Abraham Gebrekidan
EIP 01

Reading #15
August 12, 2024

Event-Driven Programs

Event-driven Programs

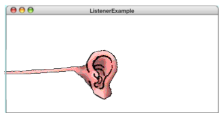
Abraham Gebrekidan
EIP 01
August 09, 2024

The JavaScript Event Model

- Graphical applications usually make it possible for the user to control the action of a program by using an input device such as a mouse. Programs that support this kind of user control are called ***interactive programs***.
- User actions such as clicking the mouse are called ***events***. Programs that respond to events are said to be ***event-driven***.
- In modern interactive programs, user input doesn't occur at predictable times. A running program doesn't tell the user when to click the mouse. The user decides when to click the mouse, and the program responds. Because events are not controlled by the program, they are said to be ***asynchronous***.
- In JavaScript program, you write a function that acts as a ***listener*** for a particular event type. When the event occurs, that listener is called.

The Role of Event Listeners

- One way to visualize the role of a listener is to imagine that you have access to one of Fred and George Weasley's "Extendable Ears" from the Harry Potter series.
- Suppose that you wanted to use these magical listeners to detect events in the canvas shown at the bottom of the slide. All you need to do is send those ears into the room where, being magical, they can keep you informed on anything that goes on there, making it possible for you to respond.



First-Class Functions

- Writing listener functions requires you to make use of one of JavaScript's most important features, which is summed up in the idea that functions in JavaScript are treated as data values just like any others.
- Given a function in JavaScript, you can assign it to a variable, pass it as a parameter, or return it as a result.
- Functions that are treated like any data value are called ***first-class functions***.
- The reading material includes examples of how first-class functions can be used to write a program that generates a table of values for a client-supplied function. The focus in today's session is using first-class functions as listeners.

Declaring Functions using Assignment

- The syntax for function definitions you have been using all along is really just a convenient shorthand for assigning a function to a variable. Thus, instead of writing

```
function fahrenheitToCelsius(f) {
   return 5 / 9 * (f - 32);
}
```

JavaScript allows you to write

```
var fahrenheitToCelsius = function(f) {
   return 5 / 9 * (f - 32);
};
```

- Note that this form is a declaration and requires a semicolon.

Closures

- The assignment syntax has few advantages over the more familiar definition for functions defined at the highest level of a program.
- The real advantage of declaring functions in this way comes when you declare one function as a local variable inside another function. In that case, the inner function not only includes the code in the function body but also has access to the local variables in the outer function.
- This combination of a function definition and the collection of local variables available in the stack frame in which the new function is defined is called a ***closure***.
- Closures are essential to writing interactive programs in JavaScript, so it is worth going through several examples in detail.

A Simple Interactive Example

- The first interactive example in the text is `DrawDots`:

```
function DrawDots() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var clickAction = function(e) {
      var dot = GOval(e.getX() - DOT_SIZE / 2,
                      e.getY() - DOT_SIZE / 2,
                      DOT_SIZE, DOT_SIZE);
      dot.setFilled(true);
      gw.add(dot);
   };
   gw.addEventListener("click", clickAction);
}
```

- The key to understanding this program is the `clickAction` function, which defines what to do when the mouse is clicked.
- It is important to note that `clickAction` has access to the `gw` variable in `DrawDots` because `gw` is included in the closure.

Registering an Event Listener

- The last line in the `DrawDots` function is

```
gw.addEventListener("click", clickAction);
```

which tells the graphics window (`gw`) to call `clickAction` whenever a mouse click occurs in the window.

- The definition of `clickAction` is

```
var clickAction = function(e) {
   var dot = GOval(e.getX() - DOT_SIZE / 2,
                   e.getY() - DOT_SIZE / 2,
                   DOT_SIZE, DOT_SIZE);
   dot.setFilled(true);
   gw.add(dot);
};
```

Callback Functions

- The `clickAction` function in the `DrawDots.js` program is representative of all functions that handle mouse events. The `DrawDots.js` program passes the function to the graphics window using the `addEventListener` method. When the user clicks the mouse, the graphics window, in essence, calls the client back with the message that a click occurred. For this reason, such functions are known as ***callback functions***.
- The parameter `e` supplied to the `clickAction` function is a data structure called a ***mouse event***, which gives information about the specifics of the event that triggered the action.
- The programs in the reading material use only two methods that are part of the mouse event object: `getX()` and `getY()`. These methods return the *x* and *y* coordinates of the mouse click in the coordinate system of the graphics window.

Mouse Events

- The following table shows the different mouse-event types:

`"click"`	The user clicks the mouse in the window.
`"dblclk"`	The user double-clicks the mouse.
`"mousedown"`	The user presses the mouse button.
`"mouseup"`	The user releases the mouse button.
`"mousemove"`	The user moves the mouse with the button up.
`"drag"`	The user drags the mouse with the button down.

- Certain user actions can generate more than one mouse event. For example, clicking the mouse generates a `"mousedown"` event, a `"mouseup"` event, and a `"click"` event, in that order.
- Events trigger no action unless a client is listening for that event type. The `DrawDots.js` program listens only for the `"click"` event and is therefore never notified about any of the other event types that occur.

A Simple Line-Drawing Program

In all likelihood, you have at some point used an application that allows you to draw lines with the mouse. In JavaScript, the necessary code fits easily on a single slide.

```
import "graphics";

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var line = null;
   var mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   var dragAction = function(e) {
      line.setEndPoint(e.getX(), e.getY());
   };
   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
}
```

A Simple Line-Drawing Program

As you drag the mouse, the line will stretch, contract, and change direction as if the two points were connected by an elastic band. This technique is therefore called ***rubber-banding***.

```
import "graphics";

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var line = null;
   var mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   var dragAction = function(e) {
      line.setEndPoint(e.getX(), e.getY());
   };
   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
}
```

A Simple Line-Drawing Program

The effect of this strategy is that the user sees the line as it grows, providing the necessary visual feedback to position the line correctly.

```
import "graphics";

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var line = null;
   var mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   var dragAction = function(e) {
      line.setEndPoint(e.getX(), e.getY());
   };
   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
}
```

A Simple Line-Drawing Program

The effect of this strategy is that the user sees the line as it grows, providing the necessary visual feedback to position the line correctly.

```
import "graphics";

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var line = null;
   var mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   var dragAction = function(e) {
      line.setEndPoint(e.getX(), e.getY());
   };
   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
}
```

Simulating the `DrawLines` Program

– The two calls to **`addEventListener`** register the listeners.

– Depressing the mouse button generates a **`"mousedown"`** event.

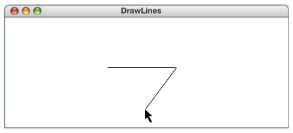
– The **`mousedownAction`** call adds a zero-length line to the canvas.

– Dragging the mouse generates a series of **`"drag"`** events.

– Each **`dragAction`** call extends the line to the new position.

– Releasing the mouse stops the dragging operation.

– Repeating these steps adds new lines to the canvas.



Timer Events

- The programs you saw on Friday responded to mouse events by adding an event listener to the **`GWindow`** object.
- JavaScript also allows you to listen for ***timer events***, which occur after a specified time interval.
- As with mouse events, you specify the listener for a timer event in the form of a callback function that is automatically invoked at the end of the time interval.
- You can add animation to a JavaScript program by setting a timer for a short interval and having the callback function make small updates to the graphical objects in the window.
- If the time interval is short enough (typically between 20 and 30 milliseconds), the animation will appear smooth to the human eye.

Timeouts

- JavaScript supports two kinds of timers. A ***one-shot timer*** invokes its callback function once after a specified delay. You create a one-shot timer by calling

  `setTimeout(`*function*, *delay***`);`**

  where *function* is the callback function and *delay* is the time interval in milliseconds.
- An ***interval timer*** invokes its callback function repeatedly at regular intervals. You create an interval timer by calling

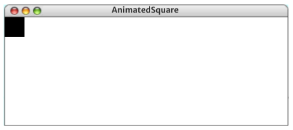
  `setInterval(`*function*, *delay***`);`**

  The **`setInterval`** function returns a numeric value that you can later use to stop the timer by calling **`clearTimeout`** with that numeric value as an argument.

A Simple Example of Animation

```
function AnimatedSquare() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS;
   var dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS;
   var square = GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
   square.setFilled(true);
   gw.add(square);
   var stepCount = 0;
   var step = function() . . .
   var timer = setInterval(step, TIME_STEP);
}
```

gw	dx	dy	square	stepCount	step	timer
	4.5	2.5		1	• • •	*id*

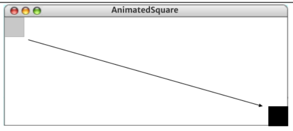


A Simple Example of Animation

```
function AnimatedSquare() {
```

```
function() {
   square.move(dx, dy);
   stepCount++;
   if (stepCount === N_STEPS) clearTimeout(timer);
}
```

gw	dx	dy	square	stepCount	step	timer
	4.5	2.5		100	• • •	*id*



The End