CHAPTER 17

Creating Animated Apps

This chapter discusses methods for creating apps with simple animations (objects that move). You'll learn the basics of creating two-dimensional games with App Inventor and become comfortable with image sprites and handling events such as two objects colliding.

Figure 17-1.

When you see an object moving smoothly along the computer screen, what you're really seeing is a quick succession of images with the object in a slightly different place each time. It's an illusion not much different from flipbooks, in which you see a moving picture by flipping quickly through the pages. It's the concept behind how animated films are made!

With App Inventor, you'll program animation by placing `Ball` and `ImageSprite` components within a `Canvas` component and then moving and transforming those objects every successive fraction of a second. In this chapter, you'll learn how the `Canvas` coordinate system works, how you can use the `Clock.Timer` event to trigger movement, how to control the speed of objects, and how to respond to events such as two objects crashing into each other.

Adding a Canvas Component to Your App

From the Drawing and Animation palette, drag a `Canvas` component into your app. After you place it, specify its `Width` and `Height`. Often, you'll want the `Canvas` to span the width of the device screen. To do this, choose "Fill parent" when specifying the `Width`.

You can do the same for the `Height`, but generally you'll set this to some number (e.g., 300 pixels) to leave room for other components above and below the `Canvas`.

The Canvas Coordinate System

A drawing on a `Canvas` is really a grid of *pixels*, where a pixel is the tiniest possible dot of color that can appear on the screen. Each pixel's location is defined by x-y coordinates on a grid system, as illustrated in *Figure 17-1*. In this coordinate system, x defines a location on the horizontal plane (starting at 0 on the far left and increasing as you move to the right across the screen), and y defines a location on the vertical plane (starting at 0 at the top and increasing as you move down the screen).

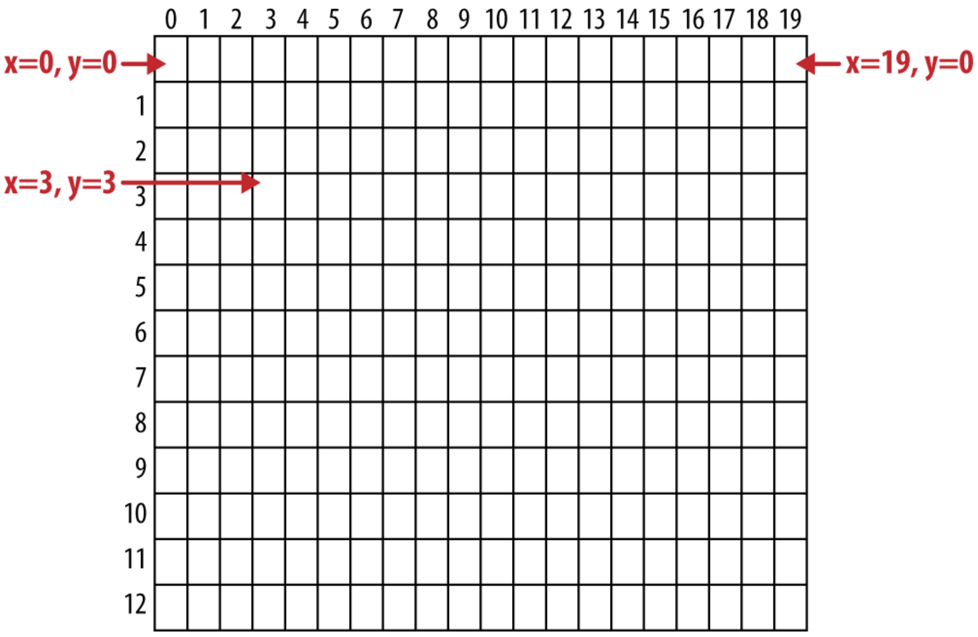


Figure 17-2. The Canvas coordinate system

The top-left cell in a `Canvas` starts with 0 for both coordinates, so this position is represented as (x=0,y=0). As you move right, the x coordinate increases; as you move down, the y coordinate increases. The cell to the immediate right of the top-left corner is (x=1,y=0). The upper-right corner has an x coordinate equal to the width of the `Canvas` minus 1. Most phone screens have a width around 300 pixels, but for the sample shown here, the `Width` is 20, so the upper-right corner is the coordinate (x=19,y=0).

You can change the appearance of the canvas in two ways: by painting on it, or by placing and moving objects within it. This chapter focuses primarily on the latter, but let's first discuss how you "paint" and how to create animation by painting (this is also the topic of the PaintPot app in *Chapter 2*).

Each cell of the Canvas holds a pixel defining the color that should appear there. The Canvas component provides the Canvas.DrawLine and Canvas.DrawCircle blocks for painting pixels. You first set the Canvas.PaintColor property to the color you want and then call one of the Draw blocks to draw in that color. With DrawCircle, you can paint circles of any radius, but if you set the radius to 1, as shown in *Figure 17-2*, you'll paint an individual pixel.

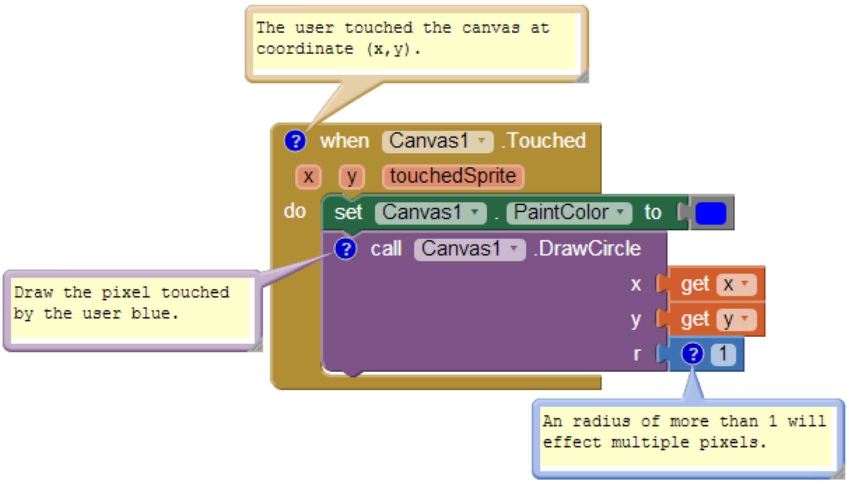

The user touched the canvas at coordinate (x,y).
when Canvas1 .Touched
x y touchedSprite
do set Canvas1 . PaintColor to
call Canvas1 .DrawCircle
x get x
y get y
r 1
Draw the pixel touched by the user blue.
An radius of more than 1 will effect multiple pixels.

Figure 17-3. DrawCircle with a radius of 1 paints an individual pixel with each touch

App Inventor provides a palette of basic colors that you can use to paint pixels (or other user interface components). You can access a wider range of colors by using the color numbering scheme explained in the App Inventor documentation at *http://appinventor.mit.edu/explore/ai2/support/blocks/colors.html*.

Besides painting individual pixels, you can also place Ball and ImageSprite components on a Canvas. A *sprite* is a graphical object placed within a larger scene (in App Inventor, the "scene" is a Canvas component). Both the Ball and ImageSprite components are sprites; they are different only in appearance. A Ball is a circle that has an appearance that can only be modified by changing its color or radius, whereas an ImageSprite can take on any appearance, as defined by an image file. Balls and ImageSprites can only be added within a Canvas; you can't drag them into the user interface outside of one.

Animating Objects with Timer Events

One way to specify animation in App Inventor is to change an object in response to a timer event. Most commonly, you'll move sprites to different locations on the canvas at set time intervals. Using timer events is the most common method of defining those set time intervals. Later, we'll also discuss an alternative method of programming animation using the `Speed` and `Heading` properties of the `ImageSprite` and `Ball` components.

Button clicks and other user-initiated events are simple to understand: the user does something, and the app responds by performing some operations. Timer events are different, though, because they aren't triggered by the end user but instead by the passing of time. You need to conceptualize the phone's clock triggering events in the app instead of a user doing something.

To define a timer event, you first drag a `Clock` component into your app within the Component Designer. The `Clock` component has a `TimerInterval` property associated with it. The interval is defined in milliseconds (1/1,000 of a second). If you set the `TimerInterval` to 500, that means a timer event will be triggered every half-second. The smaller the `TimerInterval`, the faster the frame-rate of the animation.

After adding a `Clock` and setting a `TimerInterval` in the Designer, you can drag out a `Clock.Timer` event in the Blocks Editor. You can put any blocks you like in this event, and they'll be performed every interval.

Creating Movement

To show a sprite moving over time, you'll use the `MoveTo` function found in both the `ImageSprite` and `Ball` components. For example, to move a ball horizontally across the screen, you'd use blocks like those in *Figure 17-3*.

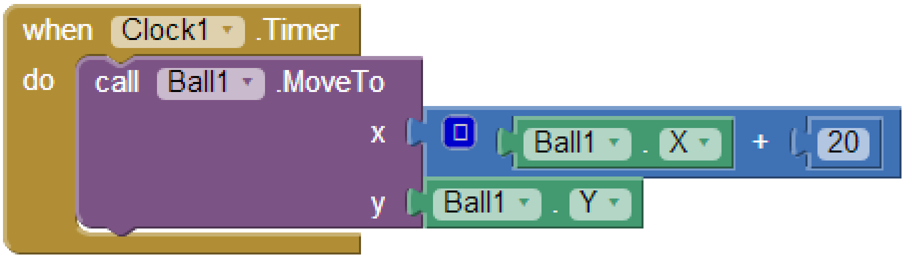

when Clock1 .Timer
do call Ball1 .MoveTo
x Ball1 . X + 20
y Ball1 . Y

Figure 17-4. Moving the ball horizontally across the screen

`MoveTo` moves an object to an *absolute* location on the canvas, not a relative amount. So, to move an object some amount, you set the `MoveTo` arguments to the object's current location plus an offset. Because we're moving horizontally, the x argument is set to the current x location (`Ball1.X`) plus the offset 20, whereas the y argument is set to stay at its current setting (`Ball1.Y`).

To move an object down, you'd modify just the `Ball1.Y` coordinate and leave `Ball1.X` the same. If you wanted to move the ball diagonally, you'd add an offset to both the x and y coordinates, as shown in *Figure 17-4*.

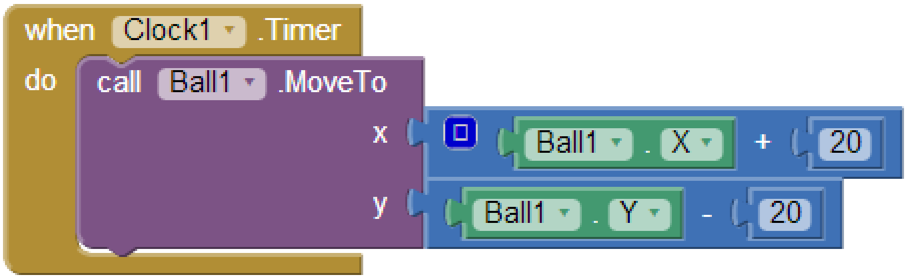


Figure 17-5. Offsetting both the x and y coordinates to move the ball diagonally

Speed

How fast is the ball moving in the preceding example? The speed depends on both the settings you provide for the `TimerInterval` property of `Clock1` and the parameters you specify in the `MoveTo` function. If the `Clock.TimerInterval` is set to 1,000 milliseconds, this means that a `Clock1.Timer` event will be triggered every second. For the horizontal example shown in *Figure 17-3*, the ball will move 20 pixels per second.

But a `TimerInterval` of 1,000 milliseconds doesn't provide very smooth animation; the ball will only move once per second, which will appear jerky. To achieve smoother movement, you need a shorter interval. If the `TimerInterval` were instead set to 100 milliseconds, the ball would move 20 pixels every tenth of a second, or 200 pixels per second—a rate that will appear much smoother.

Collision Detection

To create games and other animated apps, you need more complex functionality than just movement. Fortunately, App Inventor provides some high-level blocks for dealing with animation events such as an object reaching the screen's edge or two objects colliding.

In this context, *high-level block* means that App Inventor takes care of the *lower-level* details of determining when an event such as a collision occurs. You could check for these occurrences manually by checking sprite and canvas properties within `Clock.Timer` events. This level of programming requires some fairly complex logic, however. Fortunately, App Inventor provides them for you, and it's a good thing that it does because these events are common to many games and other apps.

EdgeReached

Consider again the animation in *Figure 17-4*, in which the object is moving diagonally from the upper left to the lower right of the canvas. As programmed, the ball would move diagonally and then stop when it reached the right or bottom edge of the canvas (the system won't move an object past the canvas boundaries).

If you instead wanted the object to reappear at the upper-left corner each time it reached the bottom or right edge, you could define a response to the `Ball.EdgeReached` event similar to that shown in *Figure 17-5*.

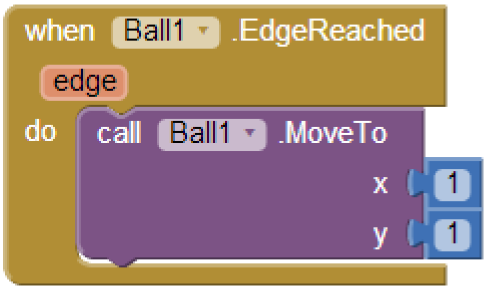


Figure 17-6. Making the ball reappear in the upper-left corner when it reaches an edge

`EdgeReached` is triggered when a ball or image sprite hits any edge of a canvas. This event handler, combined with the diagonal movement specified with the previously described timer event (*Figure 17-4*), causes the ball to move diagonally from upper left to lower right, pop back up to the upper left when it reaches an edge, and then do it all over again, forever (or until the app tells it otherwise).

For this example, we didn't distinguish between which edge was hit. The `EdgeReached` event does have a parameter, `edge`, which specifies the particular edge by using the following code:

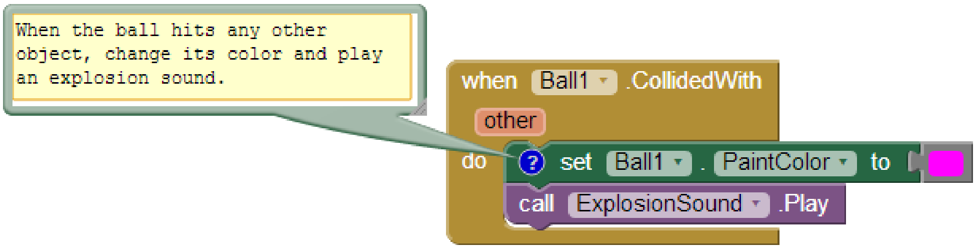
- North = 1
- Northeast = 2
- East = 3
- Southeast = 4
- South = –1
- Southwest = –2

- West = –3
- Northwest = –4

CollidingWith and NoLongerCollidingWith

Games and other animated apps often rely on activity occurring when two or more objects collide (e.g., a bat hitting a ball).

Consider a game, for instance, in which an object changes colors and plays an explosion sound when it hits another object. *Figure 17-6* shows the blocks for such an event handler.

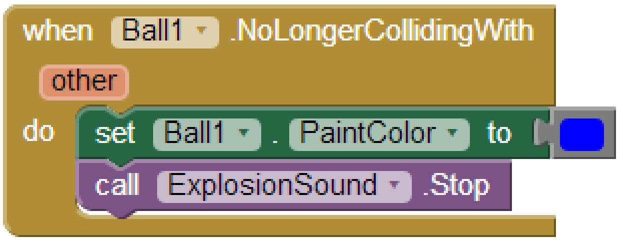


```
When the ball hits any other
object, change its color and play
an explosion sound.

when Ball1 .CollidedWith
  other
do  set Ball1 . PaintColor to
    call ExplosionSound .Play
```

Figure 17-7. Making the ball change color and play an explosion sound when it hits another object

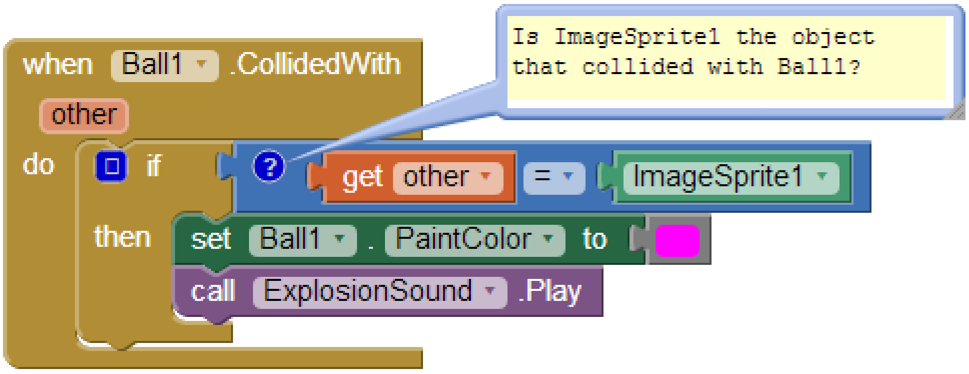
`NoLongerCollidingWith` provides the opposite event of `CollidedWith`. It is triggered only when two objects have come together and then separated. So, for your game, you might include the blocks depicted in *Figure 17-7*.



```
when Ball1 .NoLongerCollidingWith
  other
do  set Ball1 . PaintColor to
    call ExplosionSound .Stop
```

Figure 17-8. Changing the color back and stopping the explosion noise when the objects separate

Note that both `CollidedWith` and `NoLongerCollidingWith` have an argument, `other`, which specifies the particular object with which you collided (or from which you separated). This allows you to perform operations only when the object (e.g., `Ball1`) interacts with a particular other object, as shown in *Figure 17-8*.



```
when Ball1 .CollidedWith
  other
do  if  get other = ImageSprite1
    then  set Ball1 . PaintColor to
          call ExplosionSound .Play
```

Figure 17-9. Only perform the response if Ball1 hit ImageSprite1

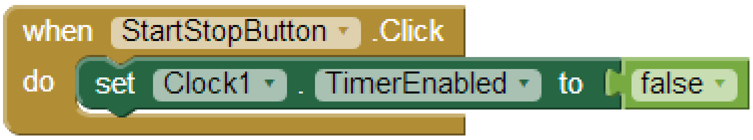
The `ImageSprite1` block is one we haven't yet discussed. This block refers to the component as a whole, not a particular property of the component. When you need to compare components (e.g., to know which ones have collided), you use this block. Each component has such a block in its drawer, and the block has the same name as the component.

Interactive Animation

In the animated behaviors we've discussed so far, the end user isn't involved. Games are interactive, of course, with the end user playing a central role. Often, the end user controls the speed or direction of an object with buttons or other user interface objects.

As an example, let's update the diagonal animation by giving the user the ability to stop and start the diagonal movement. You can do this by programming a `Button.Click` event handler to disable and re-enable the timer event of the clock component.

By default, the `Clock` component's `timerEnabled` property is checked. You can disable it dynamically by setting it to false in an event handler. The event handler in *Figure 17-9*, for example, would stop the activity of a `Clock` timer on the first click.



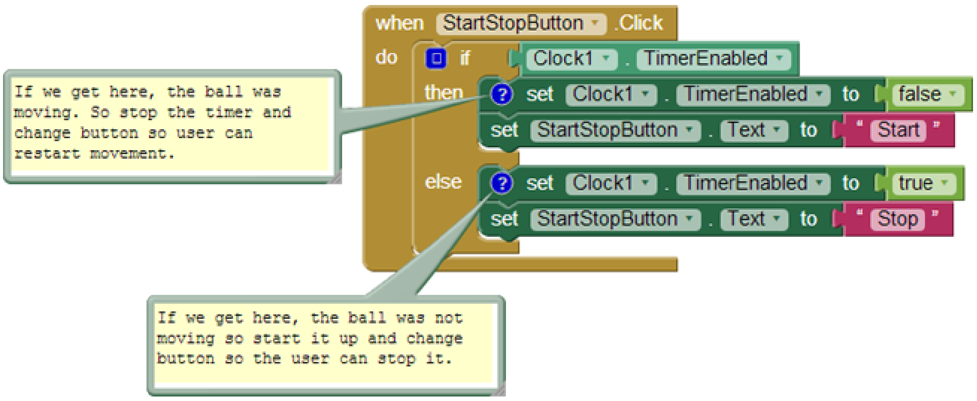
```
when StartStopButton .Click
do  set Clock1 . TimerEnabled to false
```

Figure 17-10. Stopping the timer the first time the button is clicked

After the `Clock1.TimerEnabled` property is set to false, the `Clock1.Timer` event will no longer trigger, and the ball will stop moving.

Of course, stopping the movement on the first click isn't too interesting. Instead, you could "toggle" the movement of the ball by adding an `if else` in the event handler that either enables or disables the timer, as demonstrated in *Figure 17-10*.

This event handler stops the timer on first click and resets the button so that it displays "Start" instead of "Stop." The second time the user clicks the button, the `TimerEnabled` is false, so the "else" part is executed. In this case, the timer is enabled, which gets the object moving again, and the button text is switched back to "Stop." For more information about `ifelse` blocks, see *Chapter 18*, and for examples of interactive animations that use the orientation sensor, see *Chapter 5* and *Chapter 23*.



```
when StartStopButton.Click
do  if Clock1.TimerEnabled
    then set Clock1.TimerEnabled to false
         set StartStopButton.Text to "Start"
    else set Clock1.TimerEnabled to true
         set StartStopButton.Text to "Stop"
```

If we get here, the ball was moving. So stop the timer and change button so user can restart movement.

If we get here, the ball was not moving so start it up and change button so the user can stop it.

Figure 17-11. Adding an if else so that clicking the button starts and stops the movement of the ball

Specifying Sprite Animation Without a Clock Timer

The animation samples described so far use a `Clock` component and specify that an object should move each time the `Clock.Timer` event is triggered. The `Clock.Timer` event scheme is the most general method of specifying animation. Beyond simply moving an object, you could also have it change an object's color over time, change some text (to appear as though the app is typing), or have the app speak words at a certain pace.

If you only want to move objects, App Inventor provides an alternative that doesn't require the use of a `Clock` component. As you might have noticed, the `ImageSprite` and `Ball` components have properties for `Heading`, `Speed`, and `Interval`. Instead of defining a `Clock.Timer` event handler, you can set these properties in the Component Designer or Blocks Editor to control how a sprite moves.

To illustrate, let's reconsider the example that moved a ball diagonally. The `Heading` property of a sprite or ball has a range of 360 degrees, as illustrated in *Figure 17-11*.

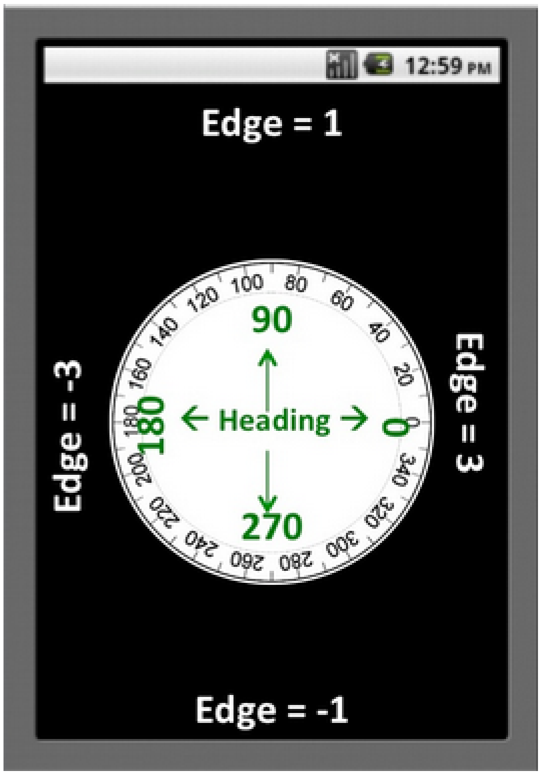


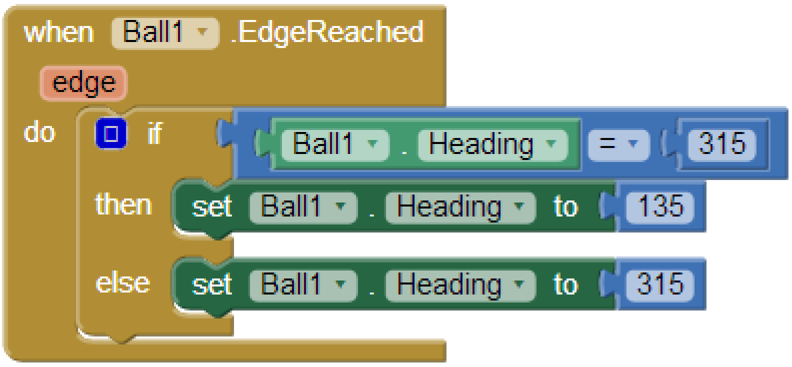
Figure 17-12. The Heading property has a range of 360 degrees

If you set the `Heading` to 0, the ball will move left to right. If you set it to 90, it will move bottom to top. If you set it to 180, it will move right to left. If you set it to 270, it will move top to bottom. And if you set it to 315, the ball will move from upper left to lower right.

To cause an object to move, you also need to set the `Speed` property to a value other than 0. The speed the object moves is actually determined by the `Speed` and `Interval` properties together. The `Speed` property is the distance, in pixels, that the object will move each `Interval`.

To try out these properties, create a test app with a `Canvas` and `Ball` and connect your device or emulator for live testing. Then, modify the `Heading`, `Speed`, and `Interval` properties of the ball to see how they work.

For instance, suppose that you wanted to move a ball back and forth from the upper left to the lower right of the canvas. In the Designer, you might initialize the ball's `Speed` to 5 and `Interval` to 100, and then set the `Heading` property to 315. You'd then add the `Ball1.EdgeReached` event handler, which you can see in *Figure 17-12*, to change the ball's direction when it reaches either edge.



```
when Ball1 .EdgeReached
  edge
do  if    Ball1 . Heading = 315
    then  set Ball1 . Heading to 135
    else  set Ball1 . Heading to 315
```

Figure 17-13. Changing the ball's direction when it reaches either edge

Summary

Using the `Canvas` component, you can define a sub-area of the device's screen in which objects can move around and interact. You can put only two types of components within a `Canvas`: `ImageSprites` and `Balls`.

Animation is an object moving or otherwise transforming over time. You can program animation, including movement and other graphical transformations, with the `Clock` component's `Timer` event. If you just want to move objects, you can use an alternative method based on the `Heading`, `Speed`, and `Interval` properties internal to `ImageSprite` and `Ball` components.

With either method, you can also take advantage of high-level functionality for handling events that handle collisions.