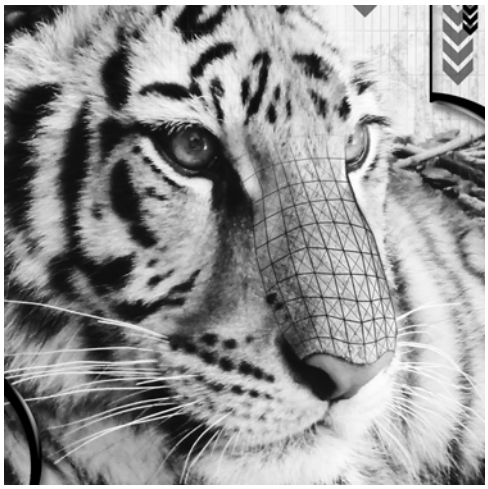


Chapter 12

Case Studies: Maze and Platform Games

We've now got a large toy box of game design skills we can use to start building an increasingly complex number of games in different genres. In this chapter we're going to take a close look at two of the most popular: maze games and platform games. We're also going to look at a few important new techniques along the way:

- Converting a sub-object's local coordinates to global coordinates
- The display list and object stacking order
- Rotating an object toward another object
- Random motion
- Artificial intelligence (AI)

We're going to use all these new techniques to build a platform game called Bug Catcher. But before we do, let's take one small step further and apply what we've learnt about loops and arrays to find out how you can add a maze environment to Monster Mayhem from chapter 8 that the monsters can wander through.

A-MAZE-ing Monster Mayhem

Now that we know how to use loops and arrays to efficiently check for collisions with lots of objects, we can use these new skills to create a maze for the Monster Mayhem game from chapter 8. Appropriately enough, this new version is called Amazing Monster Mayhem, and you'll find it in the chapter's source files. It's a streamlined version of the game which is just focused on the new maze features. You can move the character around with the arrow keys, and the monsters wander around randomly, as shown in Figure 12-1.

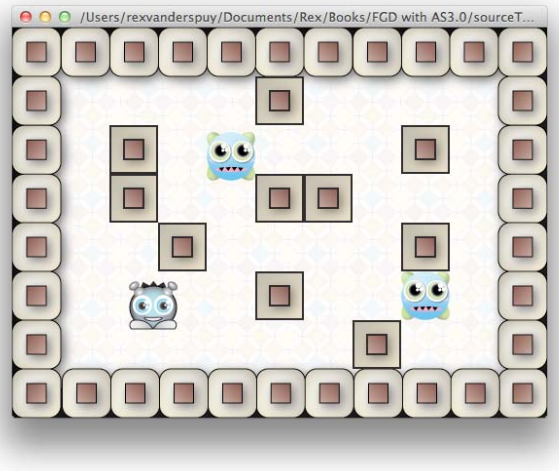


Figure 12-1. Use loops and arrays to efficiently manage the monsters maze walls.

There are some important new techniques at work here that we'll look at in detail ahead:

- Both monsters are created in the same way that the boxes are. They're added to an array, and all their movement and collision detection is managed by loops.
- The monsters don't change direction based on a timer. Instead, they change direction when they've moved a distance that's equal to the width of one of the square boxes. This is to ensure that the monster can accurately change direction at an intersection in the maze.
- A **constant** is used to represent the size of one of the maze boxes. Constants are variables that never change their values in a program.

Here's the entire **`AmazingMonsterMayhem`** application class that you can use as a reference. The application class uses all the other support classes, such as `Character`, `Box`, and `Monster` from the original game, so refer back to chapter 8 if you have any questions about how those classes work. We'll examine all the new features of this example in detail in the pages ahead.

```
package
{
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.ui.Keyboard;

  [SWF(width="550", height="400",
  backgroundColor="#FFFFFF", frameRate="50")]

  public class AmazingMonsterMayhem extends Sprite
  {
    //Declare the variables to hold
    //the game objects
    private var _character:Character;
    private var _background:Background;
    private var _monster1:Monster;
    private var _monster2:Monster;

    //Arrays to store the boxes and their x and y positions
    private var _boxes:Array = new Array();
    private var _boxPositions:Array = new Array();

    //Arrays to store the monsters and their x and y positions
```

```
private var _monsters:Array = new Array();
private var _monsterPositions:Array = new Array();

//A variable to count the frames
private var _frameCounter:uint = 0;

//A constant to represent the size of one of
//the boxes that makes up the maze walls
private const BOX_SIZE:uint = 50;

public function AmazingMonsterMayhem()
{
  //Create the game objects
  _character = new Character();
  _background = new Background();

  //Add the game objects to the stage
  addGameObjectToStage(_background, 0, 0);
  addGameObjectToStage(_character, 250, 300);

  //Event listeners
  stage.addEventListener
    (KeyboardEvent.KEY_DOWN, keyDownHandler);
  stage.addEventListener
    (KeyboardEvent.KEY_UP, keyUpHandler);
  this.addEventListener
    (Event.ENTER_FRAME, enterFrameHandler);

  //The Monsters
  //Set the monster x and y positions
  _monsterPositions
    = [
        [50, 300],
        [450, 50]
      ];

  //Make the monsters
  for(var i:int = 0; i < _monsterPositions.length; i++)
  {
    //Create a monster object
    var monster:Monster = new Monster();

    //Add the monster to the stage
    addChild(monster);
    monster.x = _monsterPositions[i][0];
    monster.y = _monsterPositions[i][1];

    //Add it to the monsters array
    //for future use
    _monsters.push(monster);
  }

  //The Boxes
  //Set the box x and y positions
  _boxPositions
    = [
        [100, 100],
        [100, 150],
        [150, 200],
        [250, 50],
        [250, 150],
        [250, 250],
        [300, 150],
        [350, 300],
```

```
        [400, 100],
        [400, 200]
      ];

  //Make the boxes
  for(var j:int = 0; j < _boxPositions.length; j++)
  {
    //Create a box object
    var box:Box = new Box();

    //Add the box to the stage
    addChild(box);
    box.x = _boxPositions[j][0];
    box.y = _boxPositions[j][1];

    //Add it to the boxes array
    //for future use
    _boxes.push(box);
  }
}

private function enterFrameHandler(event:Event):void
{
  //Move the game character and
  //check its stage boundaries
  _character.x += _character.vx;
  _character.y += _character.vy;
  checkStageBoundaries(_character);

  //Collision between the character and boxes
  for(var i:int = 0; i < _boxes.length; i++)
  {
    Collision.block(_character, _boxes[i]);
  }

  //The monsters
  //Update the frame counter
  _frameCounter++;

  for(var j:int = 0; j < _monsters.length; j++)
  {
    //Create a temporary local variable
    //to easily access the current monster in the loop
    var monster:Monster = _monsters[j];

    //Make the monsters change direction
    //every 50 frames
    if(_frameCounter == BOX_SIZE)
    {
      changeMonsterDirection(monster);
    }

    //Move the monster
    monster.x += monster.vx;
    monster.y += monster.vy;

    //Check its stage boundaries
    checkStageBoundaries(monster);

    //Check for collisions with the boxes
    for(var k:int = 0; k < _boxes.length; k++)
    {
      Collision.block(monster, _boxes[k]);
         }
```

```
  }

  //Reset the frame counter if it equals
  //the size of a box that makes up
  //the maze walls (50)
  if(_frameCounter == BOX_SIZE)
  {
    _frameCounter = 0;
  }
}
private function checkStageBoundaries(gameObject:Sprite):void
{
  if (gameObject.x < 50)
  {
    gameObject.x = 50;
  }
  if (gameObject.y < 50)
  {
    gameObject.y = 50;
       }
  if (gameObject.x + gameObject.width > stage.stageWidth - 50)
  {
    gameObject.x = stage.stageWidth - gameObject.width - 50;
  }
  if (gameObject.y + gameObject.height > stage.stageHeight - 50)
  {
    gameObject.y = stage.stageHeight - gameObject.height - 50;
  }
}

private function changeMonsterDirection(monster:Monster):void
{
  var randomNumber:int = Math.ceil(Math.random() * 4);
  if(randomNumber == 1)
  {
    //Right
    monster.vx = 1;
    monster.vy = 0;
  }
  else if (randomNumber == 2)
  {
    //Left
    monster.vx = -1;
    monster.vy = 0;
  }
  else if(randomNumber == 3)
  {
    //Up
    monster.vx = 0;
    monster.vy = -1;
  }
  else
  {
    //Down
    monster.vx = 0;
    monster.vy = 1;
  }
}
private function keyDownHandler(event:KeyboardEvent):void
{
  if (event.keyCode == Keyboard.LEFT)
  {
    _character.vx = -5;
  }
```

```
      else if (event.keyCode == Keyboard.RIGHT)
      {
        _character.vx = 5;
      }
      else if (event.keyCode == Keyboard.UP)
      {
        _character.vy = -5;
      }
      else if (event.keyCode == Keyboard.DOWN)
      {
        _character.vy = 5;
      }
    }
    private function keyUpHandler(event:KeyboardEvent):void
    {
      if (event.keyCode == Keyboard.LEFT
      || event.keyCode == Keyboard.RIGHT)
      {
        _character.vx = 0;
      }
      else if (event.keyCode == Keyboard.DOWN
      || event.keyCode == Keyboard.UP)
      {
        _character.vy = 0;
      }
    }
    private function addGameObjectToStage
      (gameObject:Sprite, xPos:int, yPos:int):void
    {
      this.addChild(gameObject);
      gameObject.x = xPos;
      gameObject.y = yPos;
    }
  }
}
```

Making monsters

The monsters are created and added to the maze using exactly the same technique we used to add the boxes. In fact, the code is identical except for the object names. The code first needs two arrays: one to store the x and y positions of the monsters, and another to store the monster objects themselves.

```
private var _monsters:Array = new Array();
private var _monsterPositions:Array = new Array();
```

The class constructor then creates the monsters objects, adds them to the stage at their assigned positions, and copies them into the `_monsters` array so that we can use them again later.

```
//Set the monster x and y positions
_monsterPositions
        = [
                [50, 300],
                [450, 50]
          ];

//Make the monsters
for(var i:int = 0; i < _monsterPositions.length; i++)
{
        //Create a monster object
        var monster:Monster = new Monster();

        //Add the monster to the stage
        addChild(monster);
        monster.x = _monsterPositions[i][0];
        monster.y = _monsterPositions[i][1];
```

```
        //Add it to the monsters array
        //for future use
        _monsters.push(monster);
}
```

As you can see, this code is identical to the code used to make and add the boxes that are used for the maze walls. You'll be using this code routinely from now on whenever you need to add more than one object of a single type in your games.

In this example there are only two monsters, so it might seem like overkill to manage them using an array. But here's the great advantage to this system: if you need to add any more monsters to the game, *all you need to do is add their new positions in the `_monsterPositions` array*. Nothing else. The rest of the code will automatically adjust to accommodate for them – whether it's two more monsters or a hundred. You don't have to change a single line of code anywhere else in the class. Your life as a game programmer has just become infinitely easier.

An important thing to know about `for` loops is that if you use two or more of them in the same code block, you have to change the name of the loop counter variable for the second loop. That means that if your counter variable is "i" in the first loop, use "j" in the second loop. You then need to remember to use the correct loop counter variable name to access the array elements.

In this example, the loop that makes the monsters uses "i", and the loop that makes the boxes uses "j", like this:

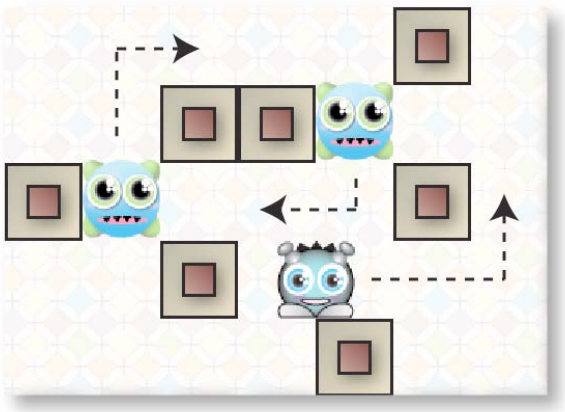
```
for(var i:int = 0; i < _monsterPositions.length; i++)
{
  var monster:Monster = new Monster();
  addChild(monster);
  monster.x = _monsterPositions[i][0];
  monster.y = _monsterPositions[i][1];
  _monsters.push(monster);
}

for(var j:int = 0; j < _boxPositions.length; j++)
{
  var box:Box = new Box();
  addChild(box);
  box.x = _boxPositions[j][0];
  box.y = _boxPositions[j][1];
  _boxes.push(box);
}
```

If you have any more for loops in the same block of code, just keep going down the alphabet for your counter variable names, such as "k" and "l". This is a standard programming convention, and you'll see it in use in all the code in the rest of the book.

Moving monsters in a maze

When you make a maze game, you want your game objects to be able to smoothly and cleanly turn corners and move through intersections without getting stuck on the edges of the maze walls, as shown in Figure



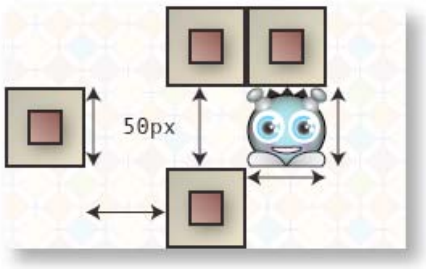
12-2.

Figure 12-2. Make your game objects move smoothly through the maze environment.

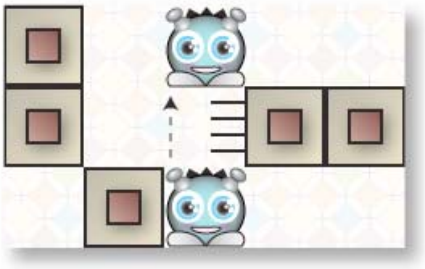
This is easy to get right if you stick to a few simple guidelines:

- Make sure that your moving objects, like the monsters and game character, are the same size as one of the boxes that make up the maze wall. So, if the boxes are 50 by 50 pixels, make sure that your characters are exactly the same size.
- Make sure that the gaps between wall segments are the same size as your game objects. So if your game objects are 50 by 50 pixels, makes sure you leave an exact 50 pixel wide gap between maze wall boxes. This will let your objects squeeze through perfectly without getting stuck.
- Make sure that the vx and vy values of your moving objects are numbers that evenly divide the size of each wall segment. For example, if the maze wall boxes are 50 pixels square, make sure that your moving objects move at the rate of 1, 2, 5 or 10 pixels per frame. That will ensure that the moving object will eventually align at the exact corner of an intersection. It will allow the object turn into the corner without getting stuck on an edge. If you use vx or vy values that don't divide evenly into 50, such as 3, 4 or 6, your objects will slightly overshoot or fall short of the intersection and you won't be able to squeeze them through. In this example, the character moves at 5 pixels per frame and the monsters move at 1 pixel per frame, so they can move through the maze with perfect ease.

Figure 12-3 illustrates these simple guidelines.



Use consistent object sizes
and spacing.



Move objects at a speed which
divides evenly into the size
of a wall segment, such as
1, 2, 5 or 10 pixels per frame,
as in this example.

Figure 12-3. Guidelines to keep in mind for maze games.

Changing direction based on the number of elapsed frames

There's one more thing you need to keep in mind if you're using moving objects that change their direction based on time, like our monsters do. *You can't use a timer to change their direction*. Instead, you have to count up to a maximum number of frames, such as 50, and then change their directions when that number of frames have elapsed.

The reason for this is that AS3.0's timer isn't exactly synchronized with the frame rate. Even though your game might be set to run at 60 frames per second, there can be slight drops in the frame rate due to things like heavy CPU usage or other things going on in your computer that might slightly slow the Flash Player down. As a result, timers, which are synchronized with your computer's internal clock, will go slightly out of sync with the Flash Player's frame rate. This isn't usually a problem, but it is for maze games. You have to ensure that your maze game objects are synchronized with the frame rate so that when they change direction they're in a position that's exactly aligned with the maze passages. If you depend on a timer to change their direction, they'll eventually start changing positions slightly ahead of the frame rate, and become misaligned with the maze walls and intersections. This will make it really difficult for them to turn corners into passageways.

This is not difficult to implement, here's how:

1. Create a variable to count the number of frames. Initialize it to zero. Update this variable by 1 In the `enterFrameHandler`, like this:

```
_frameCounter++;
```

With every new frame that passes, this variable will be updated by one.

2. When the number of frames elapsed matches the height or width of one of the maze wall boxes, make your objects change direction, like this:

```
if(_frameCounter == BOX_SIZE)
{
  changeMonsterDirection(monster);
}
```

3. Reset the `_frameCounter` variable back to zero, so that it can start counting from the beginning all over again.

```
_frameCounter = 0;
```

As long as your monsters are moving with a velocity that divides evenly into the width or height of the maze wall boxes, they'll change direction when they've moved exactly 50 pixels. This will align them perfectly with the maze walls and let them freely change direction at maze intersections and move cleanly through passageways.

Let's take a look at how this technique is implemented, and how loops are used to move the monsters in the `_monsters` array.

The code creates a variable called `_frameCounter` to count the frames and initializes it to zero when the game starts.

```
private var _frameCounter:uint = 0;
```

It also creates a **constant** that stores the number that represents the size of a box.

```
private const BOX_SIZE:uint = 50;
```

Now whenever the code reads the constant `BOX_SIZE`, it will replace it with "50" which is the width and height of each box that makes up the maze walls.

Constants are declared using the `const` keyword and this is the first time we've used one in this book. A constant is exactly the same as a variable, except that its value can never be changed. Its value is *constant.* This is helpful because it means that your program will be free of errors that result from values being accidentally overwritten by other values when your program depends on them to remain unchanged.

Constants also make it easy to quickly change the functioning of the program. By changing the values of the constants at the top of the class, you can completely change the fundamental parameters of your game. It makes the class very customizable by just changing a few simple values.

By convention, constants are always written in full uppercase characters, which make them easy to spot in your code. If the name of a constant is made up of two or more words, the convention is to separate the words with an underscore character, like this: TWO_WORDS. Like variables, constants are also referred to as **properties**. We'll be using constants in this book from now on for any values which don't change.

Here's all the code in the `enterFrameHandler` that makes the monsters move:

```
//Update the frame counter
_frameCounter++;

for(var j:int = 0; j < _monsters.length; j++)
{
  //Create a temporary local variable
  //to easily access the current bug in the loop
  var monster:Monster = _monsters[j];

  //Make the monsters change direction
  //every 50 frames
  if(_frameCounter == BOX_SIZE)
  {
    changeMonsterDirection(monster);
  }

        //Move the monster
        monster.x += monster.vx;
        monster.y += monster.vy;

        //Check its stage boundaries
        checkStageBoundaries(monster);

        //Check for collisions with the boxes
        for(var k:int = 0; k < _boxes.length; k++)
        {
          Collision.block(monster, _boxes[k]);
        }
}
//Reset the frame counter if it equals
//the size of the boxes that make up
//the maze walls (50)
if(_frameCounter == BOX_SIZE)
{
        _frameCounter = 0;
}
```

The code that moves the monster is a `for` loop, and it sits between the directive that updates the `_frameCounter` and the directive that resets it if the it equals the size of a box. Here's the way this bit of code is structured:

4. Update frame counter.

```
_frameCounter++;
```

5. Loop through the `_monsters` array to move the monsters, change their directions and check their collisions.

```
for(var j:int = 0; j < _monsters.length; j++)
{
```

```
  //... move monsters...
}
```

6. Reset the frame counter to zero if it equals the value of `BOX_SIZE` (which is 50).

```
if(_frameCounter == BOX_SIZE)
{
        _frameCounter = 0;
}
```

The code has to be structured in this way so that the frame counter is accurate and doesn't come up one early or one late.

The `for` loop first creates a temporary local variable to store the current monster object that the loop is accessing in the array:

```
var monster:Monster = _monsters[j];
```

The first time the loop runs, this will represent the first monster object in the array. The second time the loop runs, it will represent the second monster. (There are only two monster objects in the `_monsters` array, so the loop will only run twice).

The code then checks to see if `_frameCounter` equals 50:

```
if(_frameCounter == BOX_SIZE)
{
  changeMonsterDirection(monster);
}
```

This will be true every 50 frames and means that `changeMonsterDirection` will be called every 50 frames. Because the monsters move at the rate of 1 pixel per frame, they will have moved exactly 50 pixels each time this method is called. 50 pixels is of course exactly the same height and width as the boxes that make up the maze walls. This means the monsters will always change direction when they're exactly aligned with a box, intersection or passageway.

The loop then moves the monster and checks its stage boundaries:

```
//Move the monster
monster.x += monster.vx;
monster.y += monster.vy;

//Check its stage boundaries
checkStageBoundaries(monster);
```

A second loop inside the first then checks the monster for any collisions with the maze walls

```
for(var k:int = 0; k < _boxes.length; k++)
{
        Collision.block(monster, _boxes[k]);
}
```

It loops through all the boxes in the `_boxes` array and checks for collisions using the familiar `Collision.block` method. This prevents the monster from moving through the maze walls.

If you're feeling adventurous (and you should be!) you could make a few advanced modifications to this code:

- Use the `Collision` class's `collisionSide` property to find out on which side the monster is hitting the box, as we did with our jumping character in the previous section. You can make the monster take some intelligent action, such as reversing its direction or looking for a direction which isn't blocked by a piece of the wall.
- You could of course apply some of the Artificial Intelligence (AI) techniques we looked at in chapter 8 to make the monsters actively seek out and hunt the character. This could form the basis of a very interesting and challenging game, and is not difficult to implement. You already know how!
- Instead of moving monsters, how about moving walls?

Have fun with this, there's a whole universe of game making potential this little bit of code now puts at your fingertips.

Case study: Bug Catcher

In this section of the chapter, I'll show you how I've used all these techniques together to make a simple game called Bug Catcher, which you'll find in the **`BugCatcher`** project folder. Run and jump around the game world to collect the three flying bugs, but don't step on the spider! While you play, the frog in the corner follows all your movements with its big eyes. Figure 12-4 shows what the game looks like.

Figure 12-4. Catch 3 bugs, but don't step on the spider.

I'm not providing this example as a game that you should make, or completely understand in full, but rather to demonstrate some new techniques in a real, living, breathing game. At this stage, you have all the technical skills you need to make this game yourself – and I'm sure you could make a far better one than I have. I'll list most of the source code in the pages ahead, but just think of it as a reference you can refer to if you need some help to solve some tricky design problems in your own games. The only way you can really understand a game as complex as Bug Catcher is if you write your own game like it from scratch.

Here are the new techniques we'll learn:

- How to make and use a `GameUtilities` class to reduce the amount of code clutter in the main application class.
- How to make a `Sounds` class to store and play all your game sounds.
- How to use a `switch` statement to manage different game states, such as "game running" and "game over".
- Moving objects above or below other objects by changing their stacking order.
- Making an object rotate towards another object.
- Converting an object's local coordinate system to the stage's coordinate system.
- Creating a customizable game message display system.
- Using Brownian motion to make the bugs move like real bugs.
- Using a timer to make the bugs flap their wings.

- Using simple Artificial Intelligence to make the bugs avoid the cat and the frog.

Bug Catcher contains all of these features, including the new techniques of using loops, arrays, and physics that we've already covered in this chapter.

Looking at the source code

Most of the game object classes are the same as those used in Time Bomb Panic, Monster Mayhem, and the classes that we've already been using in this chapter, so refer back to those if you have any questions about the classes in Bug Catcher. There are few exceptions, however, like the `MessageDisplay`, `Frog`, and `Bug` classes that we'll look at in detail ahead. Figure 12-05 illustrates all the classes in the game and all the files contained in the project folder.

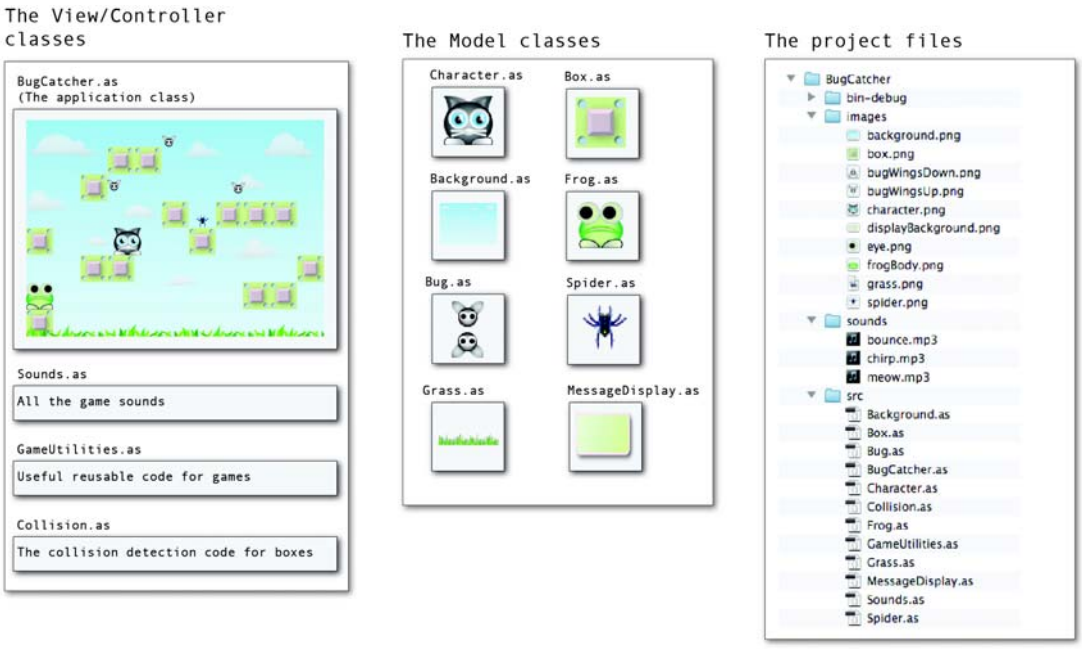


Figure 12-05. The classes and files used in BugCatcher.

Here's the entire application class, which is just for your reference, but I'll explain all the unique features of this game in the sections ahead.

```
package
{
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.geom.Point;
  import flash.ui.Keyboard;

  [SWF(width="550", height="400",
  backgroundColor="#FFFFFF", frameRate="60")]

  public class BugCatcher extends Sprite
  {
    //Properties to manage the game states
    private const RUNNING:uint = 1;
    private const OVER:uint = 2;
    private var _gameState:uint = RUNNING;
```

```
//The game objects
private var _character:Character = new Character();
private var _background:Background = new Background();
private var _grass:Grass = new Grass();
private var _frog:Frog = new Frog();
private var _spider:Spider = new Spider();

//Arrays to store the boxes and their x and y positions
private var _boxes:Array = new Array();
private var _boxPositions:Array = new Array();

//Arrays to store the bugs and their initial x and y positions
private var _bugs:Array = new Array();
private var _bugPositions:Array = new Array();

 //Create the _sounds object from the custom Sounds class
private var _sounds:Sounds = new Sounds();

//Game variables
private var _bugsCollected:uint = 0;

public function BugCatcher()
{
  //Add the game objects
  GameUtilities.addObjectToGame(_background, this);
  GameUtilities.addObjectToGame(_grass, this, 0, 370);
  GameUtilities.addObjectToGame(_character, this, 150, 300);
  GameUtilities.addObjectToGame(_frog, this, 0, 300);
  GameUtilities.addObjectToGame(_spider, this, 313, 175);

  //THE BUGS
   //Set the initial bug x and y positions
  _bugPositions
    = [
        [25, 50],
        [400, 50],
        [250, 150]
           ];

  //Make the bugs
  for(var i:int = 0; i < _bugPositions.length; i++)
  {
    //Create a bug object
    var bug:Bug = new Bug();

    //Add the bug to the stage
    GameUtilities.addObjectToGame
      (bug, this, _bugPositions[i][0], _bugPositions[i][1]);

    //Add it to the _bugs array
     //for future use
    _bugs.push(bug);
  }

  //THE BOXES
   //Set the box x and y positions
  _boxPositions
    = [
        [0, 350],
        [0, 200],
        [100, 100],
        [100, 250],
        [150, 50],
        [150, 250],
```

```
        [200, 50],
        [300, 200],
        [250, 150],
        [350, 150],
        [400, 150],
        [400, 300],
        [450, 150],
        [450, 300],
        [500, 250]
    ];

  //Make the boxes
  for(var j:int = 0; j < _boxPositions.length; j++)
  {
    //Create a box object
    var box:Box = new Box();

     //Add the box to the stage
    GameUtilities.addObjectToGame
      (box, this, _boxPositions[j][0], _boxPositions[j][1]);

    //Add it to the _boxes array
     //for future use
    _boxes.push(box);
  }
  //Add the event listeners
  stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownHandler);
  stage.addEventListener(KeyboardEvent.KEY_UP, keyUpHandler);
  stage.addEventListener(Event.ENTER_FRAME, enterFrameHandler);

  //View the number of objects on the stage
  //trace(numChildren);
  //View the stacking order of objects on the stage
  /*
  for (var k:int = 0;  k < numChildren;  k++)
  {
    trace(k + ".  " + getChildAt(k));
  }
  */

  //Move the grass to the top of the stack
  setChildIndex(_grass, numChildren - 1);

  //Swap the display list positions of the
  //character and grass objects
  //swapChildren(_character, _grass);

  //View the updated stacking order
  /*
  for (var l:int = 0;  l < numChildren;  l++)
  {
    trace(l + ".  " + getChildAt(l));
  }
  */

}
public function enterFrameHandler(event:Event):void
{
  switch(_gameState)
  {
    case RUNNING:
      runGame();
      break;
    case OVER:
```

```
      gameOver();
      break;
  }
}
public function runGame():void
{
   //Move the bugs


  //Create a temporary local variable
  //to easily access the current bug in the loops
  var bug:Bug;

  for(var i:int = 0; i < _bugs.length; i++)
  {

    bug = _bugs[i];

    if(bug.visible)
    {
      moveBug(bug);
    }
  }
  moveCharacter();

  //Make the frog's eyes follow the character
  makeFrogLook();

  //Check collisions between the character and the bugs
  for(var j:int = 0; j < _bugs.length; j++)
  {
    bug = _bugs[j];
    if(_character.hitTestObject(bug)
    && bug.visible)
    {
      bug.visible = false;
      _sounds.chirpChannel = _sounds.chirp.play();
      _bugsCollected++;
      if(_bugsCollected == 3)
      {
        _gameState = OVER;
      }
    }
  }
  //Check collisions between the character and the spider
  if(_character.hitTestObject(_spider))
  {
    _sounds.meowChannel = _sounds.meow.play();
    _gameState = OVER;
  }
}
public function gameOver():void
{
  //Create a String variable to store the
  //game over message
  var message:String = "";

  //Figure out if the player won or lost
  if(_bugsCollected == 3)
  {
    message = "You collected the bugs!";
         }
  else
  {
```

```
    message = "You stepped on a spider!";
  }

  //Display the game over message in a MessageDisplay window
  var gameOverMessage:MessageDisplay
    = new MessageDisplay(message, 1000);
  stage.addChild(gameOverMessage);
  gameOverMessage.x = 180;
  gameOverMessage.y = 130;

  //Dim the character and remove the enterFrameHandler
  _character.alpha = 0.5;
  stage.removeEventListener(Event.ENTER_FRAME, enterFrameHandler);
}
private function makeFrogLook():void
{
  //Convert points from local to global coordinates
  //Right eye
  var frogsRightEye:Point
    = new Point(_frog.rightEye.x, _frog.rightEye.y);
  var frogsRightEye_X:Number
    = _frog.localToGlobal(frogsRightEye).x;
  var frogsRightEye_Y:Number
    = _frog.localToGlobal(frogsRightEye).y;
  //Left eye
  var frogsLeftEye:Point
    = new Point(_frog.leftEye.x, _frog.leftEye.y);
  var frogsLeftEye_X:Number
    = _frog.localToGlobal(frogsLeftEye).x;
  var frogsLeftEye_Y:Number
    = _frog.localToGlobal(frogsLeftEye).y;

  //Rotate eyes
  _frog.rightEye.rotation
    = Math.atan2
      (
        frogsRightEye_Y - _character.y,
        frogsRightEye_X - _character.x
      )
      * (180/Math.PI);

    _frog.leftEye.rotation
      = Math.atan2
        (
          frogsLeftEye_Y - _character.y,
          frogsLeftEye_X  - _character.x
        )
        * (180/Math.PI);
}
private function moveCharacter():void
{
  //Apply acceleration
  _character.vx += _character.accelerationX;

  //Apply friction
  _character.vx *= _character.friction;

  //Apply gravity
  _character.vy += _character.gravity;

  //Limit the speed, except when the character
  //is moving upwards
  if (_character.vx > _character.speedLimit)
  {
```

```
  _character.vx = _character.speedLimit;
}
if (_character.vx < -_character.speedLimit)
{
  _character.vx = -_character.speedLimit;
}
if (_character.vy > _character.speedLimit * 2)
{
  _character.vy = _character.speedLimit * 2;
}

//Force the velocity to zero
//after it falls below 0.1
if (Math.abs(_character.vx) < 0.1)
{
  _character.vx = 0;
}
if (Math.abs(_character.vy) < 0.1)
{
  _character.vy = 0;
}

//Move the character
_character.x += _character.vx;
_character.y += _character.vy;

//Check stage boundaries
if (_character.x < 0)
{
  _character.vx = 0;
  _character.x = 0;
}
if (_character.y < 0)
{
  _character.vy = 0;
  _character.y = 0;
}
if (_character.x + _character.width > stage.stageWidth)
{
  _character.vx = 0;
  _character.x = stage.stageWidth - _character.width;
}
if (_character.y + _character.height > stage.stageHeight)
{
  _character.vy = 0;
  _character.y = stage.stageHeight - _character.height;
  _character.isOnGround = true;
}

for(var i:int = 0; i < _boxes.length; i++)
{
  Collision.block(_character, _boxes[i]);

  if(Collision.collisionSide == "Bottom")
  {
    //Tell the character that it's on the
    //ground if it's standing on top of
    //a platform
    _character.isOnGround = true;

    //Neutralize gravity by applying its
    //exact opposite force to the character's vy
    _character.vy = -_character.gravity;
  }
```

```
    else if(Collision.collisionSide == "Top")
    {
      _character.vy = 0;
    }
    else if(Collision.collisionSide == "Right"
    || Collision.collisionSide == "Left")
    {
      _character.vx = 0;
    }
   }
}
private function moveBug(bug:Bug):void
{
  //Add Brownian motion to the velocities
  bug.vx += (Math.random() * 0.2 - 0.1) * 40;
  bug.vy += (Math.random() * 0.2 - 0.1) * 40;

  //Add some friction
  bug.vx *= 0.96;
  bug.vy *= 0.96;

  //Move the bug
  bug.x += bug.vx;
  bug.y += bug.vy;

  //Stage Boundaries
  if (bug.x > stage.stageWidth - bug.width)
  {
    bug.x = stage.stageWidth - bug.width;

    //Reverse (bounce) bug's velocity when it hits the stage edges
    bug.vx *= -1;
  }
  if (bug.x < 0)
  {
    bug.x = 0;
    bug.vx *= -1;
  }
  //Keep the bug above the grass
  if (bug.y > stage.stageHeight - 35)
  {
          bug.y = stage.stageHeight - 35;
          bug.vy *= -1;
  }
  if (bug.y < 0)
  {
    bug.y = 0;
    bug.vy *= -1;
  }

  //Apply collision detection with the platforms
  for(var j:int = 0; j < _boxes.length; j++)
  {
    Collision.block(bug, _boxes[j]);
  }

  //ARTIFICIAL INTELLIGENCE
  //Make the bugs avoid the frog
  if ((Math.abs(bug.x - _frog.x) < 100))
  {
    if (Math.abs(bug.y - _frog.y) < 100)
    {
      bug.vx *= - 1;
      bug.vy *= - 1;
```

```
      trace(bug.name + ": Frog!");
    }
  }

  //Calculate the distance vector
  //between the center of the
  //bug and the character
  var vx:Number
    = (bug.x + (bug.width / 2))
    - (_character.x + (_character.width / 2));

  var vy:Number
    = (bug.y + (bug.height / 2))
    - (_character.y + (_character.height / 2));


  if (Math.abs(vx) < 100 && Math.abs(vy) < 100)
  {
    //If the player is moving...
    bug.vy += _character.vy + ((Math.random() * 0.2 - 0.1) * 70);
    bug.vx += _character.vx + ((Math.random() * 0.2 - 0.1) * 70);
    trace(bug.name + ": Cat!");

    //If the player is sitting still...
    if ((Math.abs(_character.vy) < 1)
    && (Math.abs(_character.vx) < 1))
    {
      bug.y += -bug.vy;
      bug.x += -bug.vx;
      bug.vy *= -1;
      bug.vx *= -1;
    }
  }
}
public function keyDownHandler(event:KeyboardEvent):void
{
  if (event.keyCode == Keyboard.LEFT)
  {
    _character.accelerationX = -0.2;
  }
  else if (event.keyCode == Keyboard.RIGHT)
  {
    _character.accelerationX = 0.2;
  }
  else if (event.keyCode == Keyboard.UP
  || event.keyCode == Keyboard.SPACE)
  {
    if(_character.isOnGround)
    {
      _sounds.bounceChannel = _sounds.bounce.play();
      _character.vy += _character.jumpForce;
      _character.gravity = 0.3;
      _character.isOnGround = false;
    }
  }
}
public function keyUpHandler(event:KeyboardEvent):void
{
  if (event.keyCode == Keyboard.LEFT
  || event.keyCode == Keyboard.RIGHT)
  {
    _character.accelerationX = 0;
  }
}
```

```
  }
}
```

Managing game states

Games usually have a number of different **states**:

- A “waiting” state, which might be when the game displays its title or instructions, while waiting for the player to press a play game button.
- A “running” state, which is the actual game.
- An “over” state, which is what happens when the player wins or loses.

It’s useful to think about game states like this, because if your game knows what state it’s currently in, it can automatically switch how it’s behaving based on what it’s doing.

Bug Catcher only has two states: a running state and an over state. The game represents these two states as constants, which are declared in the class definition. It also creates a variable that stores the current game state. The game state is set to `RUNNING` when the game first starts.

```
private const RUNNING:uint = 1;
private const OVER:uint = 2;
private var _gameState:uint = RUNNING;
```

Bug Catcher then does something we haven’t yet seen in this book so far. The entire `enterFrameHandler` consists of only a switch statement, which calls a different method based on the game’s current state. If the current state is `RUNNING`, it calls the `runGame` method. If the state is `OVER`, it calls the `gameOver` method. Here’s the `enterFrameHandler` and the `switch` statement that does this work:

```
public function enterFrameHandler(event:Event):void
{
  switch(_gameState)
  {
    case RUNNING:
      runGame();
      break;

    case OVER:
      gameOver();
      break;
        }
}
```

The `runGame` method contains all the frame-by-frame logic and actions that the game needs to run. It’s all the stuff that’s been in our `enterFrameHandler` in previous examples. The `gameOver` method contains all the actions we want the game to perform when the game is finished. We’ll look at this soon but first, what is that `switch` statement all about and how does it work?

Using a switch statement

Switch statements are block statements that can be an alternative to using an `if` statement in many cases. It performs the same job as an `if` statement, by selecting one of many possible conditions. Whether you use an `if` statement or a `switch` statement is often a matter of personal choice.

The `switch` keyword accepts one argument, which in this example is `_gameState`:

```
switch(_gameState)
```

In this game, `_gameState` can have three possible values: `RUNNING`, or `OVER`. In a `switch` statement, the value is known as a **case**, and each case can have a different outcome. Let’s look at the first case, `RUNNING`:

```
case RUNNING:
  runGame();
  break;
```

Any directives that come after the colon are the actions the program should take. In this case, the `runGame` method should be called.

```
runGame();
```

And remember, because this `switch` statement is in the `enterFrameHandler`, the `runGame` Method is called every frame while the state continues to be `RUNNING`.

The last thing the case does is run a `break` directive:

```
break;
```

This stops the `switch` statement from continuing. It's found what it's looking for, so it doesn't need to check any of the other cases. If `_gameState` isn't `RUNNING`, however, the `switch` statement continues and checks the next case.

The next case does exactly the same thing, but checks for a different condition:

```
case OVER:
  gameOver();
  break;
```

If the `_gameState` happens to be `OVER`, then the `gameOver` method is run.

Each case in a `switch` statement can include as many directives as you need, and you can also use as many cases as you want to. Switch statements also have a special, optional case called `default`, which will always run if none of the other cases happen to be true. Here's a simple example of a `switch` statement that includes a default case.

```
var sound:String = "meow"

switch(sound)
{
  case "woof":
    //directives...
    break;

  case "meow":
        //directives...
        break;

  case "chirp":
    //directives...
    break;

  default:
    //directives to run if
    //the sound variable isn't
    //"woof", "meow" or "chirp"
}
```

In this example, "meow" would be chosen, and `break` would cause the `switch` statement to stop at that point. If the `sound` variable didn't have a value of "woof", "meow" or "chirp", the directives in the optional default case would run.

As you can see, this `switch` statement is an alternative way of writing an `if` statement that looks like this:

```
var sound:String = "meow"

if(sound == "woof")
{
  //directives...
}
else if(sound == "meow")
{
  //directives...
}
else if(sound == "chirp")
{
```

```
  //directives
}
else
{
  //directives to run if
  //the sound variable isn't
  //"woof", "meow" or "chirp"
}
```

If there is no functional benefit to using a `switch` statement, why bother using one? It's purely a stylistic difference—`switch` statements are a little easier to read. They clearly stand out in your code, and you don't need to navigate through a tangle of disorienting curly braces to clearly see which conditions result in which outcomes. If you have more than two conditions that you're checking for, try to implement a `switch` statement.

Switching to the OVER state

Because the `switch` statement is running in the `enterFrameHandler`, it's listening for a change of state each frame. As soon as the state switches from `RUNNING` to `OVER`, it will call the `gameOver` method.

There are two things in the game that change the game state to `OVER`. The first is if the character collects three bugs. A loop checks for collisions between the character and the bugs in the `_bugs` array. If it finds a collision, it sets the bug's `visible` property to `false` and updates the `_bugsCollected` variable by one. It also plays a chirp sound (poor bug… more on how that works soon!). And if it finds that `_bugsCollected` equals 3, it changes `_gameState` to `OVER`. Here's the code that does this, with the change of state highlighted in bold

```
//Check collisions between the character and the bugs
var bug:Bug;
for(var i:int = 0; i < _bugs.length; i++)
{
  bug = _bugs[i];
  if(_character.hitTestObject(bug)
        && bug.visible)
   {
    bug.visible = false;
    _sounds.chirpChannel = _sounds.chirp.play();
    _bugsCollected++;
    if(_bugsCollected == 3)
    {
      _gameState = OVER;
    }
  }
}
```

The `switch` statement in the `enterFrameHandler` will detect this as soon as the next frames swings by, and immediately call the `gameOver` method. The `gameOver` method creates a custom message based on the outcome, creates a MessageDisplay object to display the result, dims the character on the stage and removes the `enterFrameHandler`. Here's the code that does this.

```
public function gameOver():void
{
        //Create a String variable to store the
        //game over message
        var message:String = "";

        //Figure out if the player won or lost
        if(_bugsCollected == 3)
        {
                message = "You collected the bugs!";
        }
        else
        {
                message = "You stepped on a spider!";
        }
```

```
        //Display the game over message in a MessageDisplay window
        var gameOverMessage = new MessageDisplay(message, 1000);
        stage.addChild(gameOverMessage);
        gameOverMessage.x = 180;
        gameOverMessage.y = 130;

        //Dim the character and remove the enterFrameHandler
        _character.alpha = 0.5;
        stage.removeEventListener(Event.ENTER_FRAME, enterFrameHandler);
}
```

This is very different from the way the other games in the book have ended, and I've done this intentionally so that you can see there are many different techniques you can use to end games. What's unique about this system is that you can create a custom game over message and then display it in a `MessageDisplay` object that appears over the stage after a one second delay, as shown in Figure 12-06.

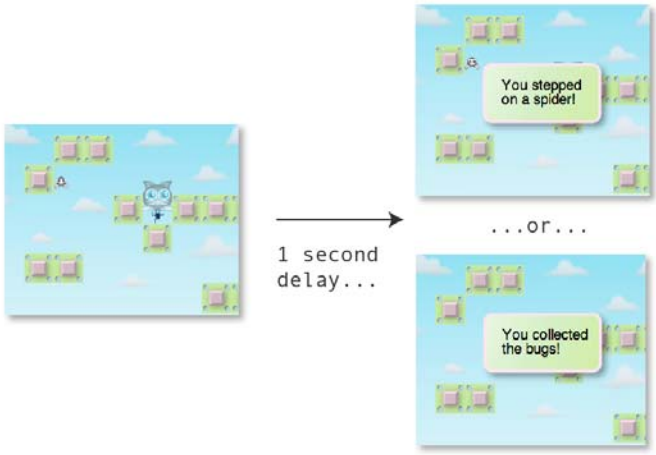


Figure 12-6. There's a one second delay at the end of the game before the MessageDisplay object appears.

To make this work, you first need some kind of text to display. Bug Catcher chooses one of two messages based on whether the character won or lost.

```
var message:String = "";
if(_bugsCollected == 3)
{
        message = "You collected the bugs!";
}
else
{
        message = "You stepped on a spider!";
}
```

You can then use this message to create a `MessageDisplay` object, like this:

```
var gameOverMessage = new MessageDisplay(message, 1000);
```

The first argument is the message, and the second is the number of milliseconds you want to wait before the message is displayed. In this example, the message will display after 1 second. (If you leave this second argument out, the message window will appear immediately.)

Here's the custom `MessageDisplay` class in the project source files that does this work. It sets the delay on the timer and displays whatever message is sent to it through its parameters in the class constructor

method. It's invisible when it's first created, and its internal timer makes it visible when the timer's displayMessageHandler is called.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;
  import flash.text.*;

  public class MessageDisplay extends Sprite
  {
    //Embed the background image
    [Embed(source="../images/displayBackground.png")]
    private var DisplayBackgroundImage:Class;

    //Private properties
    private var _displayBackgroundImage:DisplayObject
      = new DisplayBackgroundImage();
    private var _displayBackground:Sprite = new Sprite();
    private var _message:String = "";
    private var _timer:Timer;
    private var _format:TextFormat = new TextFormat();
    private var _output:TextField = new TextField();

    public function MessageDisplay(message:String, time:int = 0)
    {
      _message = message;

      _displayBackground.addChild(_displayBackgroundImage);
      this.addChild(_displayBackground);
      this.visible = false;

      //Set the text format object
      _format.font = "Helvetica";
      _format.size = 24;
      _format.color = 0x000000;
      _format.align = TextFormatAlign.LEFT;

      //Configure the _output text field
      _output.defaultTextFormat = _format;
      _output.width = 150;
      _output.height = 70;
      _output.border = false;
      _output.wordWrap = true;
      _output.text = "";

      //Display and position the _output text field
      this.addChild(_output);
      _output.x = 30;
      _output.y = 25;

      //The timer
      _timer = new Timer(time);
      _timer.addEventListener
        (TimerEvent.TIMER, displayMessageHandler);
      _timer.start();
    }

    //Private methods
    private function displayMessageHandler(event:TimerEvent):void
    {
      _output.text = _message;
      this.visible = true;
```

```
      _timer.removeEventListener
        (TimerEvent.TIMER, displayMessageHandler)
    }
  }
}
```

What's new here is that the `gameOverMessage` object is created from this class using custom arguments sent to it from the application class. That means that whenever you create a new MessageDisplay object, you can send it custom text and a custom delay time. That makes it very flexible, and you can use it over and over again in different contexts to display different messages. Let's review exactly how this works.

When the application class creates a new MessageDisplay object, it sends it two arguments: the message and the delay time:

```
var gameOverMessage = new MessageDisplay(message, 1000);
```

The MessageDisplay class's constructor method has two parameters that match the arguments that it's expecting to receive. The first is a String called `message`, and the second is an integer called `time`:

```
public function MessageDisplay(message:String, time:int = 0)
```

Now whenever the `MessageDisplay` class uses variables called `message` or `time`, they contain exactly the same values that were sent to it by the application class when it created the new object.

The message and time variables are only local to the constructor method, however, you can't use them anywhere else in the class. That means that if you want to use them with another method in the class, you need to first copy them into an instance variable. That's why the code first copies the `message` variable it received as a parameter in the constructor method to an instance variable called `_message`.

```
//An instance variable to keep the message persistent
private var _message:String = "";

public function MessageDisplay(message:String, time:int = 0)
{
  //Copy the value of message into the instance variable
  //so it can be used elsewhere in the class:
  _message = message;
```

`_message` can now be used in any other method in the class, and it will contain the same value that was passed to the class when it was created.

The time value that's passed to the class in the constructor is also used to set the delay on the `_timer`, like this:

```
public function MessageDisplay(message:String, time:int = 0)
{
  //...
  _timer = new Timer(time);
        _timer.addEventListener(TimerEvent.TIMER, displayMessageHandler);
  _timer.start();
}
```

That's why whatever time in milliseconds you pass to the object when you create it is used as the delay value.

Figure 12-07 illustrates how this whole system works.

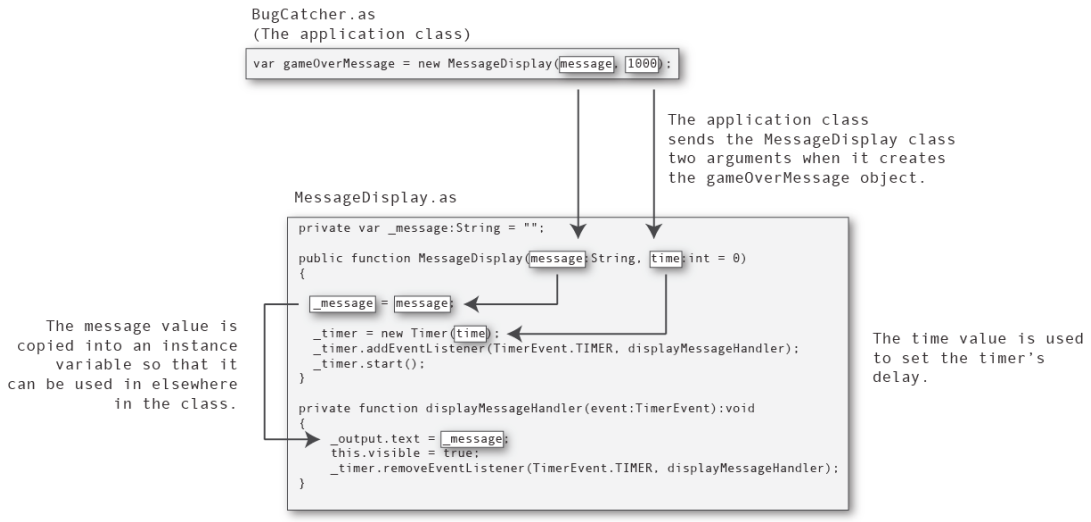


Figure 12-7. Customize an object when you create it by sending information to its class constructor method.

> *An important detail to note is that you can give parameters default values in the constructor method. MessageDisplay's constructor gives time a default value of zero:*
>
> ***`public function MessageDisplay(message:String, time:int = 0)`***
>
> *This means that if you don't provide any value for time, it will be automatically set to zero. That's why if you leave the second argument out when you create a message display object, there won't be any delay when the message is displayed on the stage.*
>
> *Assigning parameters default values like this is very good programming practice because it means that objects you create from your classes will still work fine even if you don't supply them with all the extra information they might be able to use.*
>
> *If you don't supply a default value for a parameter, and you forget to supply an argument when you create an object, you'll see this error message:*
>
> *1136: Incorrect number of arguments. Expected 1.*

An advantage to this whole system is that you can change the game over message to whatever you like during the course of the game, and you don't have to plan this in advance. You could tell the player the score, or provide any other data about how the game was played. Take a bit of time to understand how this system works, because these are important programming concepts that you'll be able to find lots of use for in your games.

Using a game utilities class

Often you'll find while you're making games that you reuse the same methods over and over again to perform mundane tasks. It's a good idea to collect all these methods together and dump them all into a game utilities class. The little `addGameObjectToStage` method we've been using to add objects to the stage comes to mind, as does the code that checks for stage boundaries or to handle keyboard interactivity. If you

put all these methods into their own class, they won't clutter up your main game code, and you can easily reuse them by just copying the class into another game project folder.

To get you started, Bug Catcher uses a class called `GameUtilites` that includes a method for adding objects to the game, and a simple method that can be used to rotate an object towards another object. Here's the entire `GameUtilities` class from the src folder in the `BugCatcher` project directory.

```
package
{
  import flash.display.Sprite;

  public class GameUtilities
  {
    public function GameUtilities()
    {
    }

    //Add an object to the game
    static public function addObjectToGame
      (
        gameObject:Sprite,
        gameWorld:Object,
        xPos:int = 0,
        yPos:int = 0
      ):void
    {
      gameWorld.addChild(gameObject);
      gameObject.x = xPos;
      gameObject.y = yPos;
    }

    //Rotate an object towards another object
    static public function rotateToObject
           (objectOne:Sprite, objectTwo:Sprite):void
    {
      objectOne.rotation
        = Math.atan2
        (
           objectOne.y - objectTwo.y,
           objectOne.x - objectTwo.x
        )
        * (180/Math.PI);
    }
  }
}
```

I've used the `addObjectToGame` method to add all the game objects in Bug Catcher. Here's the game objects that are created in the `BugCatcher` constructor method:

```
GameUtilities.addObjectToGame(_grass, this, 0, 370);
GameUtilities.addObjectToGame(_background, this);
GameUtilities.addObjectToGame(_character, this, 150, 300);
GameUtilities.addObjectToGame(_frog, this, 0, 300);
GameUtilities.addObjectToGame(_spider, this, 313, 175);
```

Because it's been declared as a static method, you need to precede the name of the method with the name of the class, `GameUtilities`. (Refer back to chapter 6 if you need a reminder about what static methods are and how they work.) This is the same as the `block` method from the `Collision` class, which is also static.

The second argument, "this", refers to the class that you want to add the game object to. Because `BugCatcher` is the application class, `this` refers to the stage, and the objects will appear on the main stage. (Remember that the application class is the only class with direct access to the stage). If you were adding these objects in another class, such as LevelOne, `this` would refer to that class. The method is general enough that you can use it in any context to display objects. (In addition, if you leave out the last two arguments, the x and y positions, the objects will be set to a default x and y positions of 0.)

This method is also used in the loops that add the boxes and bugs to the stage. Here's the code that creates and adds the bugs, with the `addObjectToGame` method highlighted in bold.

```
//Set the initial bug x and y positions
_bugPositions
  = [
      [25, 50],
      [400, 50],
      [250, 150]
    ];

//Make the bugs
for(var i:int = 0; i < _bugPositions.length; i++)
{
  //Create a bug object
  var bug:Bug = new Bug();

  //Add the bug to the stage
  GameUtilities.addObjectToGame
    (bug, this, _bugPositions[i][0], _bugPositions[i][1]);

  //Add it to the _bugs array
  //for future use
  _bugs.push(bug);
}
```

To keep the `GameUtilities` class useful, don't add any code that's just specific to one game. If you do, it will make it difficult to reuse the class with other game projects, or it could become full of code you never use. Now that I've got you started, start adding methods that you often use with your own games.

A class to keep your sounds organized

I mentioned in the last section that if you declare a method as static, you have to precede the name of the method with the name of the class, like this:

```
ClassName.staticMethod
```

You don't have to create an instance of the class using the `new` keyword.

However, sometimes you have to make an instance a class before you can use it. This is especially true if you need to pass the class some custom parameters. We just saw this in the code that created a new MessageDisplay object:

```
var gameOverMessage = new MessageDisplay(message, 1000);
```

In this case you don't access the class directly, you instead use the object you made from it: `gameOverMessage`. If the `MessageDisplay` class contains any public methods, you can access them like this:

```
gameOverMessage.publicMethod
gameOverMessage.publicProperty
```

We've been interacting with most of our game objects like this.

You also need to make an instance of a class if that class embeds any sounds or images. That's because the act of creating the object with the `new` keyword is what actively makes the class embed the sounds or images that you're using in the object. That means that if you have a class that contains embedded sounds, you have to create it in the application class's constructor method like this:

```
private var _sounds:Sounds = new Sounds();
```

That's exactly what happens in Bug Catcher. A `_sounds` object is created with the above code that stores all the sounds in the game. This reduces the clutter of all the code needed to embed the sounds, and keeps the sounds all in one place. You can then reuse the sounds in other games just by dropping your `Sounds` class into another game project folder.

Here's the `Sounds` class that you'll find in Bug Catcher's **`src`** folder. It embeds three sounds. It also makes the `sound` and `soundChannel` objects of each sound public so that they're easy to access and control from the application class.

```
package
{

  import flash.media.Sound;
  import flash.media.SoundChannel;
  import flash.media.SoundTransform;

  public class Sounds
  {
    //Embed the sounds
    [Embed(source="../sounds/bounce.mp3")]
    private var Bounce:Class;
    [Embed(source="../sounds/chirp.mp3")]
    private var Chirp:Class;
    [Embed(source="../sounds/meow.mp3")]
    private var Meow:Class;


    //Create the Sound and Sound channel objects
    //Bounce sound
    public var bounce:Sound = new Bounce();
    public var bounceChannel:SoundChannel = new SoundChannel();

    //Chirp sound
    public var chirp:Sound = new Chirp();
    public var chirpChannel:SoundChannel = new SoundChannel();

    //Meow sound
    public var meow:Sound = new Meow();
    public var meowChannel:SoundChannel = new SoundChannel();

    public function Sounds()
    {
    }
  }
}
```

We can use these sounds anywhere in the game through the `_sound` object the application class created. To make the bugs chirp, use this code:

```
_sounds.chirpChannel = _sounds.chirp.play();
```

This happens when one of the bugs is caught by the cat. You can make the cat meow when it steps on a spider in the `runGame` method, like this:

```
if(_character.hitTestObject(_spider))
{
        _sounds.meowChannel = _sounds.meow.play();
        _gameState = OVER;
}
```

This code also changes the `_gameState` to `OVER`, which ends the game.

Making the bugs flap their wings

When you play the game, you'll notice that the three bugs rapidly flap their wings as they fly across the stage, as shown in Figure 12-26.

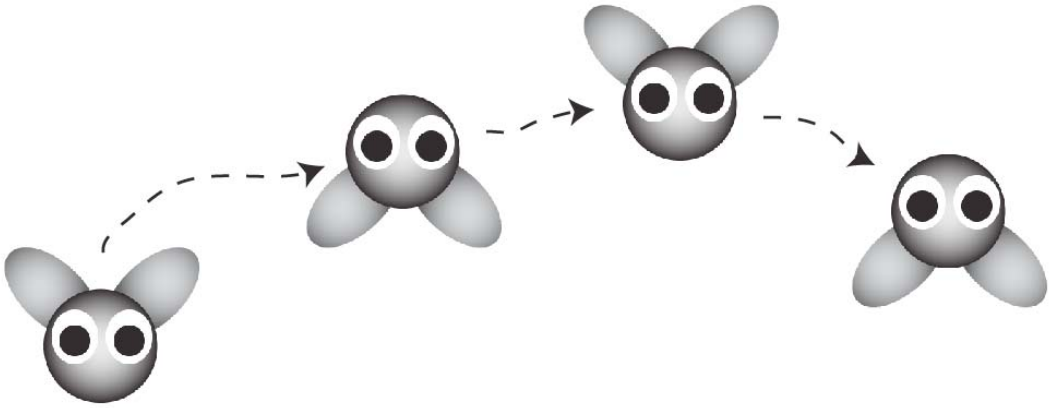


Figure 12-08. Use a timer to rapidly flip back and forth the between two images to make the bugs flap their wings.

This is done by switching between two images 20 times per second. This first image shows the bug with its wings up, and the second with its wings down. It's all handled inside the `Bug` class, and is a good example of how to use a timer to create a simple animated effect. Here's the `Bug` class that achieves this effect.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;

  public class Bug extends Sprite
  {
    //Embed the bug images
    [Embed(source="../images/bugWingsUp.png")]
    private var WingsUpImage:Class;
    [Embed(source="../images/bugWingsDown.png")]
    private var WingsDownImage:Class;

    //Private properties
    private var _wingsUp:DisplayObject = new WingsUpImage();
    private var _wingsDown:DisplayObject = new WingsDownImage();
    private var _timer:Timer;

    //Public properties
    public var vx:int = 0;
    public var vy:int = 0;

    public function Bug()
    {
      this.addChild(_wingsUp);
      this.addChild(_wingsDown);
      _wingsDown.visible = false;

      //The wing timer
      _timer = new Timer(50);
      _timer.addEventListener(TimerEvent.TIMER, flapWingHandler);
      _timer.start();
    }

    //Private methods
    private function flapWingHandler(event:TimerEvent):void
    {
```

```
      _wingsDown.visible = !_wingsDown.visible;
      _wingsUp.visible = !_wingsUp.visible;
    }
  }
}
```

Both images are added to the class when it loads, but one of them, the `wingsDown` image, is made invisible
`_wingsDown.visible = false;`

The timer sets the `flapWingHandler` to fire every 50 milliseconds

```
_timer = new Timer(50);
_timer.addEventListener(TimerEvent.TIMER, flapWingHandler);
_timer.start();
```

The `flapWingHandler` has a very simple job. It just reverses the current visibility of the images, employing the same trick we used to make the toggle button in chapter 3.

```
private function flapWingHandler(event:TimerEvent):void
{
  _wingsDown.visible = !_wingsDown.visible;
  _wingsUp.visible = !_wingsUp.visible;
}
```

The *not* operator (the exclamation mark) forces the `visible` properties of the images to be the opposite of what they currently are. One of the images was invisible and the other was visible when the class was initialized, but the `flapWingHandler` flips the visibility of these images. Run this at 20 times per seconds and you have a flapping wing effect.

The `Bug` class just uses two images for this effect, but there's no reason why you couldn't create more complex animations by displaying a whole series of slightly different images in quick succession. You could use the timer's `currentCount` property to determine the correct sequence to display the images.

Finding the global x and y position of a sub-object

The Bug Catcher game includes a frog character that sits on the platform in the bottom-left corner of the stage. If you look at the frog closely while you're playing the game, you'll notice that its eyes follow the cat wherever it goes, as shown in Figure 12-9.

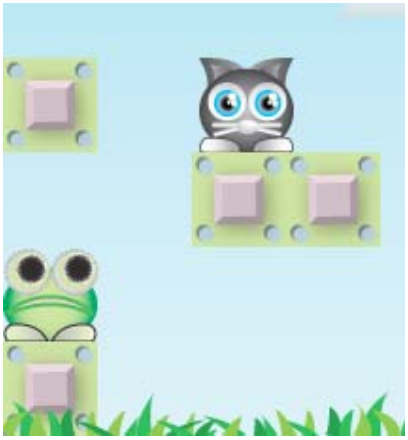


Figure 12-9. The frog's eyes follow the cat wherever it goes.

To implement this little trick, you have to create the eyes as sub-objects inside the main frog object. We've seen this done in many examples in this book already. The frog is composed of two images: an image of the frog's body, and 2 copies of a single eye image, as shown in Figure 12-10.

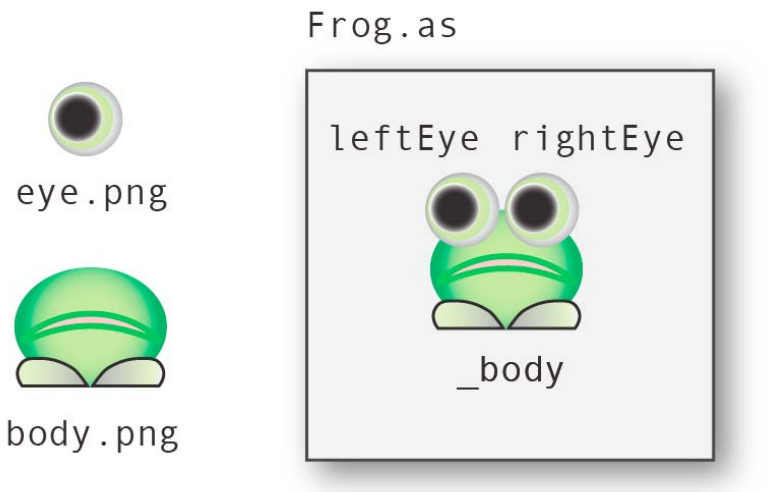


Figure 12-10. The frog is made up of three separate sub-objects made from two PNG images.

The eyes are correctly positioned inside the Frog class, and given the names `rightEye` and `leftEye`. The eye objects are declared as public so that the application class can access them to rotate them. Here's the `Frog` class that embeds the images, creates the eye sub-objects and correctly positions them on the frog's body.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;

  public class Frog extends Sprite
  {
    //Embed the body and eye images
    [Embed(source="../images/frogBody.png")]
    private var BodyImage:Class;
    [Embed(source="../images/eye.png")]
    private var EyeImage:Class;

    //Private properties
    //The frog's body
    private var _bodyImage:DisplayObject = new BodyImage();
    private var _body:Sprite = new Sprite();
    //The frog's eyes
    private var _leftEyeImage:DisplayObject = new EyeImage();
    private var _rightEyeImage:DisplayObject = new EyeImage();

    //Public properties
    public var leftEye:Sprite = new Sprite();
    public var rightEye:Sprite = new Sprite();

    public function Frog()
    {
      //Make the body
      _body.addChild(_bodyImage);
      this.addChild(_body);
```

```
      //Make the eyes
      leftEye.addChild(_leftEyeImage);
      rightEye.addChild(_rightEyeImage);

      //Center the eyes inside their containing Sprites
      //so that they will rotate around the center
      _leftEyeImage.x = -(_leftEyeImage.width / 2);
      _leftEyeImage.y = -(_leftEyeImage.height / 2);
      _rightEyeImage.x = -(_rightEyeImage.width / 2);
      _rightEyeImage.y = -(_rightEyeImage.height / 2);

      this.addChild(leftEye);
      this.addChild(rightEye);

      //Position the right eye
      rightEye.x = 38;
      rightEye.y = 13;

      //Position the left eye
      leftEye.x = 13;
      leftEye.y = 13;
    }
  }
}
```

When the eyes rotate, we want them to rotate around their center axis, not their top left corners. This means that we have to center them within their containing sprites. This is the same technique we used to center the star projectile in Monster Mayhem so that it would spin on its center axis.

```
_leftEyeImage.x = -(_leftEyeImage.width / 2);
_leftEyeImage.y = -(_leftEyeImage.height / 2);
_rightEyeImage.x = -(_rightEyeImage.width / 2);
_rightEyeImage.y = -(_rightEyeImage.height / 2);
```

This is how the Frog class is built and results in what you can see on the stage. But our next task is that we have to find exact x and y position of those `leftEye` and `rightEye` sub-objects, and then rotate them so that they point in the correct direction.

There's one big problem, however. AS3.0 interprets the x and y positions of sub-objects according to the sub-object's **local coordinates**. Those are its coordinates inside the object that it occupies, and you've seen how you can adjust those to precisely position sub-objects inside their container objects. But AS3.0 doesn't know what the x and y coordinates are that they occupy on the stage. The stage's coordinates are known as the **global coordinates**. Before you can make the frog's eyes rotate correctly, you need to know what their global x and y positions are.

AS3.0 has a system for converting local coordinates to global coordinates. As a game designer, this is something you'll find yourself needing to do all the time. Strangely, however, AS3.0 doesn't make this easy, and converting local to global coordinates is something that has stumped even very experienced developers new to AS3.0. To make matters worse, Adobe's documentation on this is far from clear, especially in the practical usage that you need to put it to in everyday situations such as these.

You won't like this system; it's cumbersome and convoluted. But it's the only way of converting coordinates. It works, and you have to learn to live with it. Don't worry, though; just bookmark this page and refer to these directions every time you need to do this. It's actually not so bad once you get used to it!

Here's the process you need to follow to convert local coordinates to global coordinates (don't let this long list scare you; you'll soon see that it's not quite as hard to implement as it might seem):

1. Import AS3.0's Point class with the import directive import `flash.geom.Point;`. This import directive should be part of the class definition, along with all the other import directives.

2. Create a new Point object to store the x and y positions of the sub-object whose local coordinates you want to convert to stage coordinates. In this example, the line of code might look like this:

```
var frogsRightEye:Point
  = new Point(_frog.rightEye.x, _frog.rightEye.y);
```

Point objects contain two built-in properties: x and y. The x and y coordinates of the object that you specify in the arguments when you create the Point object are copied into the Point object's own x and y properties. (Don't worry if this sounds confusing! We'll look at this in more detail in a moment.)

3. 3. Use AS3.0's built-in `localToGlobal` method to convert the new Point object's local coordinates to global coordinates. `localToGlobal` is a method of the Sprite class, so it needs to be called by a Sprite object. Usually, the object that calls it is the parent of the sub-object. In this game, the parent is the `_frog` object. So to convert the x coordinate, you can use some code that looks like this:

```
_frog.localToGlobal(frogsRightEye).x;
```

4. 4. You need to store this new global coordinate in yet another variable so that you can put it to some practical use in the program. You can convert the Point object's x coordinate and store it in a new variable with a single line of code that looks like this:

```
var frogsRightEye_X:Number
  = _frog.localToGlobal(frogsRightEye).x;
```

5. After all that trouble, you have a variable called `frogsRightEye_X`. It contains the global coordinate of the sub-object's x position. That's the variable you use if you want to refer to the right eye's global x position.

6. Hey, not so fast! You're not done yet! You've only converted the x position's coordinate. Repeat steps 3 to 5 to convert the y position as well. Rinse thoroughly and blow dry.

Apart from the fact that I do believe a letter is in order to Adobe before they let AS4.0 pass without simplifying this, let's not let this get you down. You can do it!

Here's how this code is implemented in Bug Catcher to find the global coordinates of the frog's right and left eyes.

```
//Right eye
var frogsRightEye:Point
  = new Point(_frog.rightEye.x, _frog.rightEye.y);
var frogsRightEye_X:Number
  = _frog.localToGlobal(frogsRightEye).x;
var frogsRightEye_Y:Number
  = _frog.localToGlobal(frogsRightEye).y;

//Left eye
var frogsLeftEye:Point
  = new Point(_frog.leftEye.x, _frog.leftEye.y);
var frogsLeftEye_X:Number
  = _frog.localToGlobal(frogsLeftEye).x;
var frogsLeftEye_Y:Number
  = _frog.localToGlobal(frogsLeftEye).y;
```

> *Remember that to use this code, you have to create a Point object. Point objects can be created only if you import Flash's Point class with an import flash.geom.Point directive.*

The process of converting local to global coordinates like this is neither obvious nor intuitive. Just a glance at the code involved is enough to stun an ox at 50 paces. The one saving grace is that you don't need to memorize any of this - ever. That's what this book is for!

Figure 12-11 takes you on a tour of how to convert a subobject's local to global coordinates.

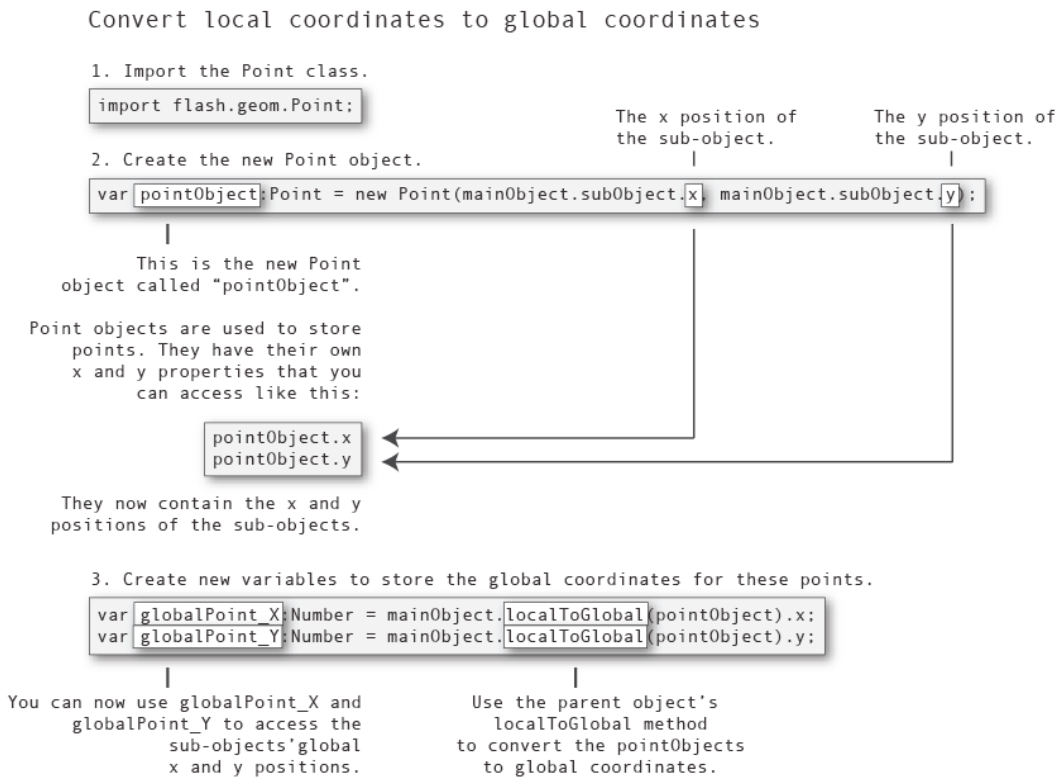


Figure 12-11. Convert local x and y coordinates to global x and y coordinates.

Now that you can access the global coordinates of the frog's left and right eyes, you can use these values to rotate them.

Rotating toward an object

To make the frog's eyes rotate to follow the cat, you need to apply a bit of trigonometry to the eyes' rotation properties.

If you're a mathophobe, rest assured that you don't need to necessarily *understand* the trigonometry we'll look at to be able to use it. If fact, you won't even see it. AS3.0 does the math for you—you just need to give it the correct numbers. And once you see it in use, you'll see how easy it is to apply whenever you need to rotate an object toward another object. It's a walk in the park compared with converting coordinate systems.

AS3.0 has a built-in method called `Math.atan2` that will give you the correct angle of rotation between two objects. One little technical detail you need to deal with, though, is that `Matn.atan2` returns the value of the angle in *radians.* Sprite objects in AS3.0 don't use radians for their rotation values; they use *degrees*. So once you've got the value in radians, you need to apply a very simple calculation to convert it into degrees. When you have that value, you can rotate the object.

Let me show you just how easy this all is. Figure 12-12 is a grid showing the positions of two objects on the stage.

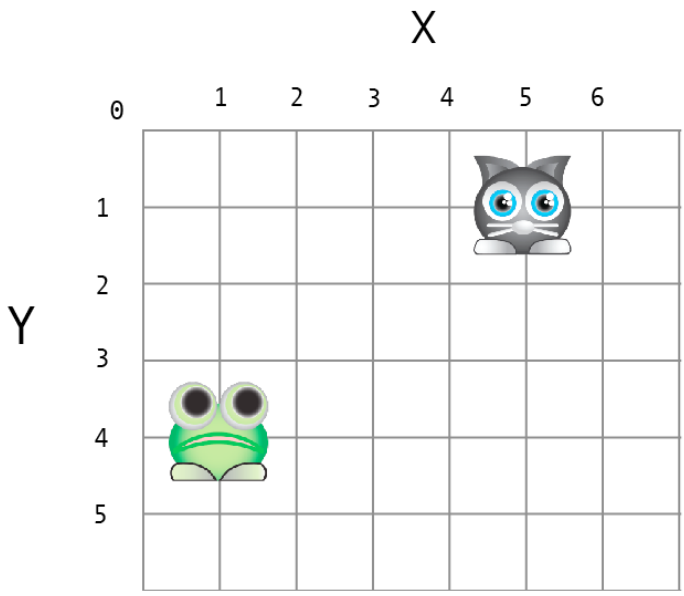


Figure 12-12. Find the angle of rotation between two objects.

Here's how to find out the angle of rotation, from the point of view of the frog:

1. Find the positions of the objects:

```
cat.x = 5
cat.y = 1
frog.x = 1
frog.y = 4
```

2. Subtract the x and y positions of the target object, from the x and y position of the object that will be doing the rotating:

```
frog.x - cat.x = -4
frog.y - cat.y = 3
```

3. Plug these numbers into the `Math.atan2` function. Very importantly, the y value has to come first:

```
Math.atan2(3, -4)
```

This returns a number in *radians.* Unfortunately, this is not useful for rotating Sprite objects. You need to convert this number to *degrees,* which is the type of value expected by the rotation property.

To convert radians to degrees, you can use a simple calculation: multiply the value in radians by 180 divided by Pi (3.14). AS3.0 has a built-in function called `Math.PI` that returns the value of Pi, just to make things easier for you. Here's what the final line of code might look like:

```
Math.atan2(3, -4) * (180/Math.PI)
```

This gives you a rounded-off value in degrees:

```
143
```

Figure 12-13 shows what this looks like in the example.

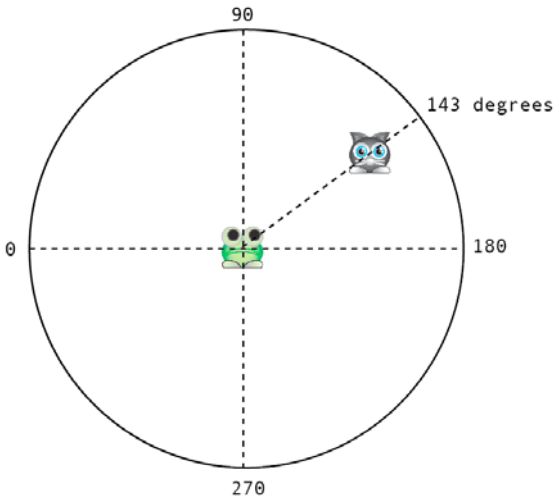


Figure 12-13. Use AS3.0's built- in Math.atan2 function to find the angle of rotation.

Great! You now have a number you can use to rotate the frog. You can use it in a directive like this:

```
frog.rotation = Math.atan2(3, -4) * (180/Math.PI);
```

Whenever you need to rotate an object toward another object, this is the formula you need to use. It's probably about 50% of the trigonometry you'll ever need to know for your games, and you didn't actually have to do the math yourself! In Chapter 10, you'll look at a few more of AS3.0's trigonometry functions.

Rotating the frog's eyes toward the player object

Of course, you don't want to rotate the frog; you want to rotate its eyes, which are sub-objects. You need to put together the rotation code with the earlier code to convert an object's local coordinates to global coordinates. This is the code that actually rotates the frog's eyes:

```
_frog.rightEye.rotation
  = Math.atan2
    (
      frogsRightEye_Y - _character.y,
      frogsRightEye_X - _character.x
    )
    * (180/Math.PI);

_frog.leftEye.rotation
  = Math.atan2
    (
      frogsLeftEye_Y  - _character.y,
      frogsLeftEye_X - _character.x
    )
    * (180/Math.PI);
```

These are two horrendously long single lines of code, so I've applied some creative formatting to make them more easily readable.

Bug Catcher rotates the frog's eyes each frame to match the character's position on the stage. As you know, the `runGame` method is called each frame when the game is in its `RUNNING` state. The `makeFrogLook` method is in the `runGame` method, so it's called every frame.

```
makeFrogLook();
```

Here's the entire `makeFrogLook` method which finds the eyes' global coordinates and rotates them towards the character.

```
private function makeFrogLook():void
```

```
{
  //Convert points from local to global coordinates
  //Right eye
  var frogsRightEye:Point
    = new Point(_frog.rightEye.x, _frog.rightEye.y);
  var frogsRightEye_X:Number
    = _frog.localToGlobal(frogsRightEye).x;
  var frogsRightEye_Y:Number
    = _frog.localToGlobal(frogsRightEye).y;
  //Left eye
  var frogsLeftEye:Point
    = new Point(_frog.leftEye.x, _frog.leftEye.y);
  var frogsLeftEye_X:Number
    = _frog.localToGlobal(frogsLeftEye).x;
  var frogsLeftEye_Y:Number
    = _frog.localToGlobal(frogsLeftEye).y;

  //Rotate eyes
  _frog.rightEye.rotation
    = Math.atan2
    (
      frogsRightEye_Y - _character.y,
      frogsRightEye_X - _character.x
    )
    * (180/Math.PI);

  _frog.leftEye.rotation
    = Math.atan2
    (
      frogsLeftEye_Y - _character.y,
           frogsLeftEye_X - _character.x
    )
    * (180/Math.PI);
}
```

Converting a point's coordinate space from local to global and rotating objects toward other objects are incredibly common requirement in games. File this one away for safekeeping; you'll be returning to it again and again.

> *The GameUtilities class contains a method called rotateToObject that will automatically rotate an object to another object for you. You can use it like this:*
>
> *`GameUtilities.rotateToObject(objectOne, objectTwo);`*

The stacking order of objects on the stage

When you add objects to the stage, they're layered in the order in which you add them. That means that if you add the background first, and the character second, the character will appear above the background. The order in which objects are added to the stage is called the **stacking order**. Sometimes you need to change the stacking order of objects while the game is in progress, so let's take a look at how you can do this.

When objects are added to the stage using `addChild`, AS3.0 adds them to something called the **display list**, which is just a list of all the objects on the stage. The first objects on the list appear *behind* objects that are added after them. To make an object appear in front of all the other objects, it needs to be the *last* item on the display list. The number of objects on the stage is stored in a built-in property called `numChildren`.

If you add the following directive in the `BugCatcher` constructor method, after you've created all the objects in the game, it will actually tell you the number of objects on the stage:

```
trace(numChildren);
```

This displays the number 23 for me in Bug Catcher.

You can find out the position of any object on the list by using another built in method called `getChildAt`. The display list numbers objects starting from 0, just like arrays do. I know that the character happens to be number 2 on the list (which actually means it's the third object, if you start counting from zero). I can use a method called `getChildAt` to make sure:

```
trace(getChildAt(2));
```

In Bug Catcher, this displays as follows:

```
[object Character]
```

`Character` is the class that the character belongs to. You can append the `name` property to `getChildAt` to find out the object's actual instance name. For example, use this line of code to find out the instance name of the object that's at position 2 in the display list:

```
trace(getChildAt(2).name);
```

This will display the following trace message:

```
instance1
```

This is the instance name that AS3.0 automatically assigned to the object.

> *If you ever need to, you can assign your own instance names to objects, like this:*
>
> ***`_character.name = "thisIsTheCharactersName";`***
>
> *You probably won't need to do this in any of your games if you're following the techniques in this book, but it's useful to keep in mind that you can do this if you ever need an extra way to differentiate between objects. However, if you're making game objects using Flash Professional software, you need to access the name property of objects to control them. You'll see how this is done in the bonus chapter "Flash Animation and Publishing Your Game" in the book's download site.*

Changing the stacking order

You can use `numChildren` with a `for` loop to view all the objects on the stage. A line of code in the `BugCatcher` constructor method does just that:

```
for (var k:int = 0;  k < numChildren;  k++)
{
        trace(k + ".  " + getChildAt(k));
}
```

Here's an abridged version of the output:

```
0.  [object Background]
1.  [object Grass]
2.  [object Character]
3.  [object Frog]
4.  [object Spider]
5.  [object Bug]
    ...
9.  [object Box]
    ...
22.  [object Box]
```

You can see from this list that the grass is behind the character. I actually want the grass to be in front of the character, so that the character is partially hidden by blades of grass when it's on the ground. Another built-in method called `setChildlndex` can help me do this, by moving the grass to the end of the display list.

To move an object to the end of the display list, use `setChildlndex` to give the object a position number that is one less than the number provided *by* `numChildren.` It has to be one less to compensate for the fact that the

list starts at zero. `setChildIndex` takes two arguments: the name of the object and the position you want to move it to. Here's how I can move the grass to the end of the display list

```
setChildIndex(_grass, numChildren - 1);
```

Now if I loop through all the objects in the display list again, I'll see that the grass is now the last object on the list.

```
...
20.  [object Box]
21.  [object Box]
22.  [object Grass]
```

And of course, I'll also notice that when I re-compile the game that the grass is in front of all the other objects. The cat can hide comfortably behind the blades of grass as it stalks the bugs, as shown in Figure 14.

Figure 12-14. Move an object to the end of the display list to it make appear in front of other objects on the stage.

Another of AS3.0's built- in methods that you'll certainly find use for in your games is `swapChildren`. You can swap the position of two objects in the display list by using a line of code that looks like this:

```
swapChildren(_character, _grass);
```

This code makes the two objects change places in the list. It's useful to be able to do this when you need to make one object appear above or below another object without having to know their actual positions in the list. Figure 12-15 shows what effect swapping the display list positions of the character and grass would have in the game.

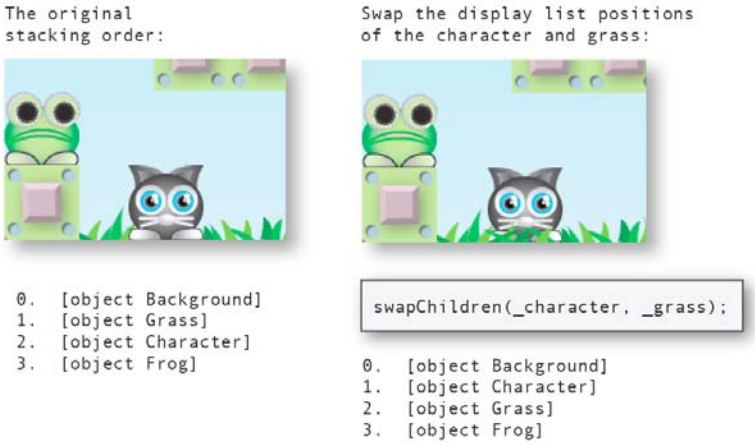


Figure 12-15. Swap the display list positions of objects to change the stacking order.

Adding some bugs to the code—literally!

The bugs are added to the game just as we've added boxes and monsters in other examples, by using arrays and loops. Bug Catcher uses two arrays for the bugs: one to store the initial bug positions and a second to store the actual bug objects themselves.

```
private var _bugs:Array = new Array();
private var _bugPositions:Array = new Array();
```

The `BugCatcher` constructor method then creates the three bugs using the code we've seen in other examples.

```
//Set the initial bug x and y positions
_bugPositions
  = [
      [25, 50],
      [400, 50],
      [250, 150]
    ];

//Make the bugs
for(var i:int = 0; i < _bugPositions.length; i++)
{
  //Create a bug object
  var bug:Bug = new Bug();

  //Add the bug to the stage
  GameUtilities.addObjectToGame
    (bug, this, _bugPositions[i][0], _bugPositions[i][1]);

  //Add it to the _bugs array
  //for future use
  _bugs.push(bug);
}
```

The `runGame` method makes the bugs move by looping through the `_bugs` array and, if they're visible, sends them to the `moveBug` method.

```
public function runGame():void
{
  //Move the bugs
  for(var i:int = 0; i < _bugs.length; i++)
  {
    //Create a temporary local variable
    //to easily access the current bug in the loop
    var bug:Bug = _bugs[i];

    if(bug.visible)
    {
      moveBug(bug);
    }
  }
  //…
}
```

The `moveBug` method is what actually makes the bugs fly around the stage. Let's find out how it works.

Making the bugs move

Here's the entire `moveBug` method, and we'll take a look at how all this code works in detail ahead.

```
private function moveBug(bug:Bug):void
{
  //Add Brownian motion to the velocities
  bug.vx += (Math.random() * 0.2 - 0.1) * 40;
  bug.vy += (Math.random() * 0.2 - 0.1) * 40;

  //Add some friction
  bug.vx *= 0.96;
  bug.vy *= 0.96;

  //Move the bug
  bug.x += bug.vx;
  bug.y += bug.vy;

  //Stage Boundaries
  if (bug.x > stage.stageWidth - bug.width)
  {
    bug.x = stage.stageWidth - bug.width;

    //Reverse (bounce) bug's velocity when it hits the stage edges
    bug.vx *= -1;
  }
  if (bug.x < 0)
  {
    bug.x = 0;
    bug.vx *= -1;
  }
  //Keep the bug above the grass
  if (bug.y > stage.stageHeight - 35)
  {
    bug.y = stage.stageHeight - 35;
    bug.vy *= -1;
  }
  if (bug.y < 0)
  {
    bug.y = 0;
    bug.vy *= -1;
  }

  //Apply collision detection with the platforms
  for(var j:int = 0; j < _boxes.length; j++)
  {
    Collision.block(bug, _boxes[j]);
  }

  //ARTIFICIAL INTELLIGENCE
  //Make the bugs avoid the frog
  if ((Math.abs(bug.x - _frog.x) < 100))
  {
    if (Math.abs(bug.y - _frog.y) < 100)
    {
      bug.vx *= - 1;
      bug.vy *= - 1;
      trace(bug.name + ": Frog!");
    }
  }

  //Calculate the distance vector
  //between the center of the
  //bug and the character
  var vx:Number
```

```
    = (bug.x + (bug.width / 2))
    - (_character.x + (_character.width / 2));

  var vy:Number
     = (bug.y + (bug.height / 2))
     - (_character.y + (_character.height / 2));

  if (Math.abs(vx) < 100 && Math.abs(vy) < 100)
  {
    //If the player is moving...
    bug.vy += _character.vy + ((Math.random() * 0.2 - 0.1) * 70);
    bug.vx += _character.vx + ((Math.random() * 0.2 - 0.1) * 70);
    trace(bug.name + ": Cat!");

    //If the player is sitting still...
    if ((Math.abs(_character.vy) < 1)
    && (Math.abs(_character.vx) < 1))
    {
      bug.y += -bug.vy;
      bug.x += -bug.vx;
      bug.vy *= -1;
      bug.vx *= -1;
    }
  }
}
```

Using Brownian motion

If you play Bug Catcher, you'll notice that the bugs move around the stage in a way that's remarkably similar to real flying insects. It's an erratic combination of randomness and determination. There is a formula for creating this effect called **Brownian motion**. The formula for Brownian motion looks like this:

```
Math.random() * 0.2 - 0.1
```

It uses the `Math.random` method to generate a random number between -0.1 and 0.1 (Refer to Chapter 4 if you need a refresher on how to generate random numbers.) This number is too small to be much use for moving an object on the stage, so you need to multiply it by another number to amplify the effect. Through trial and error, I noticed that multiplying it by 40 looked good for my bugs.

This new formula is then added to the bugs' vx and vy properties:

```
bug.vx += (Math.random() * 0.2 - 0.1) * 40;
bug.vy += (Math.random() * 0.2 - 0.1) * 40;
```

The code adds some friction and then adds the vx and vy velocities to the bugs' x and y properties to make it move.

```
//Add some friction
bug.vx *= 0.96;
bug.vy *= 0.96;

//Move the bug
bug.x += bug.vx;
bug.y += bug.vy;
```

This amazingly mundane code is all that's needed to make the bug object move like a real bug. Makes you think!

> *Brownian motion is great if you want to make objects that move in a way that mimics the organic randomness of insects, dust particles, or snow. Experiment with the friction value and change the multiplier from 40 to something like 50 or 30.*

The next section of code simply sets the stage boundaries. It uses the simple bounce formula we looked at earlier in the chapter to bounce the bugs off the stage edges when they get too close, like this:

```
bug.vx *= -1;
```

Another very simple bit of code that results in surprising complexity!

The method then checks for collisions between the bugs and platforms, using this familiar bit of code:

```
for(var j:int = 0; j < _boxes.length; j++)
{
  Collision.block(bug, _boxes[j]);
}
```

Artificial intelligence

The next step is to make the bugs aware of their environment. Bugs don't like frogs, so they fly away if they come within 100 pixels of the frog.

The code checks whether a bug is within 100 pixels of the frog's top left corner on the x axis. If it is, it checks whether the bug is within 100 pixels of the frog on the y axis. If that's true as well, the bug is too close to the frog, and it changes its direction. It also displays a trace message with the name of the bug instance and the word "Frog!" if it comes too close.

```
if ((Math.abs(bug.x - _frog.x) < 100))
{
        if (Math.abs(bug.y - _frog.y) < 100)
        {
                bug.vx *= - 1;
                bug.vy *= - 1;
                trace(bug.name + ": Frog!");
        }
}
```

Figure 12-16 shows what you'll see.

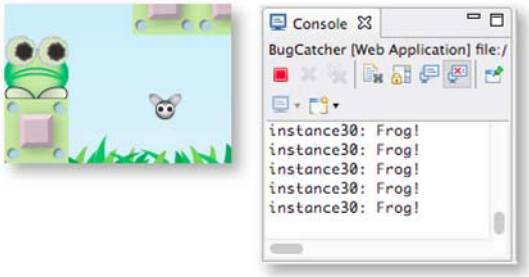


Figure 12-16. The bugs reverse direction if they come within 100 pixels of the frog.

The code pushes the bug out of the collision area and then reverses its velocity using the bounce formula. Again, this is simple code, but the effect is startlingly realistic.

The way that the bugs interact with the cat is only slightly more complex. The code first figures out the distance between the bug and the character, using the same technique we used to measure distances between objects in chapter 6. It calculates the distance vector between them, and stores this information in vx and vy variables.

```
var vx:Number
  = (bug.x + (bug.width / 2))
  - (_character.x + (_character.width / 2));

var vy:Number
  = (bug.y + (bug.height / 2))
  - (_character.y + (_character.height / 2));
```

The code then checks to see if the bug and character are within 100 pixels of each other

```
if (Math.abs(vx) < 100 && Math.abs(vy) < 100)
```

```
{...
```

The bugs have two different behaviors, depending on whether the cat is moving or not moving. When the cat's velocity is less than 1 on either axis, the bugs have the same behavior as they have with the frog. When the cat is moving, however, the bugs enter panic mode. They add the character's velocity to their own and multiply their random motion by 70, which makes their movement much more erratic.

```
//If the player is moving...
bug.vy += _character.vy + ((Math.random() * 0.2 - 0.1) * 70);
bug.vx += _character.vx + ((Math.random() * 0.2 - 0.1) * 70);
trace(bug.name + ": Cat!");

//If the player is sitting still...
if ((Math.abs(_character.vy) < 1)
&& (Math.abs(_character.vx) < 1))
{
  bug.y += -bug.vy;
  bug.x += -bug.vx;
  bug.vy *= -1;
  bug.vx *= -1;
}
```

The bugs will flee wildly from the cat if it chases them, and bob around it complacently if the cat sits still, as shown in Figure 17.

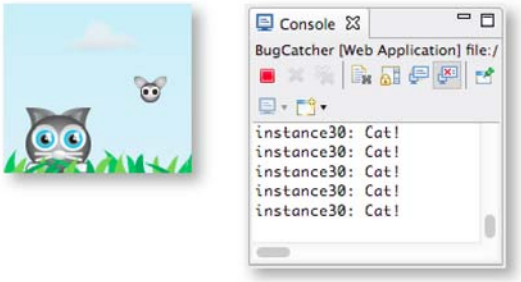


Figure 12-17. The bugs fly away if they come within 80 pixels of the cat.

All this adds up to very realistic bug behavior. But as you can see, there's nothing too special about his code, and there's certainly no magic formula for writing it. I wrote this code with a vague idea of how I wanted the bugs to behave and spent an hour or so playing around with a few different combinations of directives until I found something I was happy with. To say that there are as many different ways the code could be written as there are readers of the book is an understatement. There are millions of ways this code could have been written, and the example I present here is just one possibility I dreamt up during an afternoon monsoon downpour in a back alley in Kathmandu. When you're programming objects that need some sort of awareness of the environment they inhabit, try and break down their behavior into small steps. Solve one step and build from there. With just a few lines of code and a few simple `if` statements you can create something that appears alive and truly intelligent. It's really not hard; try it!

Summary

Bug Catcher is the most technically complex game in the book, and a good jumping off point for you to build your own platform game. Of course, you'll going to want to use a scrolling background, some enemies, and multiple levels. You can easily implement all those things by referring back to chapters 6, 7 and 8.

And what about a moving platforms, like an elevator? You learnt how to make moving enemies in chapter 8, so you can easily turn them into moving platforms using the `Collision.block` method and apply what you learnt about platform collisions in chapter 8.