

Computer Science Principles

Web Programming

JavaScript Programming Essentials

CHAPTER 10: INTERACTIVE PROGRAMMING

DR. ERIC CHOU

IEEE SENIOR MEMBER



Overview

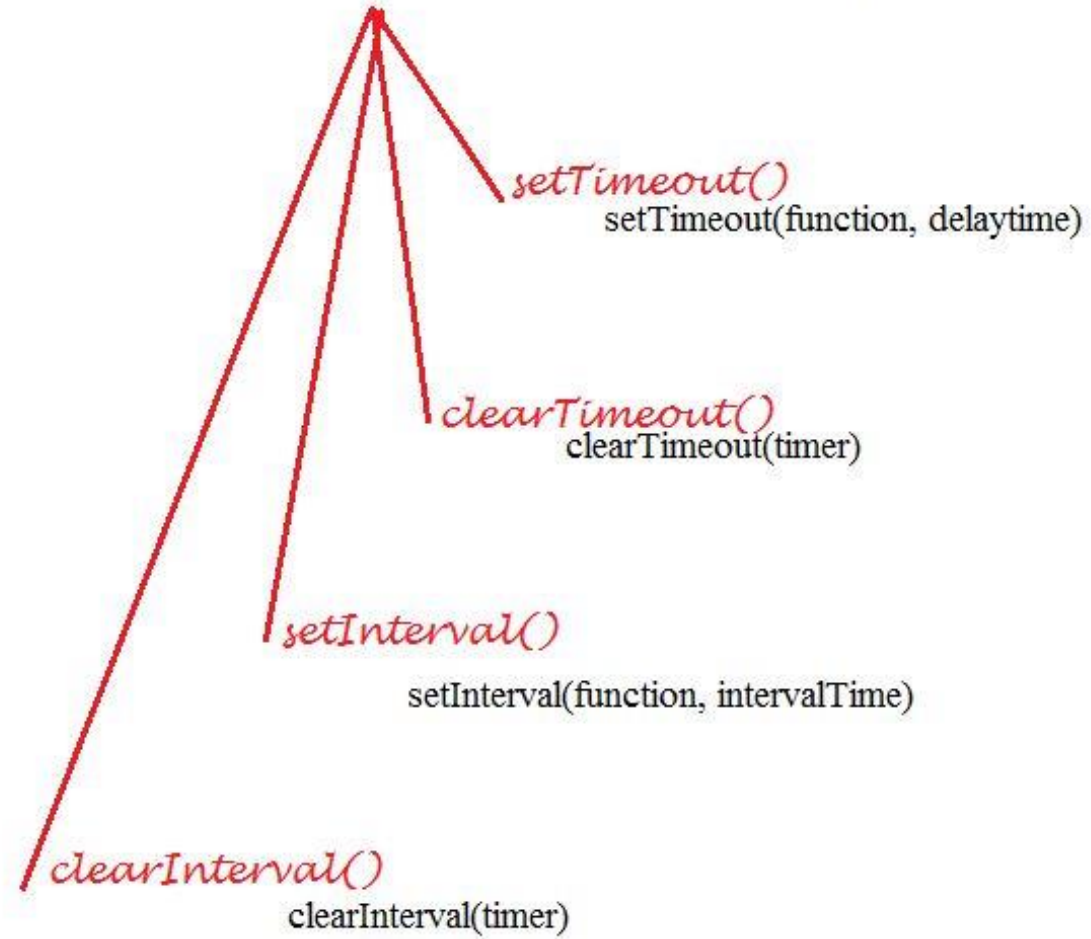
LECTURE 1



Overview: Interactive Programming

- Until now, the **JavaScript** code on our web pages has run as soon as the page is loaded, pausing only if we include a call to a function like `alert` or `confirm`. But we don't always necessarily want all of our code to run as soon as the page loads — what if we want some code to run after a delay or in response to something the user does?
- In this chapter, we'll look at different ways of modifying when our code is run. Programming in this way is called **interactive programming**. This will let us create interactive web pages that change over time and respond to actions by the user.

Timer Method in JavaScript





Time Out

LECTURE 1

setTimeout() {}

JAVASCRIPT SET TIMEOUT FUNCTION



Delaying Code with setTimeout

- Instead of having JavaScript execute a function immediately, you can tell it to execute a function after a certain period of time. Delaying a function like this is called setting a timeout. To set a timeout in JavaScript, we use the function **setTimeout**.
- This function takes two arguments (as shown in Figure 10-1): the function to call after the time has elapsed and the amount of time to wait (in milliseconds).

The function to call after
timeout milliseconds have passed



```
setTimeout(func, timeout)
```



The number of milliseconds to wait
before calling the function

Figure 10-1. The arguments for setTimeout



Delaying Code with setTimeout

- The following listing shows how we could use setTimeout to display an alert dialog.

```
❶ var timeUp = function () {  
  alert("Time's up!");  
};  
❷ setTimeout(timeUp, 3000);  
1
```

- At ❶ we create the function timeUp, which opens an alert dialog that displays the text "Time's up!". At ❷ we call setTimeout with two arguments: the function we want to call (timeUp) and the number of milliseconds (3000) to wait before calling that function. We're essentially saying, "Wait 3 seconds and then call timeUp."
- When setTimeout(timeUp, 3000) is first called, nothing happens, but after 3 seconds timeUp is called and the alert dialog pops up.



Delaying Code with setTimeout

- Notice that calling **setTimeout** returns 1. This return value is called the ***timeout ID***. The timeout ID is a number that's used to identify this particular timeout (that is, this particular delayed function call). The actual number returned could be any number, since it's just an identifier. Call **setTimeout** again, and it should return a different timeout ID, as shown here:

```
setTimeout(timeUp, 5000);
```

2

- You can use this timeout ID with the **clearTimeout** function to cancel that specific timeout. We'll look at that next.



Canceling a Timeout

- Once you've called **setTimeout** to set up a delayed function call, you may find that you don't actually want to call that function after all. For example, if you set an alarm to remind you to do your homework, but you end up doing your homework early, you'd want to cancel that alarm.
- To cancel a timeout, use the function **clearTimeout** on the timeout ID returned by **setTimeout**. For example, say we create a "do your homework" alarm like this:



Canceling a Timeout

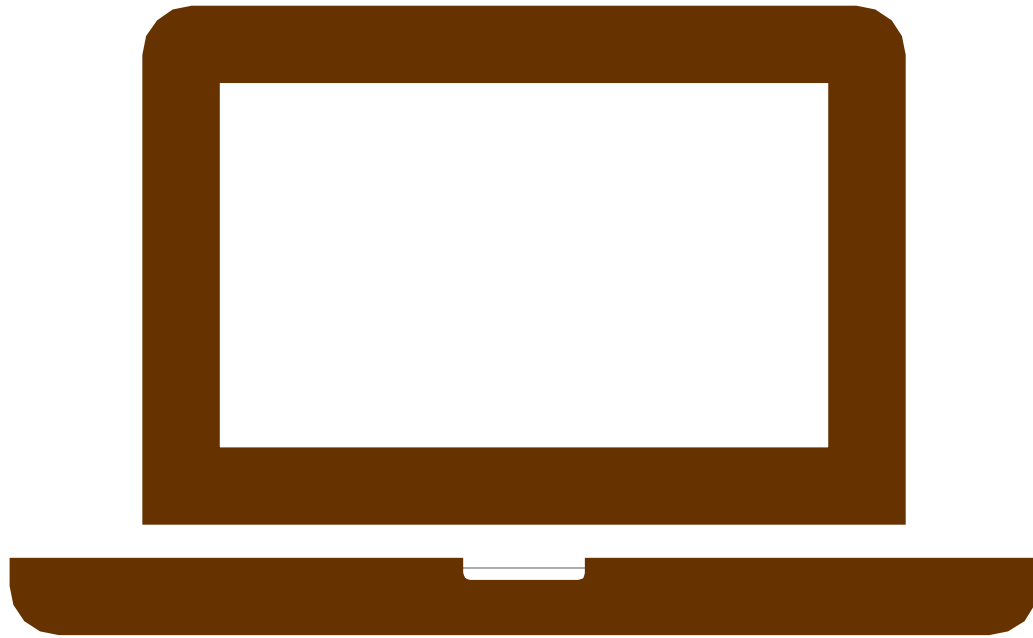
```
var doHomeworkAlarm = function () {  
  alert("Hey! You need to do your homework!");  
};  
❶ var timeoutId = setTimeout(doHomeworkAlarm, 60000);
```

The function **doHomeworkAlarm** pops up an alert dialog telling you to do your homework. When we call **setTimeout(doHomeworkAlarm, 60000)** we're telling JavaScript to execute that function after 60,000 milliseconds (or 60 seconds) has passed. At ❶ we make this call to **setTimeout** and save the timeout ID in a new variable called **timeoutId**.

To cancel the timeout, pass the timeout ID to the **clearTimeout** function like this:

```
clearTimeout(timeoutId);
```

Now **setTimeout** won't call the **doHomeworkAlarm** function after all.



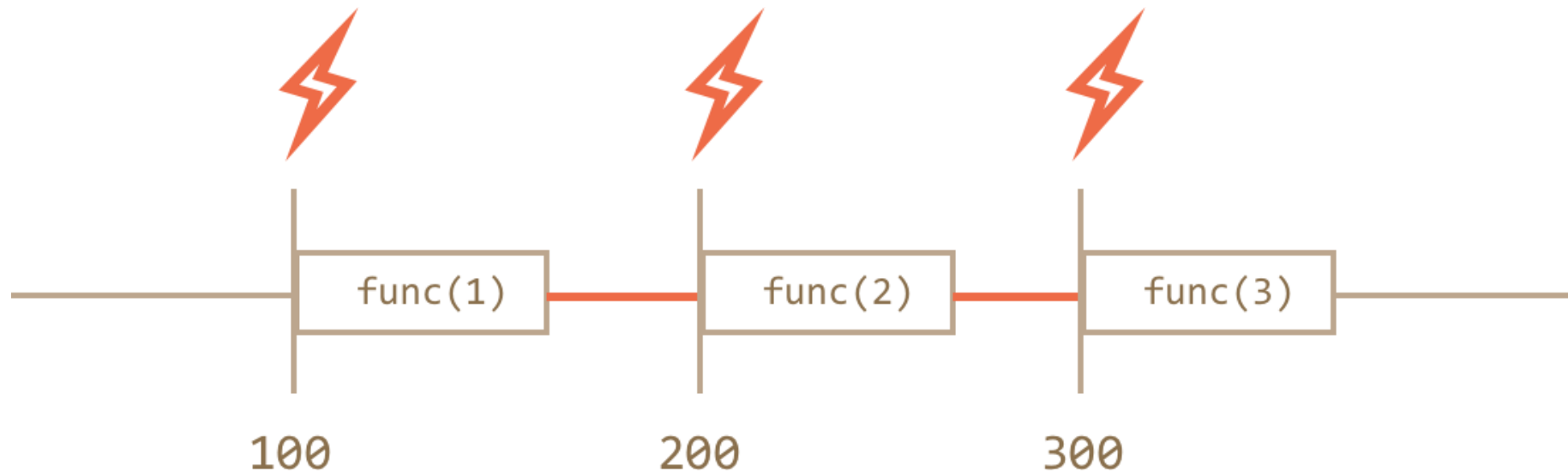
In-Class Demonstration Program

SET LONG TIMER



Flashing Text

Demo Program: flashing.html



- The time-out points will fire some function.
- We repeatedly fire `hide()` and `show()` will create flashing effect for text display.




Flashing Text Project 2:

Demo Program: `flashing2.html`

- The JavaScript section uses `addEventListener` instead of default event handler attachment by `onclick`.
- When **`addEventListener`** is used, it is safer to put the JavaScript section to the end of `<body>` section. Otherwise, the **`addEventListener`** may not work.
- An Object is return by the `blink` function. The returned object is actually the blinking controller. The blinking object is a blinking controller. There are two functions: `start()` and `stop()`. These two functions are added to the Event Handlers of the buttons.

```
cycle_time = 1000;
function blink(element, time) {
  function loop(){
    element.style.visibility = "hidden";
    setTimeout(function () {
      element.style.visibility = "visible";
    }, time);
    timer = setTimeout(function () {
      loop();
    }, time * 2);
    cleared = false;
  }
}
```



Recursion


```
function blink(element, time) {
```

```
  function loop(){
    element.style.visibility = "hidden";
    setTimeout(function () {
      element.style.visibility = "visible";
    }, time);
    timer = setTimeout(function () {
      loop();
    }, time * 2);
    cleared = false;
  }
```

```
var timer, cleared = true;
```

```
// expose methods
```

```
return {
```

```
  start: function() {
    if (cleared) loop();
  },
```

```
  stop: function() {
    clearTimeout(timer);
    cleared = true;
  }
};
```

```
};
```

```
}
```

Emergency

start blinking

stop blinking

```
<div id="blink" style="color:red; font-size:48px">Emergency</div>
<button id="start">start blinking</button><br />
<button id="stop">stop blinking</button>
```

```
var blinking = blink(document.getElementById("blink"),
  cycle_time/2);
```

```
document.getElementById("start").addEventListener("click",
  function(){
    blinking.start();
  });
```

```
document.getElementById("stop").addEventListener("click",
  function(){
    blinking.stop();
  });
```



Intervals

LECTURE 1



Calling Code Multiple Times with setInterval

The `setInterval` function is like **`setTimeout`**, except that it **repeatedly** calls the supplied function after regular pauses, or intervals. For example, if you wanted to update a clock display using JavaScript, you could use **`setInterval`** to call an update function every second. You call **`setInterval`** with two arguments: the function you want to call and the length of the interval (in milliseconds), as shown in Figure 10-2.



Calling Code Multiple Times with setInterval

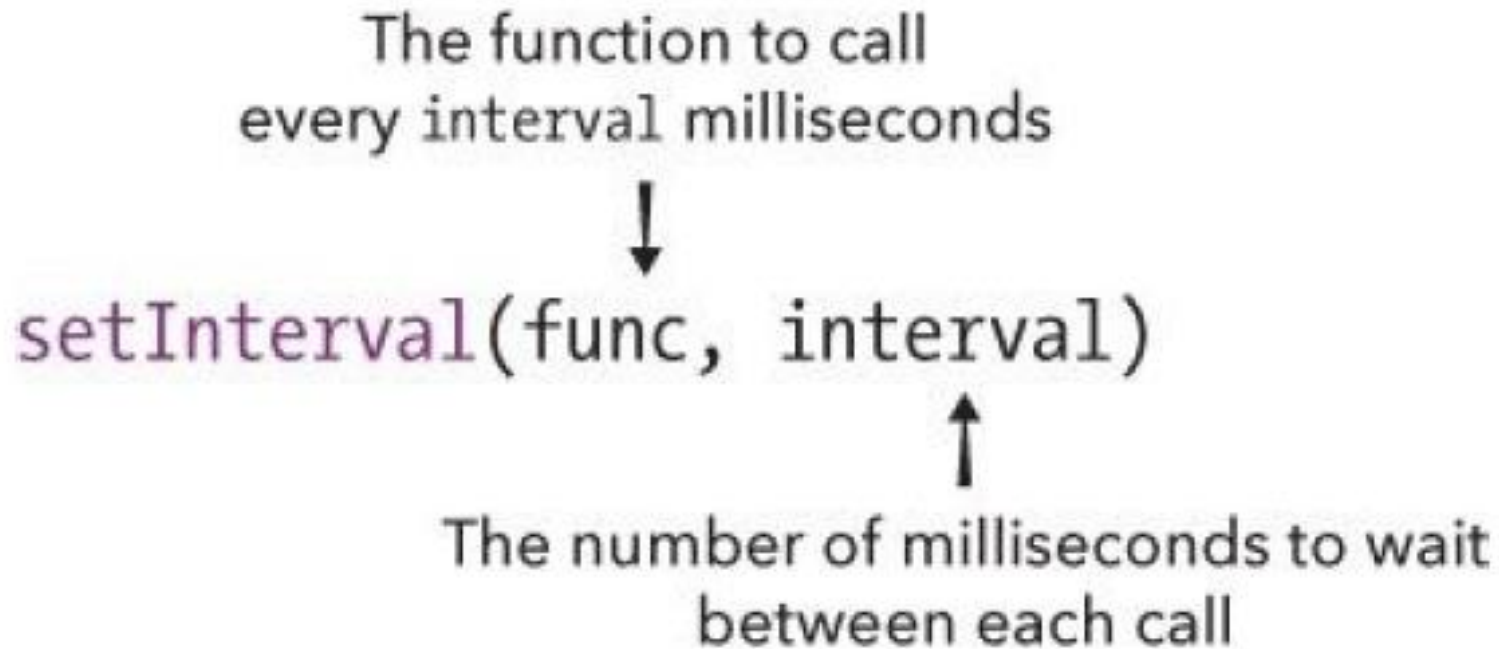


Figure 10-2. The arguments for `setInterval`



Calling Code Multiple Times with setInterval

Here's how we could write a message to the console every second:

```
❶ var counter = 1;  
❷ var printMessage = function () {  
  console.log("You have been staring at your console for " + counter  
  + " seconds");  
❸ counter++;  
};  
❹ var intervalId = setInterval(printMessage, 1000);  
You have been staring at your console for 1 seconds  
You have been staring at your console for 2 seconds  
You have been staring at your console for 3 seconds  
You have been staring at your console for 4 seconds  
You have been staring at your console for 5 seconds  
You have been staring at your console for 6 seconds  
❺ clearInterval(intervalId);
```



Calling Code Multiple Times with setInterval

At ❶ we create a new variable called counter and set it to 1. We'll be using this variable to keep track of the number of seconds you've been looking at your console.

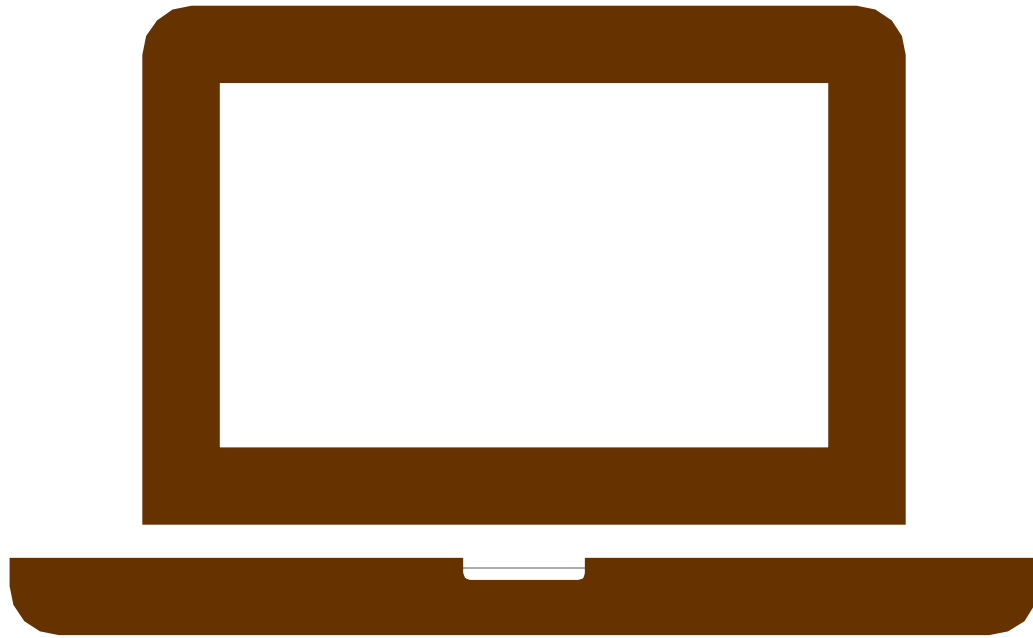
At ❷ we create a function called printMessage. This function does two things. First, it prints out a message telling you how long you have been staring at your console. Then, at ❸, it increments the counter variable.

Next, at ❹, we call setInterval, passing the printMessage function and the number 1000. Calling setInterval like this means "call printMessage every 1,000 milliseconds." Just as setTimeout returns a timeout ID, setInterval returns an *interval ID*, which we save in the variable intervalId.

We can use this interval ID to tell JavaScript to stop executing the printMessage function. This is what we do at ❺, using the clearInterval function.

Demo Program: multiple.html

```
1 ▼ <html>
2 ▼ <body>
3 ▼ <script>
4   var counter = 1;
5 ▼ var printMessage = function () {
6     document.write("You have been staring at your console for " +
7         counter + " seconds"+"<br>");
8     counter++;
9     if (counter > 20) clearInterval(intervalId);
10  };
11  var intervalId = setInterval(printMessage, 1000);
12
13  </script>
14  </body>
15  </html>
```



In-Class Demonstration Program

MULTIPLE FUNCTION CALL



JavaScript Game Development

Overview

- Web is a convenient platform for a quick start in game development, especially for those of you who are familiar with JavaScript language. This is a feature-rich platform in content rendering and input processing from different resources.
- First steps in game development for Web:
 - Sort out the game loop and rendering definitions.
 - Learn to process user input.
 - Create the main scene prototype of the game.
 - Add the rest of the game scenes.



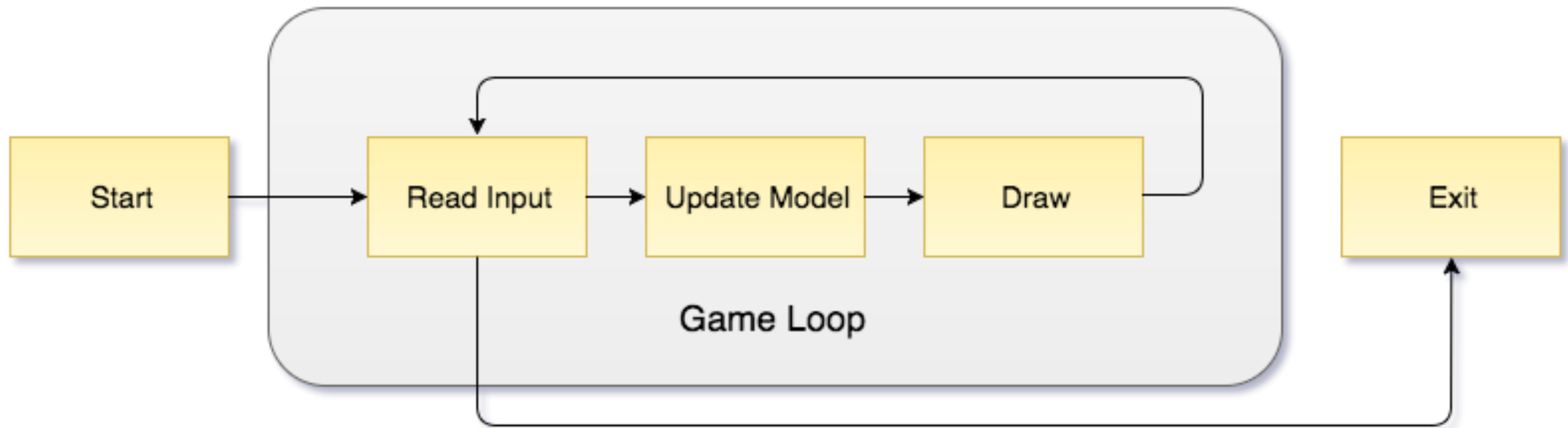
Game Loop

- Game cycle is a heart of the game, where user input and game logic processing occur along with current game state rendering. Schematically, game loop can be described in the following way:
- As a code, the simplest implementation of the game cycle in JavaScript may look like this:

```
// game loop
setInterval(() => {
  update();
  render();
}, 1000 / 60);
```



Use JavaScript setInterval for Game Loop





Game Loop

- Update() function is responsible for the game process logic and updating of game state, depending on the user input. Wherein, it absolutely doesn't matter, with the help of which technologies the rendering itself is happening (Canvas, DOM, SVG, console etc.).



Game Loop

Please note that browser environment imposes certain restrictions on the work of this code:

- Stable timer interval is not guaranteed, which means that the game process may occur with **different** speed.
- The timer in inactive browser tabs may be stopped or repeatedly launched, when the tab is activated, which may lead to strange behavior of the game.
- setInterval is already outdated for such tasks. Today it is recommended to use **requestAnimationFrame** method, because it allows to achieve better productivity and decrease energy consumption.
- Please check for our Game Design Chapter.



Animation

LECTURE 1



Animating Elements with setInterval

- As it turns out, we can use **setInterval** to animate elements in a browser. Basically, we need to create a function that moves an element by a small amount, and then pass that function to **setInterval** with a short interval time.
- If we make the movements small enough and the interval short enough, the animation will look very smooth.



Animating Elements with setInterval

Let's animate the position of some text in an HTML document by moving the text horizontally in the browser window. Create a document called *interactive.html*, and fill it with this HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Interactive programming</title>
  </head>
  <body>
    <h1 id="heading">Hello world!</h1>
    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
    <script>
      // We'll fill this in next
    </script>
  </body>
</html>
```




Animating Elements with setInterval

Now let's look at the JavaScript. As always, put your code inside the <script> tags of the HTML document.

```
❶ var leftOffset = 0;
❷ var moveHeading = function () {
❸     $("#heading").offset({ left: leftOffset });
❹     leftOffset++;
❺     if (leftOffset > 200) {
        leftOffset = 0;
    }
};
❻ setInterval(moveHeading, 30);
```



Animating Elements with setInterval

- When you open this page, you should see the heading element gradually move across the screen until it travels 200 pixels; at that point, it will jump back to the beginning and start again. Let's see how this works.
- At ❶ we create the variable **leftOffset**, which we'll use later to position our Hello world! heading. It starts with a value of 0, which means the heading will start on the far left side of the page.
- Next, at ❷, we create the function **moveHeading**, which we'll call later with setInterval. Inside the **moveHeading** function, at ❸, we use `$("#heading")` to select the element with the id of "heading" (our h1 element) and use the offset method to set the left offset of the heading — that is, how far it is from the left side of the screen.



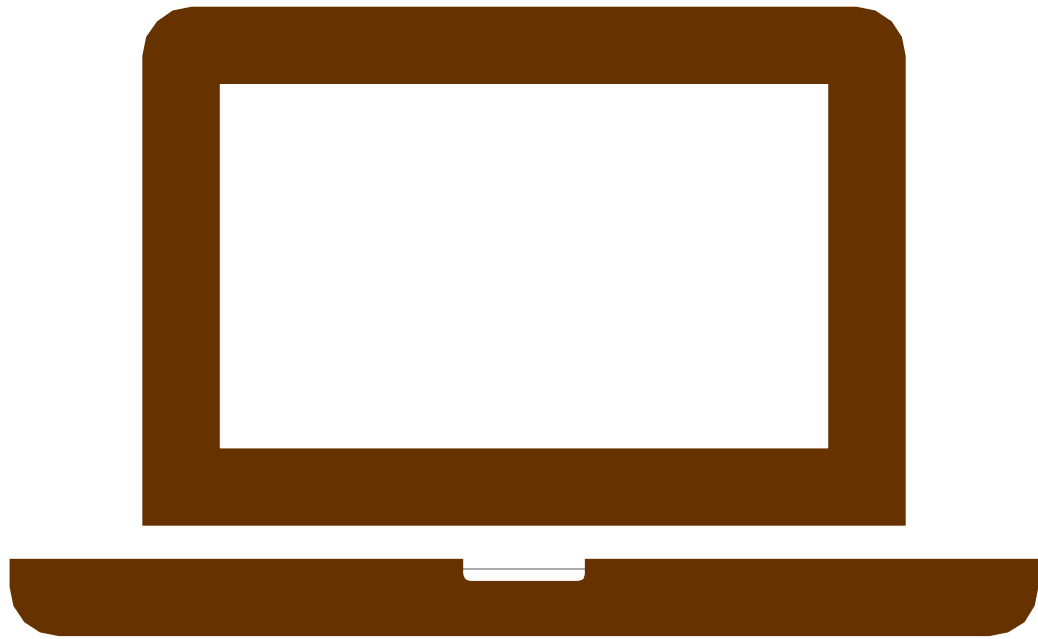
Animating Elements with setInterval

- The offset method takes an object that can contain a left property, which sets the left offset of the element, or a top property, which sets the top offset of the element. In this example we use the left property and set it to our leftOffset variable. If we wanted a static offset (that is, an offset that doesn't change), we could set the property to a numeric value. For example, calling `$("#heading").offset({ left: 100 })` would place the heading element 100 pixels from the left side of the page.
- At ④ we increment the leftOffset variable by 1. To make sure the heading doesn't move too far, at ⑤ we check to see if leftOffset is greater than 200, and if it is, we reset it to 0. Finally, at ⑥ we call setInterval, and for its arguments we pass in the function moveHeading and the number 30 (for 30 milliseconds).



Animating Elements with setInterval

- This code calls the **moveHeading** function every 30 milliseconds, or about 33 times every second. Each time **moveHeading** is called, the **leftOffset** variable is incremented, and the value of this variable is used to set the position of the heading element.
- Because the function is constantly being called and **leftOffset** is incremented by 1 each time, the heading gradually moves across the screen by 1 pixel every 30 milliseconds.



Demonstration Program

ANIMATION BE SETINTERVAL
(ANIMATION0.HTML)



Interaction

LECTURE 1



Responding to User Actions

- As you've seen, one way to control when code is run is with the **functions `setTimeout` and `setInterval`**, which run a function once a fixed amount of time has passed.
- Another way is to run code only when a user performs certain actions, such as clicking, typing, or even just moving the mouse.
- This will let users interact with your web page so that your page responds according to what they do.



Responding to User Actions

- In a browser, every time you perform an action such as clicking, typing, or moving your mouse, something called an *event* is triggered. An event is the browser's way of saying, "This thing happened!"
- You can listen to these events by adding an *event handler* to the element where the event happened.
- Adding an event handler is your way of telling JavaScript, "If this event happens on this element, call this function." For example, if you want a function to be called when the user clicks a heading element, you could add a click event handler to the heading element. We'll look at how to do that next.



Responding to Clicks

- When a user clicks an element in the browser, this triggers a *click event*. jQuery makes it easy to add a handler for a click event. Open the *interactive.html* document you created earlier, use **File ▶ Save As** to save it as *clicks.html*, and replace its second script element with this code:

```
❶ var clickHandler = function (event) {  
❷   console.log("Click! " + event.pageX + " " + event.pageY);  
   };  
❸ $("h1").click(clickHandler);
```



Responding to Clicks

- At ❶ we create the function **clickHandler** with the single argument `event`. When this function is called, the event argument will be an object holding information about the click event, such as the location of the click. At ❷, inside the handler function, we use `console.log` to output the properties **pageX** and **pageY** from the event object. These properties tell us the event's x- and y-coordinates — in other words, they say where on the page the click occurred.
- Finally, at ❸ we activate the click handler. The code `$("#h1")` selects the `h1` element, and calling `$("#h1").click(clickHandler)` means “When there is a click on the `h1` element, call the **clickHandler** function and pass it the event object.” In this case, the click handler retrieves information from the event object to output the x- and y-coordinates of the click location.



Responding to Clicks

- Reload your modified page in your browser and click the heading element. Each time you click the heading, a new line should be output to the console, as shown in the following listing. Each line shows two numbers: the x- and y-coordinates of the clicked location.

```
Click! 88 43
```

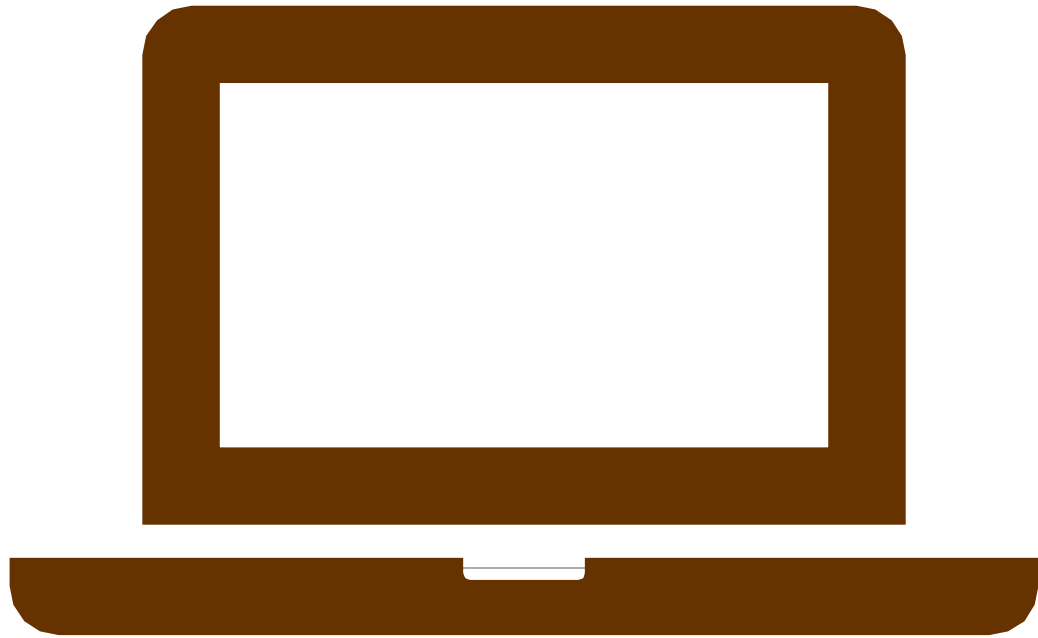
```
Click! 63 53
```

```
Click! 24 53
```

```
Click! 121 46
```

```
Click! 93 55
```

```
Click! 103 48
```



Demonstration Program

CLICKS.HTML

Browser Coordinates

- In the web browser and in most programming and graphics environments, the 0 position of the x- and y-coordinates is at the top-left corner of the screen. As the x-coordinate increases, you move right across the page, and as the y-coordinate increases, you move down the page (see Figure 10-3).

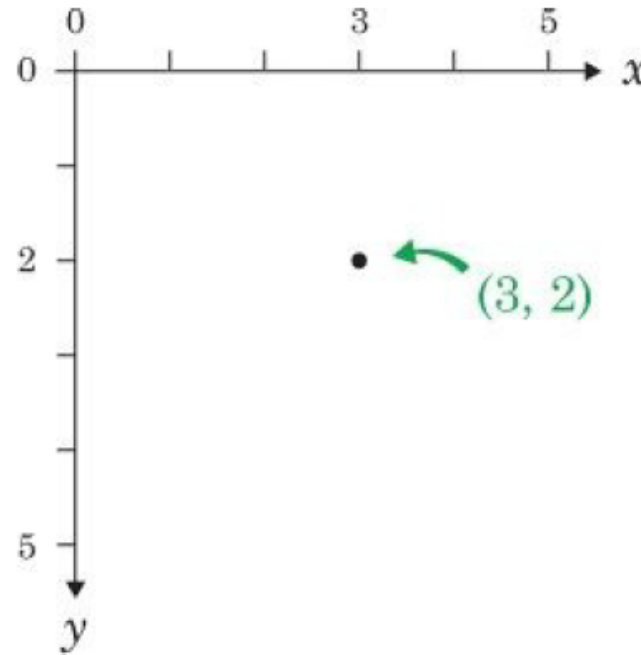


Figure 10-3. Coordinates in the browser, showing a click at the coordinate (3, 2)



The mousemove Event

The mousemove event is triggered every time the mouse moves. To try it out, create a file called *mousemove.html* and enter this code:

```
<!DOCTYPE html>
<html>
<head>
<title>Mousemove</title>
</head>
<body>
<h1 id="heading">Hello world!</h1>
<script src="https://code.jquery.com/jquery-2.1.0.js"></script>
<script>
❶ $("html").mousemove(function (event) {
❷ $("#heading").offset({
left: event.pageX,
top: event.pageY
});
});
</script>
</body>
</html>
```



The mousemove Event

At ❶ we add a handler for the mousemove event using `$("#html").mousemove(handler)`. In this case, the *handler* is the entire function that appears after mousemove and before `</script>`. We use `$("#html")` to select the html element so that the handler is triggered by mouse movements that occur anywhere on the page. The function that we pass into the parentheses after mousemove will be called every time the user moves the mouse.



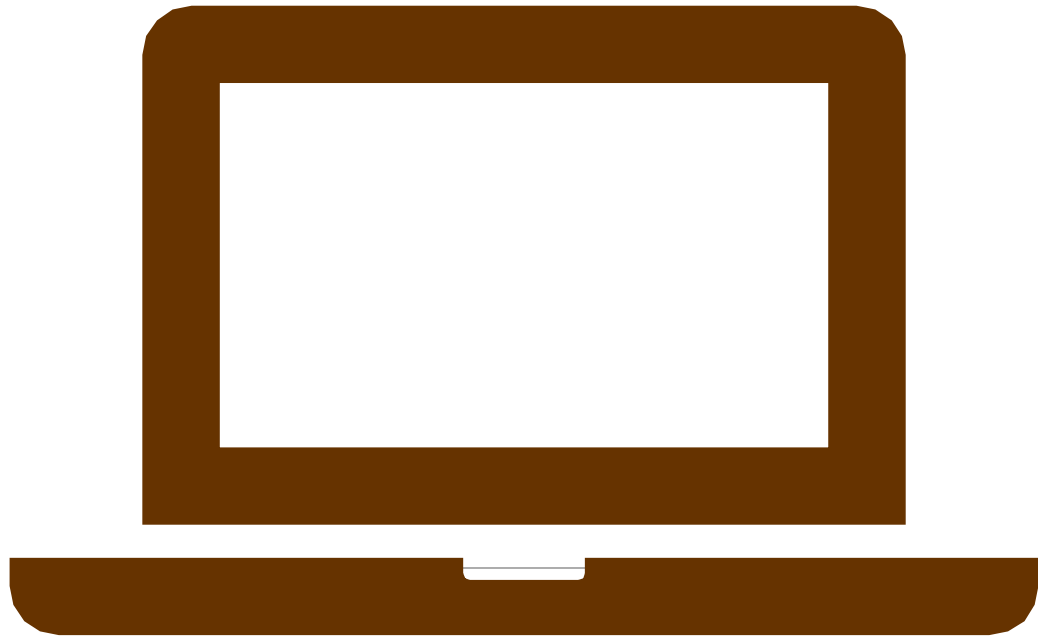
The mousemove Event

In this example, instead of creating the event handler separately and passing the function name to the mousemove method (as we did with our clickHandler function earlier), we're passing the handler function directly to the mousemove method. This is a very common way of writing event handlers, so it's good to be familiar with this type of syntax.



The mousemove Event

At ❷, inside the event handler function, we select the heading element and call the offset method on it. As I mentioned before, the object passed to offset can have left and top properties. In this case, we set the left property to `event.pageX` and the top property to `event.pageY`. Now, every time the mouse moves, the heading will move to that location. In other words, wherever you move the mouse, the heading follows it!



Demonstration Program

MOUSEMOVE.HTML

```
1  <!DOCTYPE html>
2  ▼ <html>
3  ▼   <head>
4     <title>Mousemove</title>
5     </head>
6  ▼   <body>
7     <h1 id="heading">Hello world!</h1>
8     <script src="https://code.jquery.com/jquery-3.4.1.js"></script>
9  ▼   <script>
10 ▼       $("html").mousemove(function (event) {
11 ▼           $("#heading").offset({
12               left: event.pageX,
13               top: event.pageY
14           });
15       });
16   </script>
17   </body>
18 </html>
```