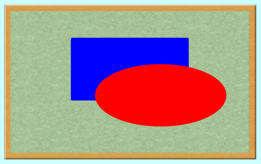
Interactive Graphics

Jerry Cain and Ryan Eberhardt
CS 106AJ
October 12, 2018
slides courtesy of Eric Roberts

Review: The Collage Model

- Our graphics library uses the same model we've used for the last decade, which is based on the metaphor of a ***collage***.
- A collage is similar to a child's felt board that serves as a backdrop for colored shapes that stick to the felt surface. As an example, the following diagram illustrates the process of adding a blue rectangle and a red oval to a felt board:



- Note that recently added objects can obscure those added prior. This layering arrangement is called the ***stacking order***.

The `GWindow` Class Revisited

The following expanded set of methods are available in the `GWindow` class:

`add(`*object*`)`	Adds the object to the canvas at the front of the stack
`add(`*object*`,` *x*`,` *y*`)`	Moves the object to (*x*, *y*) and then adds it to the canvas
`remove(`*object*`)`	Removes the object from the canvas
`removeAll()`	Removes all objects from the canvas
`getElementAt(`*x*`,` *y*`)`	Returns the frontmost object at (*x*, *y*), or `null` if none
`getWidth()`	Returns the width in pixels of the entire canvas
`getHeight()`	Returns the height in pixels of the entire canvas
`setBackground(`*c*`)`	Sets the background color of the canvas to *c*.

The Two Forms of the `add` Method

- The `add` method comes in two forms. The first is simply

```
add(object);
```

which adds the object at the location currently stored in its internal structure. You use this form when you have already set the coordinates of the object, which usually happens at the time you create it.

- The second form is

```
add(object, x, y);
```

which first moves the object to the point (*x*, *y*) and then adds it there. This form is useful when you need to determine some property of the object before you know where to put it.

Methods Common to All `GObject`s

`setLocation(`*x*`,`*y*`)`	Resets the location of the object to the specified point
`move(`*dx*`,` *dy*`)`	Moves the object *dx* and *dy* pixels from its current position
`movePolar(`*r*`,` *theta*`)`	Moves the object *r* pixel units in direction *theta*
`getX()`	Returns the *x* coordinate of the object
`getY()`	Returns the *y* coordinate of the object
`getWidth()`	Returns the horizontal width of the object in pixels
`getHeight()`	Returns the vertical height of the object in pixels
`contains(`*x*`,`*y*`)`	Returns `true` if the object contains the specified point
`setColor(`*c*`)`	Sets the color of the object to the `Color` *c*
`getColor()`	Returns the color currently assigned to the object
`scale(`*sf*`)`	Scales the shape by the scale factor *sf*
`rotate(`*theta*`)`	Rotates the shape counterclockwise by *theta* degrees
`sendToFront()`	Sends the object to the front of the stacking order
`sendToBack()`	Sends the object to the back of the stacking order
`sendForward()`	Sends the object forward one position in the stacking order
`sendBackward()`	Sends the object backward one position in the stacking order

Additional Methods for `GOval` and `GRect`

Fillable shapes (`GOval` and `GRect` [and later `GArc` and `GPolygon`])

`setFilled(`*flag*`)`	Sets the fill state for the object (`false`=outlined, `true`=filled)
`isFilled()`	Returns the fill state for the object
`setFillColor(`*c*`)`	Sets the color used to fill the interior of the object to *c*
`getFillColor()`	Returns the fill color

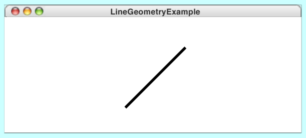
Resizable shapes (`GOval` and `GRect` [and later `GImage`])

`setSize(`*width*`,` *height*`)`	Sets the dimensions of the object as specified
`setBounds(`*x*`,` *y*`,` *width*`,` *height*`)`	Sets the location and dimensions together

Additional Methods for `GLine`

`setStartPoint(x, y)`	Sets the start point without changing the end point
`setEndPoint(x, y)`	Sets the end point without changing the start point

```
function LineGeometryExample() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let line = GLine(0, 0, 100, 100);
   gw.add(line);
   line.setLocation(200, 50);
   line.setStartPoint(200, 150);
   line.setEndPoint(300, 50);
}
```

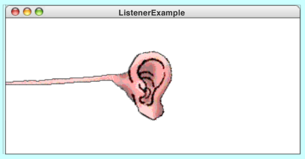


The JavaScript Event Model

- 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 have are treated like any data value are called ***first-class functions***.
- The textbook 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 lecture 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

```
let 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() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let clickAction = function(e) {
      let 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

```
let clickAction = function(e) {
   let 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 text 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.

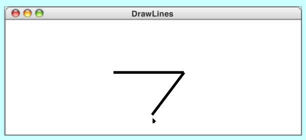
A Simple Line-Drawing Program

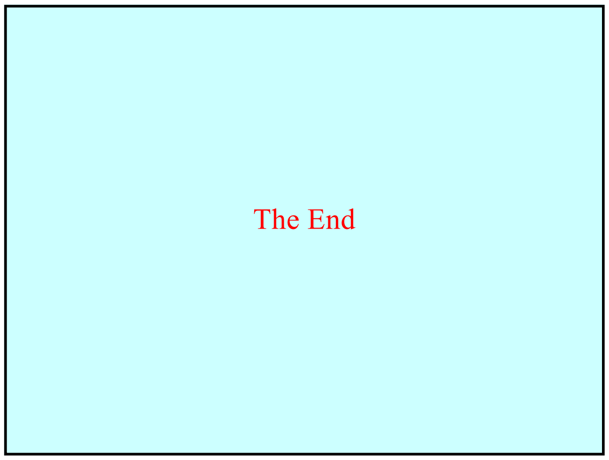
In all likelihood, you have at some point used an application that allows you to draw lines with the mouse. In JavaScript, the necessary code fits easily on a single slide.

```
const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   let line = null;

   let mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   let 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.




The End