To introduce this module, I'm going to tell a story.

This story is going to explain what asynchronous programming, callback

functions, and events are all about, and why they are important.

This is the story of the events loop.

So, what is the Event Loop?

The JavaScript runtime executes code in a single thread.

Meaning that it can only one statement of code at a time.

Code is normally placed on the call stack before being executed.

The call stack is a segment of memory that keeps track of the order

of functions in which they were called.

The events loop, is the process in which the browser queues up tasks

and executes them, one at a time, by putting them on the call stack.

Now how does the browser queue up tasks?

In the beginning of an application, all the JavaScript code is run til

completion and placed in the call stack in order of execution.

The event que is initially empty.

As events occur,

event handlers place new tasks onto the event queue.

Some examples of these events are mouse clicks, keyboard presses,

and timed events.

These tasks waits on the events queue until the call stack is empty.

Once the call stack is empty,

the first task in the queue is put on to the stack.

The subsequent tasks wait until the call stack is empty again,

and the cycle repeats.

This cycle is the events loop

Now how do asynchronous functions fit into the event loop?

When the JavaScript runtime comes across an asynchronous

function in the call stack, it does not process it immediately.

Instead of blocking the call stack until it is finished, it allows

another process to handle processing of the asynchronous function.

When the other process has finished,

it adds a task back on to the event queue.

This task Is usually a callback function, passed

in as one of the arguments, to the original asynchronous function.

You are probably thinking to yourselves,

didn't you just say JavaScript was single-threaded?

Well, the JavaScript runtime is single-threaded, but there are other

processes running in a browser, such as timers, input handlers,

and network request APIs that run in parallel with the event loop.

These parallel processes communicates with the event loop

by putting new tasks on to the event queue.

So why are asynchronous functions important?

They are important because some tasks, such as network request

are slow and users will become upset if their browsers freeze up

because of slow network request blocked other code from running.

Asynchronous requests allow slow tasks to be operated

on a separate thread to stop a task from blocking the browser.

Call backs are used to specify what new tasks are sent back to the event

queue once the other process has finished processing

the original task.

And that is the story of the event loop.

**Intro to Asynchronous Programming Fundamentals**

**What is Asynchronous Programming?**

Asynchronous programming is about processing code on a separate thread and then handling the result once it is done.

**Why is Asynchronous Programming Important?**

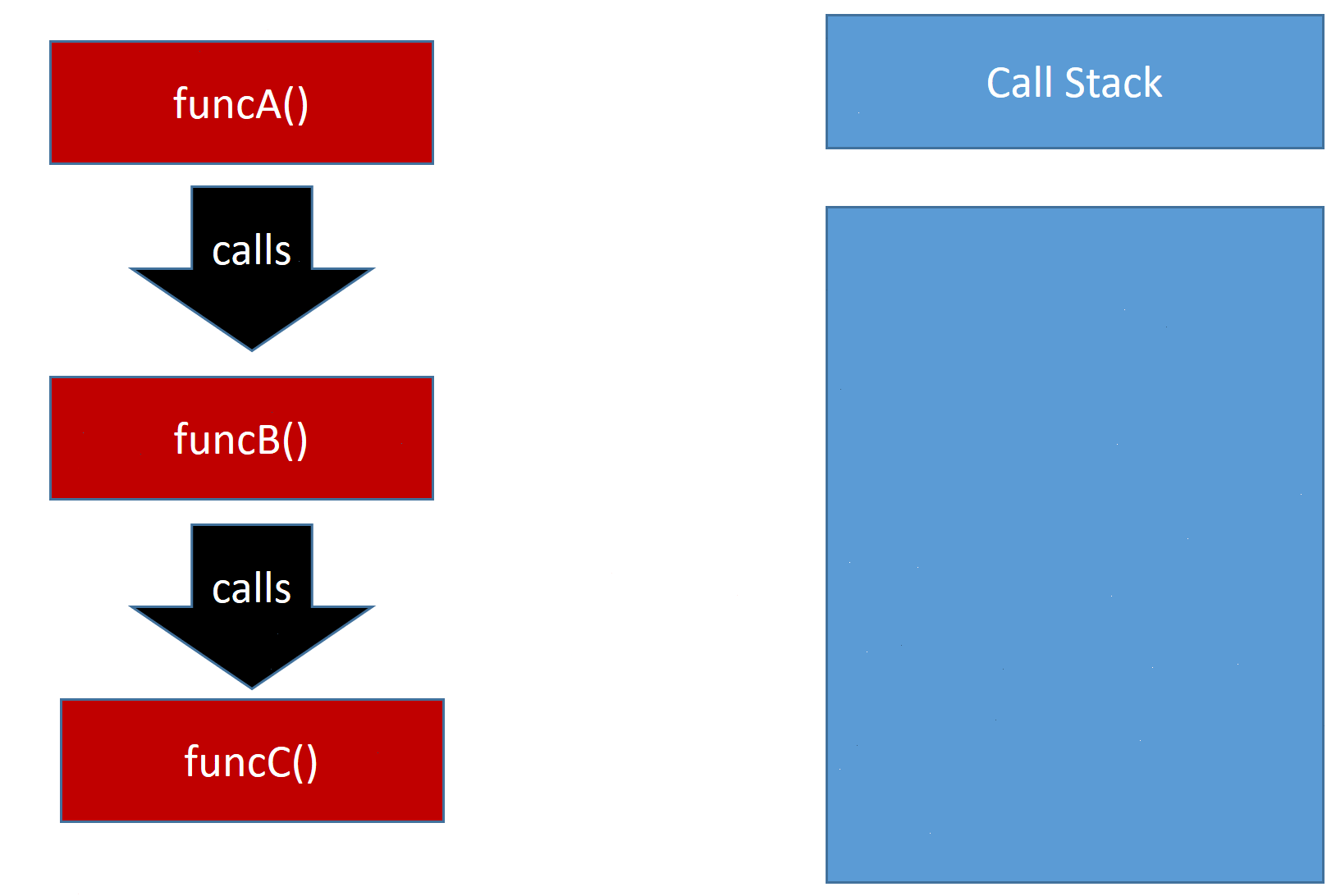
Asynchronous Programming prevents slow tasks from blocking faster tasks from running. This allows applications to run without frustrating users by stalling.

**What are we going to learn?**

* How  asynchronous code differs from synchronous code
* How to user Callback Functions
* How to use Timers
* How to handle DOM Events

**The Call Stack**

The call stack tracks functions that are currently active and are being processed.



The call stack functions in the following way:

* When a function call is encountered, it is pushed onto the call stack.
* Any additional functions called within the original function are placed higher up onto the call stack.
* When a function finishes executing, it is popped off the call stack and the next function on the call stack is processed.

Notice how the call stack keeps track of the order of function calls:

function funcA(){

funcB();

}

function funcB(){

funcC();

}

function funcC(){

console.log(Error().stack); //Error is only used to show the call stack

}

funcA();

/\*console output

"Error

at funcC (example.js:15:17) <-- funcC is at the top of the callstack because it was called last

at funcB (example.js:12:5)

at funcA (example.js:9:5) <-- funcA is at the bottom of the call stack because it was called first

at example.js:17:1"

\*/

**Stack Overflow**

If the call stack grows too large and exceeds the amount of memory allocated, a Stack Overflow Error will occur. This commonly happens when a function calls itself recursively.

Notice how a stack overflow error can occur:

function funcA(){

funcA();

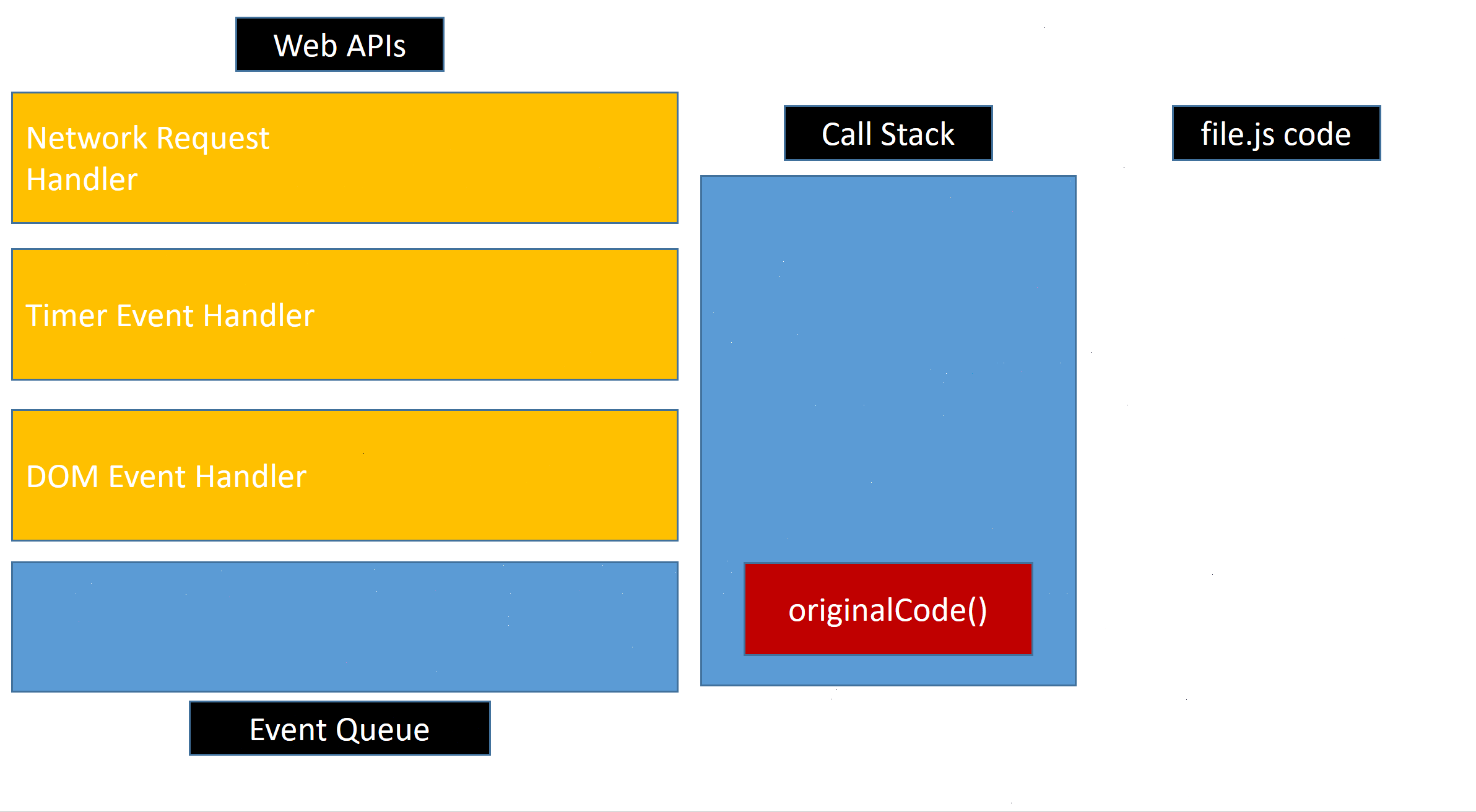
}

funcA();

//causes at stack overflow error because funcA() keeps getting called recursively

**The Event Queue**

The Event Queue is a queue that keeps track of tasks that are waiting to be put on the call stack to be executed. Tasks get added to the Event Queue by Web APIs that run in parallel with the JavaScript run time.



Here are three examples of Web APIs that add tasks to the event queue:

* Timers - Timers schedule tasks to be added to the event queue
* DOM Event Handlers - User Interactions such as mouse clicks and keyboard presses are handled by putting tasks on the event queue
* Network Requests - Network requests are processed asynchronously and send back results by putting tasks on the event queue

**The Event Loop**

When the call stack is empty, it takes the first task off the event queue and processes it. The remaining tasks on the queue wait until the call stack is empty again. This cycle is called the Event Loop.

**Asynchronous Programming**

Asynchronous Programming is achieved in JavaScript by using Web APIs that process code on separate threads. The Web API's send their processed results back as tasks on the event queue. These tasks are defined by callback functions passed into the Web APIs. This allows JavaScript to achieve multi-threading in a single threaded run time.

**Synchronous vs Asynchronous Programming**

Imagine trying to run a slow task synchronously. It will take a long time to finish processing and will prevent other tasks from running.

Notice how the **slowTask()**takes a long time to process and prevents other fast tasks from processing:

function slowTask(){

/\*takes 2 seconds to process\*/

var now = new Date().getTime();

while(new Date().getTime() < now + 2000){ /\* processing \*/ }

console.log("slow task finished");

}

function fastTask(){

console.log("fast task finished")

}

fastTask();

slowTask()

fastTask();

fastTask();

fastTask();

/\* Console Output:

> "fast task finished"

....2 seconds later

> "slow task finished"

> "fast task finished"

> "fast task finished"

> "fast task finished"

\*/

Asynchronous programming is great because it prevents slow tasks from blocking faster tasks from processing. Asynchronous code will only run when the call stack is empty.

Notice how asynchronous code prevents slow tasks from blocking other faster tasks:

function slowTask(){

console.log("slow task finished");

}

function asyncSlowTask(val){

setTimeout(slowTask,2000); //finishes in two seconds, but is processed on a separate thread

}

function fastTask(){

console.log("fast task finished!")

}

fastTask();

asyncSlowTask();

fastTask();

asyncSlowTask();

fastTask();

fastTask();

/\* Console Output:

> "fast task finished" <--faster tasks were processed first

> "fast task finished"

> "fast task finished"

> "fast task finished"

> "slow task finished" <--slow tasks were processed separately and didn't block the call stack

> "slow task finished"

\*/

**Demo Examples**

function funA() {

funB();

}

function funB() {

funC();

}

function funC() {

console.log(Error().stack);

}

funA();

**Callback Functions**

Callback functions are functions that are passed as arguments into other functions to be executed at a later point in time.

Notice how callback functions are used:

//multiplies two numbers

function mult(x,y){

return x \* y;

}

//adds to numbers

function add(x,y){

return x + y;

}

//uses a callback to process two numbers

function calculate(x,y,compute){

return compute(x,y);

}

var a = calculate(10,5,add); //uses add callback

console.log(a); // logs 15

var b = calculate(10,5,mult); //uses mult callback

console.log(b); // logs 50

**Anonymous Callbacks**

Callbacks can be created and used without being bound to a specific function name. Anonymous callbacks are useful when a callback is only needed to be declared once, since they are quicker to write than named callbacks.

Notice how an anonymous callback is used:

var c = calculate(10,5,function(x,y){ //uses an anonymous callback

return x - y; //subtracts y from x

});

console.log(c); // logs 5

Notice how an anonymous callback is used with arrow functions:

var d = calculate(10,5, (x,y) => {return x - y}); //using arrow functions

console.log(d); // logs 5

**Callback Examples**

Several JavaScript functions already take in callbacks as arguments.

**map()**

The **map()** method calls a callback function on each element in the array and then returns a new array with those results.

Notice how the **map()** method uses a callback function:

var array = [1,2,3,4,5];

var newArray = array.map(function(x){ //uses an anonymous callback function to square each element

return x \* x;

});

console.log(newArray);

// logs [1,4,9,16,25]

**filter()**

The **filter()** method removes elements in an array that do not pass a certain criteria defined by a callback function.

Notice how the **filter()** method is used to remove elements in an array that are not even:

var array = [1,2,3,4,5];

function isEven(x){ //checks if a value is even

return x % 2 == 0;

}

var newArray = array.filter(isEven); //uses a callback to check if an element is even

console.log(newArray);

// logs [2,4]

**Chaining Callbacks with Continuation Passing Style**

The Continuation Passing Style(CPS) is a programming style used to chain callback functions together. In CPS, methods with callback functions as arguments are called within other callback functions. CPS is characterized by having methods that have callback functions as their last argument.

Notice how callbacks can be chained with the Continuation Passing Style:

function myFunction(x,callback){

callback(x);

}

var answer = 0;

myFunction(10,function(x){ //callback1

var result = x \* x; //result = 100

myFunction(result, function(x){ //callback2 within callback 1

var result2 = x + x; //result2 = 200

myFunction(result2, function(x){ //callback 3 within callback 2

answer = x + 100;

console.log(answer); // logs 300

})

})

});

CPS has a tendency to become difficult to manage as more and more callback functions are chained together. We will cover better methods to chain callbacks together later in this course.

**Timers**

**SetTimeout()**

The **setTimeout()** method is used to schedule a task to be put on the event queue after a given amount of time. The first parameter to **setTimeout()** is the callback function that is going to be executed. The second parameter is the amount of time to wait before putting the task on the event queue. **setTimeout()**is non-blocking and other code may run while the **setTimeout()**task is waiting to be executed.

Notice how setTimeout is used to schedule a console log after 1000 milliseconds:

setTimeout(function(){

console.log("hello")

},1000); //waits 1 second

/\* Console Output:

> "hello" <--after 1 second

/\*

**clearTimeout()**

The **clearTimeout()** function is used to cancel a timeout that is still pending. The **setTimeout()**method call returns a numeric timerID that is used to identify the timer. This timerID can be passed into the **clearTimeout()**method call to stop the timer.

Notice how **clearTimeout()** is used to stop a **setTimeout()** callback from executing:

var timeout = setTimeout(function(){

console.log("hello")

},1000); //waits 1 second

clearTimeout(timeout); //clears the setTimeout callback from running

//nothing gets logged

**SetInterval()**

The **setInterval()** method is used to schedule a reoccurring task to be put on the event queue every time a given number of milliseconds elapses. The first parameter to **setInterval()** is the callback function that is going to be executed. The second parameter is the amount of time to wait before the reoccurring task is put back on to the event queue.

Notice how the **setInterval()** method is used to log a number every second:

var count = 0;

var interval = setInterval(function(){

count++;

console.log(count);

},1000); //executes callback every second

/\* Console Output:

> 1 <-- after 1 second

> 2 <-- after 2 seconds

> 3 <-- after 3 seconds

> 4 <-- after 4 seconds

... <-- interval continues until stopped

\*/

**ClearInterval()**

The **clearInterval()** method is used to stop an interval timer set by **setInterval()**. The **setInterval()**method call returns a numeric timerID that is used to identify the interval timer. This timerID can be passed into the **clearInterval()** method call to stop the interval timer.

Notice how **clearInterval()** is used to stop an interval from continuing after it executes three times:

var count = 0;

var interval = setInterval(function(){

count++;

console.log(count);

if(count >= 3){

clearInterval(interval); //clears the interval after it is called 3 times

}

},1000); //executes callback every second

/\*Console Output

>1 <--after 1 second

>2 <--after 2 seconds

>3 <--after 3 seconds

\*/

**Asynchronous Code Using Timers**

Synchronous code is run line by line in the order in which the code occurred.

Notice how synchronous code is executed:

console.log("first");

console.log("second");

console.log("third");

/\* Console Output:

> first

> second

> third

\*/

Asynchronous code may be executed in a different order than how it originally occurred. Asynchronous code is non-blocking and will only run when the call stack is empty.

Asynchronous code can be shown by using a **setTimeout()**method call with a timeout value of 0. This will immediately put a task on the event queue.

Notice how "second" is logged asynchronously and occurs out of order:

function asyncLog(val){ //logs values asynchronously

setTimeout(function(){ //setTimeout with a time of 0 will execute asynchronously

console.log(val);

},0)

}

console.log("first");

asyncLog("second");

console.log("third");

/\* Console Output

> first

> third <---notice this is out of order!!

> second <---this occurs only after the call stack is empty, which is why it appears last

\*/

The output appears out of order because the asynchronous console log task had to wait for the call stack to finish executing the other console logs before it could occur.

#### ****DOM Events****

DOM Event Listeners happen in parallel with the JavaScript run time. When an event occurs, the event listener detects the event and executes an event handler to put a task on the event queue. The task will eventually make its way to the call stack to be executed.

If multiple events are detected, multiple tasks will be put on the event queue in the order in which they occurred. When the call stack is empty, the first task on the event queue is pushed onto the call stack. When this task finishes, the cycle continues and the next task on the event queue is pushed onto the call stack. Thus, if a certain task takes a long time to finish, the tasks behind it on the event queue will have to wait.

#### ****Types of HTML DOM Events****

Here are some examples of HTML DOM Events:

* Click Event - occurs when a user clicks a DOM element
* Mouseenter Event - occurs when a pointer is moved over an element
* Mouseleave Event - occurs when a pointer is moved out of an element
* Keypress Event - occurs when a key is pressed

#### ****Referencing DOM Elements****

DOM elements can be referenced using the **document.getElementById(id)**method call if the DOM element has an id attribute defined.

Notice how a HTML DOM element is referenced in JavaScript:

HTML:

<button id="myId">Button</button>

JavaScript:

var button = document.getElementById('myId');

#### ****addEventListener()****

The **addEventListener(eventType,eventHandler)** method call is used to add an event listener to a DOM object. The **eventType** argument is a string that represents the type of event that is being listened for. The **eventHandler** is a callback function that handles the event once it is detected.

Notice how the **document.getElementById()**and**addEventListener()**method calls are used to reference a DOM element and add an event listener to it.

HTML:

<button id="myId">Button</button>

JavaScript:

document.getElementById('myId').addEventListener('eventType', function(){

//handle event here

});

#### Event Attributes

DOM elements have event attributes that can be used to handle events.

Here are several of the attributes that can act as event attributes:

* onclick - handles click events
* onmouseover - handles mouseover events
* onmouseleave - handles mouseleave events
* onkeypress - handles keypress events

The general format for the name of the event attributes is: "on" + "eventType". Event handler functions can be assigned to the event attributes to handle events.

Notice how an event handler function is assigned to an event attribute:

HTML:

<button id="myId">Button</button>

JavaScript:

document.getElementById('myId').oneventname = function(){

//do something

}

The event attribute can also be assigned in the HTML code.

Notice how the event attribute can be assigned in HTML:

HTML:

<button id="myId" oneventname = "eventHandler()" >Button</button>

JavaScript:

function eventHandler(){

//do something

}

### ****Handling Click Events****

Notice how the **addEventListener()** function is used to add a click event handler to a button DOM element:

HTML:

<button id="mybutton">Click</button>

JavaScript:

var value = 0;

document.getElementById('myButton').addEventListener('click', function(){

value++;

document.getElementById('myButton').innerHTML = value;

//sets the HTML text inside the button to display the number of times it has been clicked

});

The above code adds a click event listener that increments the **value** variable every time the button is clicked. The value of the **value** variable is then displayed inside the button.

The click event can also be handled with an event attribute.

Notice how an anonymous event handler is assigned to the "onclick" event attribute:

var value = 0;

document.getElementById('myButton').onclick = function(){

value++;

document.getElementById('myButton').innerHTML = value;

//sets the HTML text inside the button to display the number of times it has been clicked

}

The click event can also be handled by defining the event attribute in HTML.

Notice how the the **handleClick()** event handler is assigned to the "onclick" event attribute in HTML:

HTML:

<button id="myButton" onclick = "handleClick()">click </button>

JavaScript:

var value = 0;

function handleClick(){

value++;

document.getElementById('myButton').innerHTML = value;

//sets the HTML text inside the button to display the number of times it has been clicked

}

### ****Handling Keypress Events****

Notice how the **addEventListener()** method can be used to handle keypress events:

HTML:

<p id="text">Key Pressed: <p>

JavaScript:

document.addEventListener('keypress',handleKeyPress);

function handleKeyPress(event){

var keyPressed= event.key; //event.key contains the key that was pressed

document.getElementById("text").innerHTML = "Key Pressed: " + keyPressed;

//sets the HTML text to display the key pressed

}

In the above code, an event listener is added directly to the document DOM element. The **document** object is the root node where all of the other HTML elements stem from. When a key is pressed, the key will be displayed in the paragraph element. The first argument of the event handler will contain the Event object being handled. The **key** attribute contains the value of the key that was last pressed.

The keypress event can also be handled using an event attribute.

Notice how the the **handleKeyPress()** event handler is assigned to the "onkeypress" event attribute:

document.onkeypress = handleKeyPress;

function handleKeyPress(event){

var keyPressed= event.key; //event.key contains the key that was pressed

document.getElementById("text").innerHTML = "Key Pressed: " + keyPressed;

//sets the HTML text to display the key pressed

}