value is altered only when the elevation of the airship changes. As the airship descends closer to the surface, the shadow's scale increases. As the airship moves farther away, the shadow's scale decreases.
`World`	`x`, `y`: `World` is the only display object whose `x` and `y` positions will be altered during runtime.
	`scaleX`, `scaleY`: As with `Shadow`, the scale of `World` will be altered when the elevation of `Airship` changes.

This flight simulator application will consist of five functions: `airshipApp()`, `adjustWorld()`, `checkControlKey()`, `zoomHigher()`, and `zoomLower()` (described in Table 5-11). Figure 5-12 shows the relationship between these functions.

TABLE 5-11 Airship Flight Simulator Program Functions

Function Name	Description
`airshipApp()`	Sets the game elements and adds event listeners. The main function of the application, it is also the constructor function and executes immediately when the application is launched.
`adjustWorld()`	Adjusts `World`'s location as the airship moves. This function is an event handler that is triggered by the `ENTER_FRAME` event. It executes at regular intervals.
`checkControlKey()`	Examines the control keys (up, down, left, and right), and responds to them.
`zoomHigher()`	Creates a visual animation that uses scaling to impart the illusion of the airship climbing in altitude. This function is an event handler triggered by a timer event.
`zoomLower()`	Creates a visual animation that shows the airship decreasing in altitude. Scaling is used to trick the eye. Like `ZoomHigher()`, this function is an event handler triggered by a timer event.

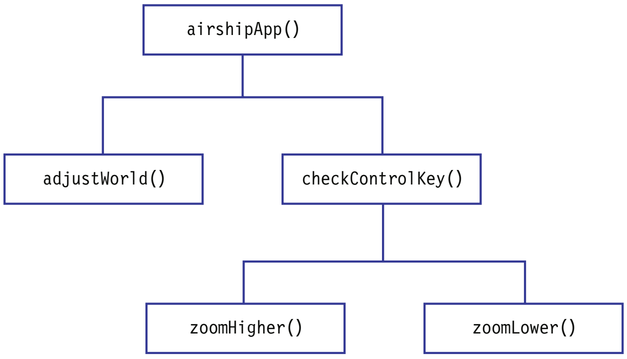


FIGURE 5-12 Relationship of functions used in the airship flight simulator application.

The first line of this `airshipApp.as` AS3 file consists of the class `package`. The required library classes to be imported are `display`, `events`, and `utils`. The `utils` library classes are needed for the timer event. The final curly brackets, located on the last two lines of the file, conclude the `airshipApp` class and the package.

```
1   package {
2      import flash.display.*;
3      import flash.events.*;
4      import flash.utils.*;
5      public class airshipApp extends MovieClip {

⋮

103    }
104 }
```

Game Constants

Two categories of constant values are needed. The first category defines game constants. The first constant is `RADIANS`, which will be used in the game mathematics. Recall that Flash trigonometric functions use radians instead of degrees as units. The other constants include the airship's cruising altitude and minimum and maximum flying altitudes. Line 12 contains the constant `ALTITUDE_AMT`, which specifies the altitude increase and decrease amounts when the user presses the up and down arrow control keys. The constant `TURN` specifies the turn amount in degrees for maneuvering left or right when the user presses the left and right arrow keys.

```
6          //GAME CONSTANTS
7          const RADIANS:Number=Math.PI/180;
8          const MIN_ALTITUDE:Number=30;
9          const CRUISING_ALTITUDE:Number=50;
10         const MAX_ALTITUDE:Number=70;
11         const CRUISING_VELOCITY:Number=.3;
12         const ALTITUDE_AMT:Number=0.1;
13         const TURN:Number=4;
```

The last category of constants defines the flight control keys. These constants represent the specific ASCII key values used to direct the airship right and left and to adjust its altitude.

```
14         //FLIGHT CONTROL CONSTANTS
15         const RIGHTARROW:Number=39;
16         const LEFTARROW:Number=37;
17         const UPARROW:Number=38;
18         const DOWNARROW:Number=40;
```

Timer Variable

In addition to the constants, there will be a global variable, `atimer`, that stores a timer object. This variable is used by the functions `checkControlKey()`, `zoomHigher()`, and `zoomLower()` to smooth the scaling animation of `World` as the airship gains or losses altitude. This variable object is made global so that all three functions that use it will have unfettered access to the variable. Chapter 8 discusses the alternative to creating global variables such as `atimer`.

```
//TIMER VARIABLES
var atimer:Timer;
```

The functions for this application are designed as follows.

Function `airshipApp()` Design

The main algorithm, `airshipApp()`, is a constructor that loads immediately once the application is launched. The objective of this function is to set the initial properties of the elements on the Stage and add event listeners for the control keys and adjustments made to the `World` at every frame loop.

Task 1: Initialize the properties of `Airship`. The two properties `Velocity` and `Altitude` are constructed when they are initialized.

Task 2: Initialize the properties of `Shadow`. The initial scale of `Shadow` is set to 25% of the size of the airship and 25% opacity.

Task 3: Initialize the position and scale of `World`.

Task 4: Register the event listeners for the `ENTER_FRAME` event and the `KEY_DOWN` event. The keyboard event listener waits for the user to interactively control the altitude and direction of the airship.

```
function airshipApp() {
   //TASK 1:  SET AIRSHIP PROPERTIES TO INITIAL VALUES
   Airship.x=Airship.y=350;
   Airship.Altitude=CRUISING_ALTITUDE;
   Airship.Velocity=CRUISING_VELOCITY;

   //TASK 2:  SET AIRSHIP SHADOW PROPERTIES TO VALUES
   Shadow.alpha=.25;
   Shadow.scaleY=.25;
   Shadow.scaleX=.25;

   //TASK 3:  SET VILLAGE PROPERTIES TO INITIAL VALUES
   World.x=World.y=350;
   World.scaleY =.25;
   World.scaleX=.25;

   //TASK 4: ADD EVENT LISTENERS
   stage.addEventListener(Event.ENTER_FRAME, adjustWorld);
   stage.addEventListener(KeyboardEvent.KEY_DOWN, checkControlKey);
}

```

Function `adjustWorld()` Design

This function is an event handler that is called at every frame loop. Its primary objective is to adjust the location of the world below. Its complete set of tasks is as follows:

Task 1: Adjust the velocity of the airship to make the world below appear to move faster at low altitudes and slower at high altitudes. This small trick has a nuanced effect in creating a more realistic simulation.

Task 2: Change the world's `x` and `y` positions on the stage. As shown in Figure 5-13, the calculation of this altered world is based on the velocity and rotation of the airship. For example, as `Airship` changes direction by turning to the right, `World` will shift to the left.

The trigonometric functions `sin()` and `cos()` are used to compute the correct angle (in radians) in which to shift the world below. The radian value is needed to use the `sin()` and `cos()` function.

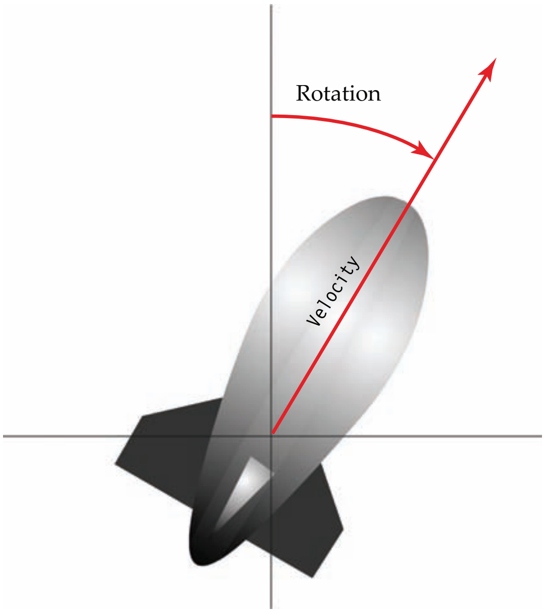


FIGURE 5-13 Angle and velocity diagram of the airship.

```
function adjustWorld(event:Event) {
   //TASK1:  ADJUST THE VELOCITY OF THE AIRSHIP ACCORDING TO ALTITUDE
   Airship.Veloicity=CRUISING_VELOCITY-.01 * (Airship.Altitude - CRUISING_
ALTITUDE);

```

```
    //TASK 2: CHANGE WORLD'S LOCATION RELATIVE TO THE AIRSHIP
    var toRadians:Number=Math.PI/180;
    World.x += (Airship.Velocity)*Math.sin(Airship.rotation*toRadians);
    World.y -= (Airship.Velocity)*Math.cos(Airship.rotation*toRadians);
}

```

Function `checkControlKey()` Design

This function, an event handler, has the simple task of examining the input from the user and responding. Using a `switch` statement makes this code more readable.

Responses to individual arrow keys occur by changing `Airship`'s rotation and altitude as follows:

`LEFTARROW` or `RIGHTARROW`: Rotate `Airship` and `Shadow` the appropriate number of degrees defined by the constant `TURN`.

`UPARROW`: Increase the altitude of `Airship`. If the maximum altitude has not been exceeded, activate a timer to decrease, in gradual measure, the scale of `World`. The timer is set to execute four times, for 40 milliseconds each time. The animation generated by this timer creates the illusion of the airship moving farther away.

`DOWNARROW`: Decrease the altitude of `Airship`. If the altitude has not gone below the minimum, a timer animation is triggered to create the illusion of the airship moving closer to the world below.

```

function checkControlKey(event:KeyboardEvent) {
    //TASK : EXAMINE AND RESPOND TO INPUT FROM USER
    switch (event.keyCode) {
        case RIGHTARROW :
            Airship.rotation+=TURN;
            Shadow.rotation+=TURN;
            break;
        case LEFTARROW :
            Airship.rotation-=TURN;
            Shadow.rotation-=TURN;
            break;
        case UPARROW :
            //AIRSHIP INCREASES ITS ALTITUDE
            Airship.Altitude+=ALTITUDE_AMT;
            if (Airship.Altitude > MAX_ALTITUDE) {
                Airship.Altitude=MAX_ALTITUDE;
            } else {
                atimer=new Timer(40,4);
            atimer.addEventListener(TimerEvent.TIMER, ZoomHigher);
            atimer.start();
         }
         break;
      case DOWNARROW :
         //AIRSHIP DECREASES ITS ALTITUDE
         Airship.Altitude-=ALTITUDE_AMT;
         if (Airship.Altitude < MIN_ALTITUDE) {
            Airship.Altitude=MIN_ALTITUDE;
         } else {
            atimer=new Timer(40,4);
            atimer.addEventListener(TimerEvent.TIMER, ZoomLower);
            atimer.start();
         }
   }
}

```

Function `ZoomHigher()` Design

This function has the simple task of examining the input from the user and responding to arrow keys. Using a `switch` statement makes this code more readable. Responses to individual arrow keys occur by changing the airship's rotation and altitude.

```
function ZoomHigher(event:TimerEvent) {
   var newSize:Number;
   newSize=World.scaleY-.0001;
   World.scaleY=World.scaleX=newSize;
   Shadow.scaleY=Shadow.scaleX=newSize;
   atimer.removeEventListener(TimerEvent.TIMER, ZoomHigher);
}

```

Function `ZoomLower()` Design

This function has the simple task of examining the input from the user and responding to arrow keys. Using a `switch` statement makes this code easier to read. Responses to individual arrow keys occur by changing the airship's rotation and scale.

```
function ZoomLower(event:TimerEvent) {
   var newSize:Number;
   newSize=World.scaleY+.0001;
   World.scaleY=World.scaleX=newSize;
   Shadow.scaleY=Shadow.scaleX=newSize;
   atimer.removeEventListener(TimerEvent.TIMER, ZoomLower);
}
```

5.11 Case Study 5: Billiard Physics

In this case study, we explore several important physical concepts in the game of billiards. Our focus is a pared-down game that revolves around a user hitting a cue ball with a cue stick.

Algorithm Design

Three display objects will be used by the ActionScript 3.0 code: `Stick`, `CueBall`, and `RedBall`. In addition, a table and an image of a green cloth will provide visual boundaries to aid the player in the game (Figure 5-14). We assume both balls on the billiard table are of the same mass. The frame rate for this application is set to 30 frames per second to smooth animations.

In addition to delving into the mechanisms behind aiming the cue stick and striking a ball with it, four important physical concepts are explored in this algorithm:

1. Velocity vectors
2. Friction
3. Impulse
4. Conservation of momentum

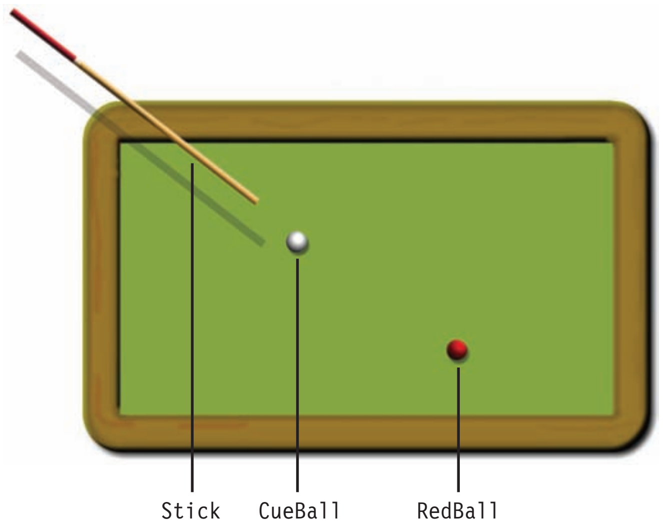


FIGURE 5-14 Billiards application.

Velocity Vector

The velocity of a given ball is the speed at which it moves in a given direction. Figure 5-15 shows the velocity vectors indicating both direction and speed for two balls. Notice that the red ball moves from left to right at nearly half the speed of the cue ball sitting to the right. The velocity vector of a given ball can be deconstructed into its velocity along the *x*-axis and its velocity along the *y*-axis. In this case study, we will refer to these velocities as `xVelocity` and `yVelocity`, respectively.

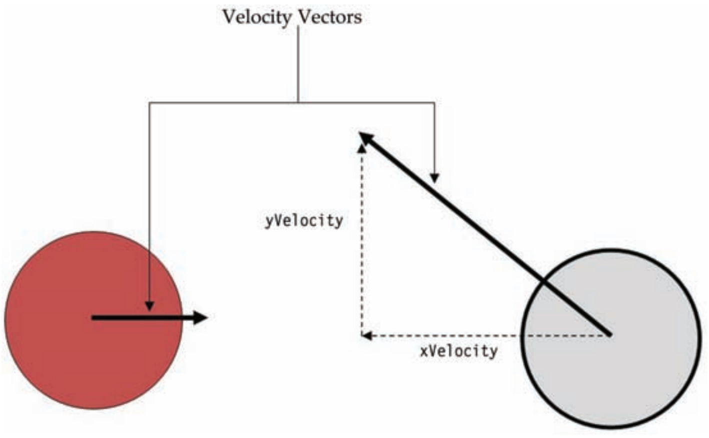


FIGURE 5-15 Velocity vectors for two balls in motion.

Friction

One important element of billiards is the green felt table cloth, which provides traction as well as friction. Friction is the resistance the ball encounters as it rolls over the surface of the table cloth, causing it to slow down. The velocity of a moving ball is constantly changing due to the force of friction. Friction will be applied to moving balls in this case study.

Impulse

Impulse is the change in momentum of an object. In other words, it refers to the hit and the resulting transference of velocity from one object to the object it has impacted.

In this game, the user attempts to strike the cue ball with the stick. Once hit, the cue ball moves with similar velocity and direction. Thus the momentum from the stick is transferred to the cue ball, forcing it to move across the billiards table with an initial velocity.

Conservation of Momentum

Momentum is a conserved quantity. The total momentum during and following a collision will be the same. For example, when one ball hits another, the first ball imparts velocity to the second through momentum. As shown in Figure 5-16, when the cue ball hits the red ball, the red ball, which is initially at rest, is transferred a good deal of speed; simultaneously, the velocity of the cue ball decreases. Together, the new velocities add up to the momentum before impact. Notice that after impact, the red ball moves in the direction of the impulse, which is the line joining the center of the two balls.

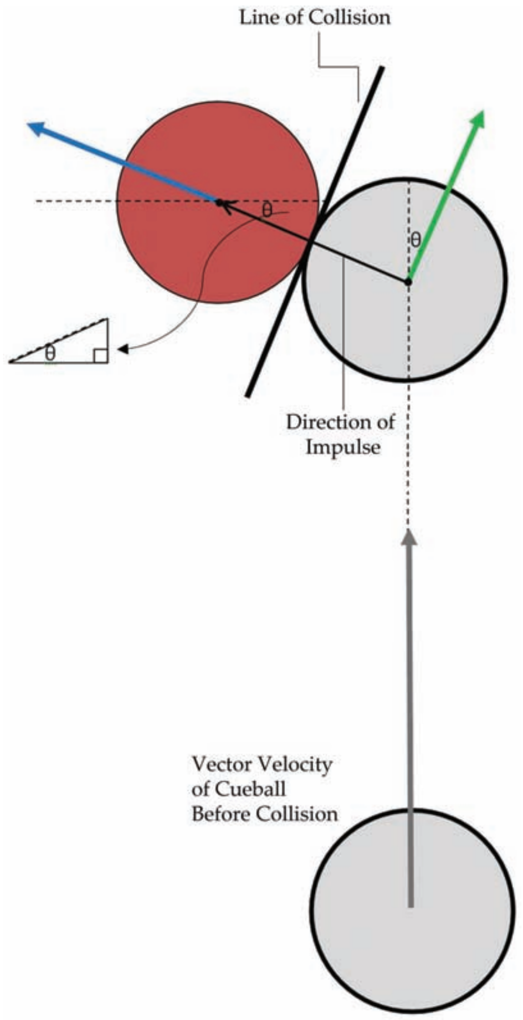


FIGURE 5-16 Momentum before and after the collision of two balls in motion.

The theory of conservation of momentum, along with the initial velocity vectors of the colliding balls, can be used to compute the trajectories after impact. The line of collision is drawn at a tangent to both balls at the point of contact. This line is perpendicular to the line of impulse, which passes through the center of the two balls at the point of contact. Using geometry, we can see that the line of collision also makes an angle θ with the vertical, and the line of impulse makes an angle θ with the horizontal. This behavior follows the geometric principle that the angle of incidence will equal the angle of reflection.

As you will notice soon in the AS3 code, the physics of billiards is complex. To develop such a game, programmers need to be proficient in trigonometry and the practical usage of the physics.

Interactive Cue Stick Functionality

The cue stick is the user's only tool for interaction. Placing a strict restriction in its functionality so that it aims solely at the cue ball will provide a better gaming experience for the user. The stick is flexible only in its ability to follow the mouse cursor, which enables the user to direct a shot from any angle. As Figure 5-17 illustrates, trigonometry is used to compute the angle of rotation, θ, for the stick.

```
θ = Math.atan2(dy, dx);
```

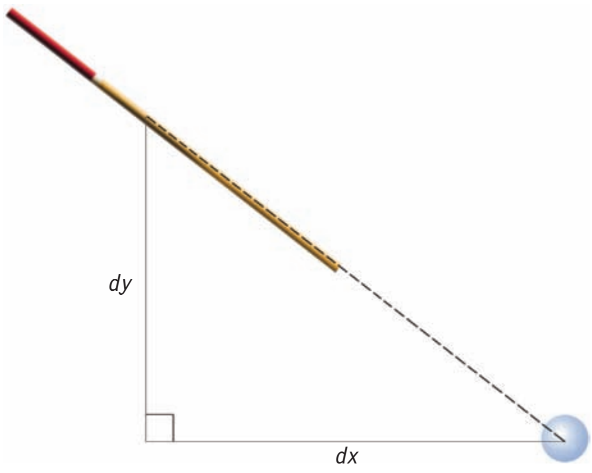


FIGURE 5-17 The billiards stick shooting the cue ball from angle θ.

Algorithm Functions

The AS3 code for this application consists of the class constructor `billiardsApp()`, which initializes the game, and `aimStick()`, `startShoot()`, and `billiardsEngine()`, which control the animation and drive the game. In addition, three auxiliary functions perform the tasks completing a shot and responding to movement and collisions. Table 5-12 lists the seven functions and Figure 5-18 illustrates their relationship.

TABLE 5-12 `billiardsApp` Program Functions

Function Name	Description
`billiardsApp()`	The class constructor. After launching, it initializes the visual element properties and registers the events for interaction and animation.
`aimStick()`	Performs the task of aiming the cue stick at the cue ball. This event handler imposes a strict and constant aim on the cue ball by responding to the `ENTER_FRAME` event.
`startShoot()`	An event handler that responds to the event `MOUSE_DOWN`. This function starts the task of allowing the user to shoot the cue ball. It also initiates the event handler, finishShoot(), for completing the shot. Finally, it initializes collision detection tasks by calling the appropriate functions.
`finishShoot()`	An event handler that responds to the event `MOUSE_UP`. This function ends the task of the user shooting the cue ball.
`billiardsEngine()`	Drives the game of billiards by calling appropriate functions to move the balls and check for and respond to collisions.
`moveBall()`	Moves an individual ball, computes the decrease in velocity from frictional force, and responds to ball and table wall collisions.
`checkCollision()`	Checks for a collision between the two balls. If a collision has occurred, it responds by computing the new trajectories.

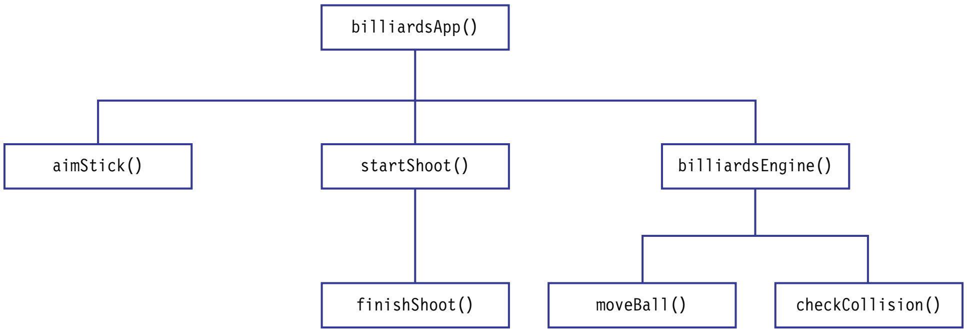


FIGURE 5-18 Relationship of the functions used by the billiards game.

Game Constants

Billiards will utilize many of the same constants required by the game of Pong. The game of Pong used constant values when identifying collisions and reversing direction of the ball. These constants can be divided into three categories for the billiards game. The first represents the radius of the ball, which is set to 16 pixels. This constant is used to calculate the boundaries within which the ball can freely move. The second category represents the concrete boundaries of the table—`TOP`, `BOTTOM`, `LEFT`, and `RIGHT`. As in the Pong game, the `TOP` boundary is not zero, but rather the point at which the ball strikes the top rail of the table, which is the radius of the ball. Lines 12–15 are constant declarations for these boundaries.

The final constant is `REVERSE`, −1. When the ball's current velocity is multiplied by this constant, the velocity is negated to reverse the ball's direction.

The `package` requires the `display` and `events` library classes. The final lines contain closing curly brackets to conclude the `billiardsApp` class and the package.

```
1   package {
2      import flash.display.*;
3      import flash.events.*;
4
5      public class billiardsApp extends MovieClip {
6         // GAME CONSTANTS
7         const REVERSE:Number=-1;     //REVERSE DIRECTION OF MOVING OBJECTS
8         const DIAMETER:Number=20;    //DIAMETER OF THE BILLIARD BALLS
9         const FRICTION:Number=.96;   //FRICTION OF THE BALLS ON THE TABLE
10        const MINSPEED:Number=.1;    //A BALL IS CONSIDERED STOPPED
11
12        const LEFT:Number=285;       //LEFT OF THE TABLE
13        const RIGHT:Number=734;      //RIGHT OF THE TABLE
14        const TOP:Number=241;        //TOP OF THE TABLE
15        const BOTTOM:Number=480;     //BOTTOM OF THE TABLE
16

⋮

157    }
158 }
```

The initial velocity vector of the cue ball is used to compute the resulting velocity vectors for both balls following a collision. Thus, if the ball hits the side of the table at an angle of 25 degrees, it will rebound at that angle.

Function `billiardsApp()` Design

This function initializes the ball properties and registers an event to wait for the user to use the cue stick.

```
17  function billiardsApp() {
18     //TASK 1:  SET THE VELOCITIES OF EACH BALL TO ZERO
19     CueBall.width=DIAMETER;
20     CueBall.height=DIAMETER;
```

```
    RedBall.width=DIAMETER;
    RedBall.height=DIAMETER;

    //TASK 2:  POSITION BALLS ON THE TABLE
    CueBall.x=400;
    CueBall.y=350;
    RedBall.x=600;
    RedBall.y=350;

    //TASK 3:  SET THE VELOCITY OF EACH BALL TO ZERO
    CueBall.xVelocity=0;
    CueBall.yVelocity=0;
    RedBall.xVelocity=0;
    RedBall.yVelocity=0;

    //TASK 4:  ALWAYS AIM THE STICK AT THE CUE BALL
    stick.addEventListener(Event.ENTER_FRAME,aimStick);

    //TASK 5:  WAIT FOR USER TO START TO SHOOT MOUSE DOWN
    stick.addEventListener(MouseEvent.MOUSE_DOWN,startShoot);
    //TASK 6:  AT EVERY FRAME, CHECK IF A BALL SHOULD MOVE
    addEventListener(Event.ENTER_FRAME,gameEngine);
}

```

Function `aimStick()` Design

Aiming the stick requires positioning its tip at the cue ball. The angle of the stick can be adjusted by moving it along with the mouse. The angle is computed using right triangle properties.

```
function aimStick(event:Event) {
    //ROTATES THE ANGLE OF THE CUE STICK TO POINT AT THE WHITE BALL
    var dx:Number=CueBall.x-mouseX;
    var dy:Number=CueBall.y-mouseY;
    var angle:Number=Math.atan2(dy,dx);
    stick.rotation=angle*180/Math.PI;
    stick.x=mouseX;
    stick.y=mouseY;
}

```

Function `startShoot()` Design

This function begins the process of shooting the cue ball with the stick. If the ball has successfully been hit, the event handler function `FinishShoot()` is called to complete the task.

```
function startShoot(event:MouseEvent) {
   //TASK 1:  LOCATE THE DISTANCE BETWEEN THE STICK AND THE BALL
   var dx:Number=CueBall.x-mouseX;
   var dy:Number=CueBall.y-mouseY;
   var dist:Number=Math.sqrt(dx*dx+dy*dy);

   //TASK 2:  IF WITHIN SHOOTING DISTANCE,
   //         WAIT FOR THE USER TO FINISH THE SHOT
   if (dist > 110) {
      //TASK 3: STORE THE STARTING POSITION OF THE SHOT
      stick.startx=stick.x;
      stick.starty=stick.y;
      stick.addEventListener(Event.ENTER_FRAME,FinishShoot);
   }
}

```

Function `FinishShoot()` Design

This function is the event handler for completing a shot. The velocity of `CueBall` is computed based on the distance of the stick and the ball when the process began.

```
function FinishShoot(event:Event) {
   //TASK 1:  COMPUTE DISTANCE BETWEEN STICK AND CUE BALL
   var dx:Number=CueBall.x-stick.x;
   var dy:Number=CueBall.y-stick.y;
   var dist:Number=Math.sqrt(dx*dx+dy*dy);

   //CHECK IF THE STICK HAS JUST HIT THE WHITE BALL
   if (dist < 110) {
      //TASK 2:  COMPUTE THE NEW VELOCITY OF THE WHITE BALL
      CueBall.xVelocity=(stick.x-stick.startx)/4;
      CueBall.yVelocity=(stick.y-stick.starty)/4;
      //TASK 3:  THE FINISH SHOOT OPERATION IS DONE
      stick.removeEventListener(Event.ENTER_FRAME,FinishShoot);
   }
}

```

Function `gameEngine()` Design

This function is the event handler for the `ENTER_FRAME` event registered by the application constructor `billiardsApp()`. It is also the animation engine that takes the game from static mode to dynamic play by directing the movement at regular intervals. In addition, this function is responsible for directing collision detection between the two balls on the Stage.

```
function gameEngine(event:Event) {
   //TASK 1: MOVE EACH BALL ON STAGE
   moveBall(CueBall);
   moveBall(RedBall);

   //TASK 1: CHECK IF BALLS HAVE COLLIDED
   checkBalltoBall();
}
```

Function moveBall() Design

This function moves an individual ball, applies friction, and checks for collisions along the table walls. The object `Ball` is a parameter: It is a variable that represents either the `CueBall` or the `RedBall`. By utilizing this variable, this function is written to perform the tasks for either ball.

```
function moveBall(Ball) {
   //TASK 1: MOVE THE GIVEN BALL ITS FIXED SPEED
   Ball.x+=Ball.xVelocity;
   Ball.y+=Ball.yVelocity;

   //TASK 2:  APPLY FRICTION TO THE  BALL
   Ball.xVelocity*=FRICTION;
   Ball.yVelocity*=FRICTION;

   //TASK 3: IF A WALL IS HIT, CHANGE  DIRECTION
   if (Ball.x > RIGHT) {
      Ball.xVelocity*=REVERSE;
      Ball.x=RIGHT;
   } else if (Ball.x < LEFT) {
      Ball.xVelocity*=REVERSE;
      Ball.x=LEFT;
   }
   if (Ball.y > BOTTOM) {
      Ball.yVelocity*=REVERSE;
      Ball.y=BOTTOM;
   } else if (Ball.y < TOP) {
      Ball.yVelocity*=REVERSE;
      Ball.y=TOP;
   }

   //TASK 4: DETERMINE IF THE BALL HAS STOPPED MOVING
   var speed:Number;
   speed =Math.sqrt(Ball.xVelocity*Ball.xVelocity+Ball.yVelocity*Ball.yVe-
locity);
```

```
            if (speed < MINSPEED) {
               Ball.xVelocity=0;
               Ball.yVelocity=0;
      }
}

```

Function checkBalltoBall() Design

This function is called to compute the resulting velocities and angles when the balls have collided with each other.

```
function checkBalltoBall() {
   //TASK 1:  COMPUTE THE DISTANCE BETWEEN THE TWO BALLS
   var dx:Number=CueBall.x-RedBall.x;
   var dy:Number=CueBall.y-RedBall.y;
   var dist:Number=Math.sqrt(dx*dx+dy*dy);

   // HAVE THE BALLS COLLIDED?
   if (dist < DIAMETER) {
      //TASK 2: COMPUTE THE ANGLE OF COLLISION
      var angle=Math.atan2(dy,dx);

      //TASK 3:  COMPUTE THE COSINE AND SINE OF THE ANGLE OF COLLISION
      var cosineAngle=Math.cos(angle);
      var sinAngle=Math.sin(angle);

      //TASK 4:  COMPUTE THE VELOCITIES ALONG THE ANGLE OF COLLISION
      var xVelocity2=cosineAngle*CueBall.xVelocity+sinAngle*CueBall.yVelocity;
      var yVelocity1=cosineAngle*CueBall.yVelocity-sinAngle*CueBall.xVelocity;
      var xVelocity1=cosineAngle*RedBall.xVelocity+sinAngle*RedBall.yVelocity;
      var yVelocity2=cosineAngle*RedBall.yVelocity-sinAngle*RedBall.xVelocity;

      //TASK 5:  ASSIGN NEW TRAJECTORIES FOR BOTH BALLS
      CueBall.xVelocity=cosineAngle*xVelocity1-sinAngle*yVelocity1;
      CueBall.yVelocity=cosineAngle*yVelocity1+sinAngle*xVelocity1;
      RedBall.xVelocity=cosineAngle*xVelocity2-sinAngle*yVelocity2;
      RedBall.yVelocity=cosineAngle*yVelocity2+sinAngle*xVelocity2;
   }
}
```

Review Questions

1. What is a logical expression?
2. Give an example of a simple Boolean expression.
3. Give an example of a compound Boolean expression. How does a compound Boolean expression differ from a simple Boolean expression?

4. Identify the three logical operators and describe how they work.
5. What is the `if` statement used for?
6. List and briefly describe the six relational operators provided by AS3.
7. What value is used to represent false in the computer?
8. What value is used to represent true in the computer?
9. How does a `switch` statement differ from an `if` statement?

Exercises

Evaluate the following logical expressions. Provide an answer of true or false.

1. `(!0 )`
2. `(5 + 4 < 3 && 7 + 3 <= 20 )`
3. `(int (3.9) != 3)`
4. `(!(7 == 7 ))`
5. `(3 % 2)`
6. `(!1 || !0)`
7. `(3 != 2 || 7 == 7 && 10 < 9)`

Determine the output for each of the following program segments. Assume that `n1` and `n2` have the following assignments prior to the execution of each `if` operation:

```
var n1:Number = 2;
var n2:Number = 3;
```

8.
```
if (n1 < n2){
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

9.
```
if (n1 == '2'){
    trace ( "n1 = " , n1 );
}
```

10.
```
if (n1){
    trace ( "The value of n1 is nonzero." );
}
```

11.
```
if (n1 == n2 - 1){
    var temp:Number = n2;
    n2 = n1;
    n1 = temp;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

12.
```
if ((n1 < n2) && ( n2 != 10)){
    var sum:int = n1 + n2;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
    trace ( "Sum = " , sum );
}
```

13.
```
if ((n1 > n2) || ( n1 - n2 < 0)){
    n1 += 1;
    n2 -= 1;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

14.
```
if (n1 > n2){
    n1 += 1;
}else{
    n2 -= 1;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

15.
```
if (n1 < n2){
    n1 += 1;
}else{
    n2 -= 1;}
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
```

16.
```
if (!(n1 > n2)){
    n1 += 1;
}else{
    n2 -= 1;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

17.
```
if ((n1 > n2) || ( n1 * n2 < 0)){
    n1 +=1;
    n2 -= 1;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
trace ( "n1 = " , n1 );
trace ( "n2 = " , n2 );
```

18.
```
if (n1 < n2){
    n1 +=1;
    n2 -= 1;
    trace ( "n1 = " , n1 );
    trace ( "n2 = " , n2 );
}
```

```
trace ( "n1 = " , n1 );
trace ( "n2 = " , n2);
```

19. Write an `if` statement that displays “not zero” when the value of variable `num1` (a `Number`) is nonzero.

20. Write an `if` statement that displays “BLUE” when both `num1` and `num2` (`Number` variables) are positive or both negative.

21. Write an `if` statement that displays “IN RANGE” when `num1` (a `Number` variable) is between −10 and 10.

22. Rewrite the following segment of code in the most efficient way possible.

```
if ((n1 < n2) && (n1 == 0)){
    trace ( "ORANGE");
}
if ((n1 < n2) && ( n1 != 0)){
    trace ( "APPLE");
}
if (n1 >= n2){
    trace ( "BANANA");
}
```

23. Write an `if` statement that displays “Out of Range” if the input text box entry `input0` is negative or is greater than 100.

24. Write an efficient `if` statement to assign `num3` the following values:

    8 if `num1` is less than 1.5

    7 if `1.5 <= num1 < 2.5`

    6 otherwise

25. Write a segment of AS3 code to do the following:
    a. Assume values exist for variables `num1`, `num2`, and `num3` (all uints).
    b. If `num3` is a 1, calculate and display the sum of `num1` and `num2`.
    c. Otherwise, output the difference of `num1` and `num2`.

26. Write a `switch` statement that does the following:
    a. Increases `balance` (a `Number` variable) by adding `amount` (a `Number` variable) to it if the value of `transaction` (a `String` variable) is “Deposit”.
    b. Decreases `balance` by subtracting `amount` from it if the value of `transaction` is “Withdrawal”.
    c. Display the value of `balance` if the value of `transaction` is “Display”.
    d. Display “Illegal transaction” otherwise.

Projects

1. The city of Flowerville bills its residents for water consumption. The charges are based on usage according to the following table:

Water Used	Rate
First 200 cubic meters	$5.00 minimum cost
Next 430 cubic meters	$0.10 per cubic meter
Next 570 cubic meters	$0.07 per cubic meter
More than 1000 cubic meters	$0.02 per cubic meter

   Create an application that computes the charges for a given amount of water usage. Provide error detection capabilities to identify incorrect input by the user.

2. Write an application that allows the user to enter a date and then determine its validity. If the date is invalid, an error message should be displayed explaining precisely where the error is. If the date is valid, the program should compute the day of the year. During leap years, there are 29 days in February. During non-leap years, there are exactly 28 days. To be a leap year, the year must be evenly divisible by 4. However, not all years evenly divisible by 4 are leap years. Years whose last two digits are zero are century years; for example, 1800, 1900, and 2000 are century years. Century years are leap years only if they are evenly divisible by 400. Thus the years 1600 and 2000 are leap years; 1700, 1800, and 1900 are not leap years.

   Example date 1:
   Day: 1 Month: 13 Year: 2001
   Display: This date has an error. 13 is an invalid month.
   Example date 2:
   Day: 12 Month: 3 Year: 3012
   Display: This is day number 72 of the leap year 3012.

3. Write an interactive application that plays the game of Rock, Paper, Scissors. In this game, the user will play against the computer. Provide buttons for the user to choose "rock," "paper," or "scissors." Your program must be able to generate a random choice for the computer. The winner is the one whose choice dominates the other.

4. Rewrite the game of Pong in Case Study 1 so that when the ball falls through the `BOTTOM` of the stage, the listener event moving the ball is eliminated.

5. Enhance the game of Pong further by keeping track of how many successful hits the user makes. Make the game more challenging by increasing the velocity of the ball with each successful hit.
6. Modify the pet fish application to have the fish die after a set number of feedings and rotate and float to the top of the tank.
7. Modify the pet fish application to have the fish randomly select two toys to play with.
8. Make enhancements to the pet fish application from Case Study 3.
   a. Add animation to each stage of its life—eating, sleeping, and moving. Use a `switch` statement.
   b. Introduce an obstacle to fish tank that requires the fish to navigate around it.
   c. Decrease the food size as the pet fish feeds.
   d. Add a predator object to the aquarium.