Agenda

1. JavaScript Event Loop
2. SetTimeOut(), SetInterval()

JavaScript Event Loop

JavaScript single-threaded model

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

The JavaScript engine executes a script from the top of the file and works its way down. It creates the execution contexts, pushes, and pops functior onto and off the call stack in the execution phase.

If a function takes a long time to execute, you cannot interact with the web browser during the function's execution because the page hangs.

A function that takes a long time to complete is called a blocking function. Technically, a blocking function blocks all the interactions on the webpag such as mouse click.

An example of a blocking function is a function that calls an API from a remote server.

The following example uses a big loop to simulate a blocking function:

```javascript
function task(message) {
// emulate time consuming tasklet n = 10000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Download a file.');
console.log('Done!');
Code language: JavaScript (javascript)
```

In this example, we have a big `while` loop inside the `task()` function that emulates a time-consuming task. The `task()` function is a blockir function.

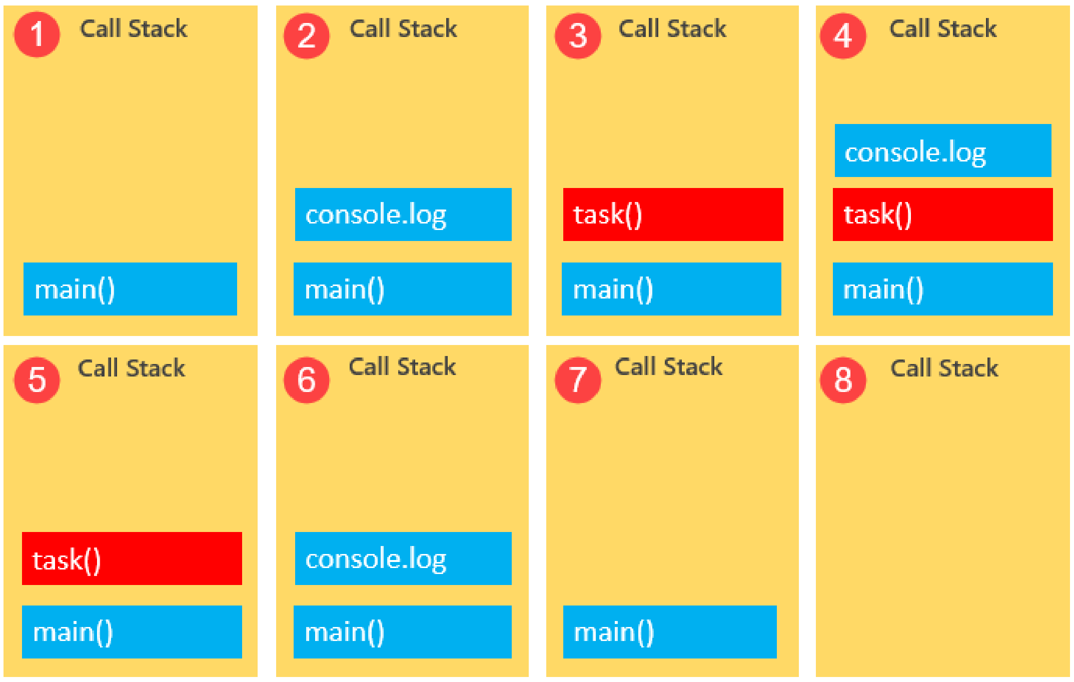
The script hangs for a few seconds (depending on how fast the computer is) and issues the following output:

```
Start script...
Download a file.
Done!
```

To execute the script, the JavaScript engine places the first call `console.log()` on top of the call stack and executes it. Then, it place the `task()` function on top of the call stack and executes the function.

However, it'll take a while to complete the `task()` function. Therefore, you'll see the message `'Download a file.'` a little time later. Aft the `task()` function completes, the JavaScript engine pops it off the call stack.

Finally, the JavaScript engine places the last call to the `console.log('Done!')` function and executes it, which will be very fast.



Callbacks to the rescue

To prevent a blocking function from blocking other activities, you typically put it in a callback function for execution later. For example:

```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
}, 1000);

console.log('Done!');
Code language: JavaScript (javascript)
```

In this example, you'll see the message `'Start script...'` and `'Done!'` immediately. And after that, you'll see the message `'Download file'` .

Here's the output:

```
Start script...
Done!
Download a file.
```

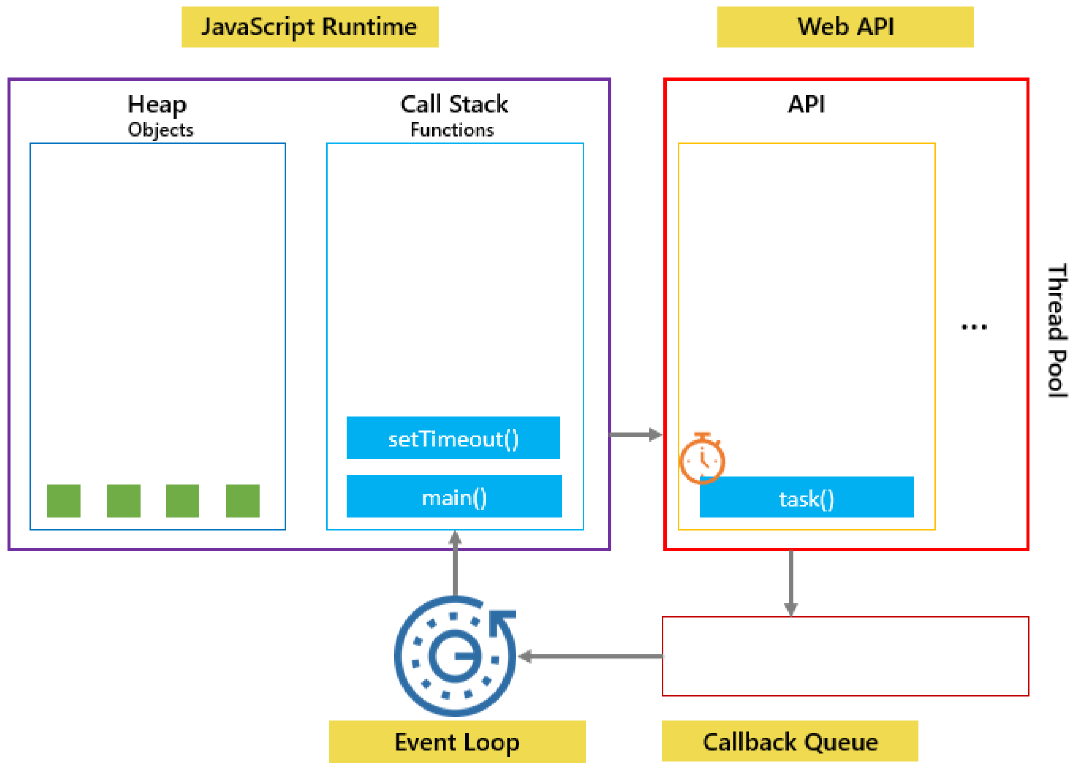
As mentioned earlier, the JavaScript engine can do only one thing at a time. However, it's more precise to say that the JavaScript runtime can do or thing at a time.

The web browser also has other components, not just the JavaScript engine.

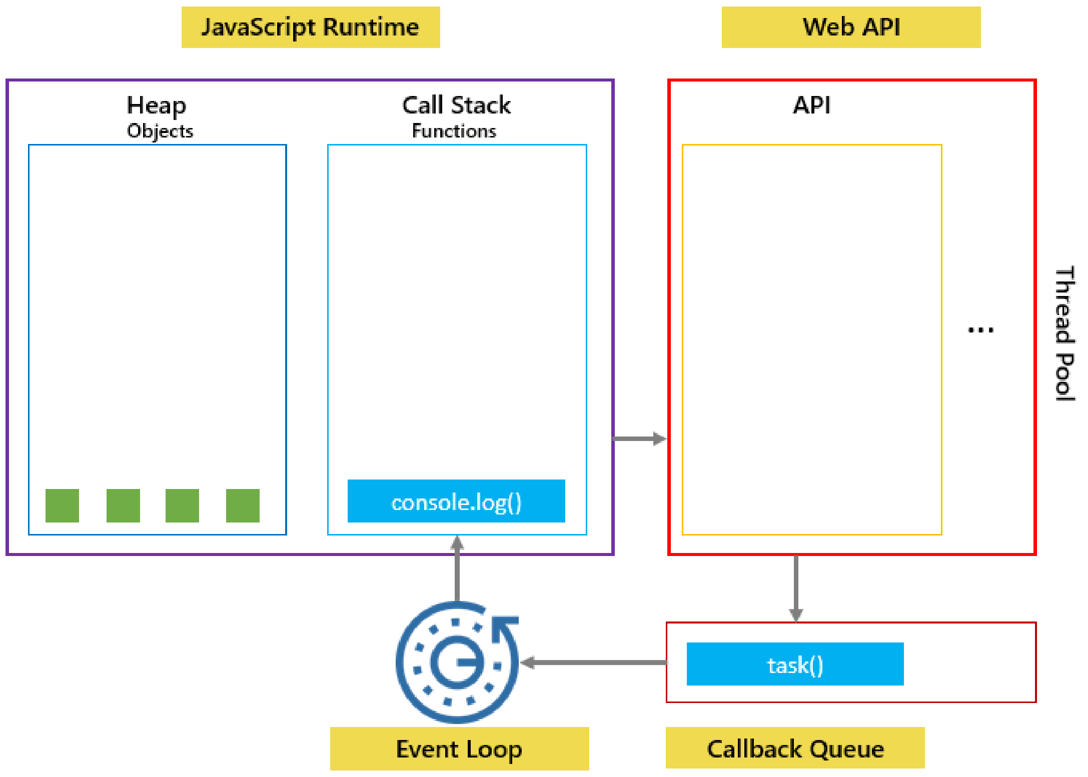
When you call the `setTimeout()` function, make a fetch request, or click a button, the web browser can do these activities concurrently ar asynchronously.

The `setTimeout()` , fetch requests, and DOM events are parts of the Web APIs of the web browser.

In our example, when calling the `setTimeout()` function, the JavaScript engine places it on the call stack, and the Web API creates a timer th expires in 1 second.

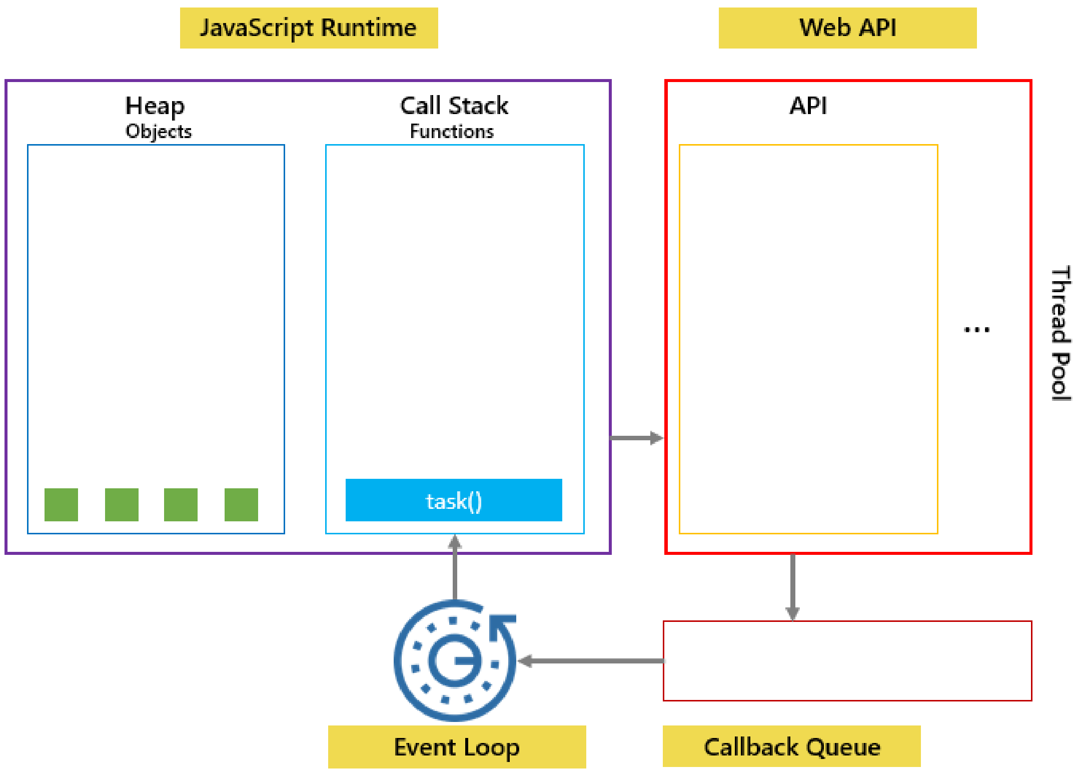


Then JavaScript engine place the task() function is into a queue called a callback queue or a task queue:



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callba
queue is empty, nothing will happen:



See another example:

```
console.log('Hi!');

setTimeout(() => {
    console.log('Execute immediately.');
}, 0);

console.log('Bye!');

Code language: JavaScript (javascript)
```

In this example, the timeout is 0 second, so the message `'Execute immediately.'` should appear before the message `'Bye!'`. However, doesn't work like that.

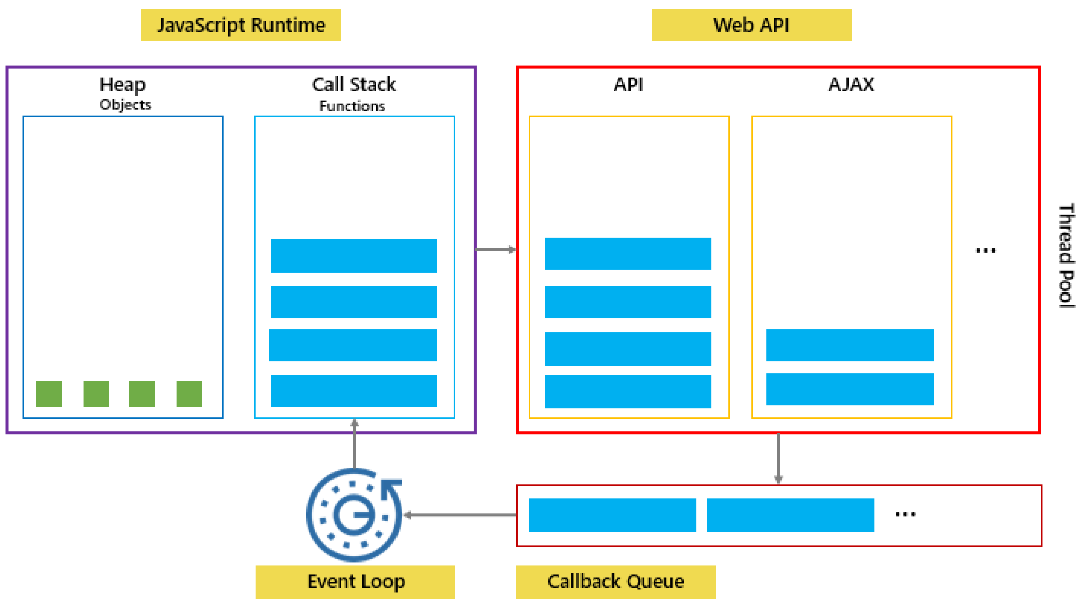
The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, tl JavaScript engine executes it after the `console.log('Bye!')`.

```
console.log('Execute immediately.');
Code language: JavaScript (javascript)
```

Here's the output:

```
Hi!
Bye!
Execute immediately.
```

The following picture illustrates JavaScript runtime, Web API, Call stack, and Event loop:



Note: We have learned about the JavaScript event loop, a constantly running process that coordinates the tasks between the call stack and callba
queue to achieve concurrency.

setTimeout()

The `setTimeout()` method executes a block of code after the specified time. The method executes the code only once.

The commonly used syntax of JavaScript setTimeout is:

```
setTimeout(function, milliseconds);
```

Its parameters are:

- **function** - a function containing a block of code
- **milliseconds** - the time after which the function is executed

The `setTimeout()` method returns an **intervalID**, which is a positive integer.

Example 1: Display a Text Once After 3 Second

```
// program to display a text using setTimeout method
function greet() {
    console.log('Hello world');
}

setTimeout(greet, 3000);
console.log('This message is shown first');
```

Output

```
This message is shown firstHello world
```

In the above program, the `setTimeout()` method calls the `greet()` function after **3000** milliseconds (**3** second).

Hence, the program displays the text Hello world only once after **3** seconds.

Note: The `setTimeout()` method is useful when you want to execute a block of once after some time. For example, showing a message to a us
after the specified time.

The `setTimeout()` method returns the interval id. For example,

```
// program to display a text using setTimeout method
function greet() {
    console.log('Hello world');
}
```

```
let intervalId = setTimeout(greet, 3000);
console.log('Id: ' + intervalId);
```

Output

```
Id: 3
Hello world
```

Example 2: Display Time Every 3 Second

```
// program to display time every 3 seconds
function showTime() {

    // return new date and time
    let dateTime= new Date();

    // returns the current local time
    let time = dateTime.toLocaleTimeString();

    console.log(time)

    // display the time after 3 seconds
     setTimeout(showTime, 3000);
}

// calling the function
showTime();
```

Output

```
5:45:39 PM
5:45:43 PM
5:45:47 PM
5:45:50 PM
..................
```

The above program displays the time every **3** seconds.

The `setTimeout()` method calls the function only once after the time interval (here **3** seconds).

However, in the above program, since the function is calling itself, the program displays the time every **3** seconds.

This program runs indefinitely (until the memory runs out).

Note: If you need to execute a function multiple times, it's better to use the `setInterval()` method. It will be covered later in this lecture.

JavaScript clearTimeout()

As you have seen in the above example, the program executes a block of code after the specified time interval. If you want to stop this function call, yo can use the `clearTimeout()` method.

The syntax of `clearTimeout()` method is:

```
clearTimeout(intervalID);
```

Here, the `intervalID` is the return value of the `setTimeout()` method.

Example 3: Use clearTimeout() Method

```
// program to stop the setTimeout() method

let count = 0;

// function creation
function increaseCount(){

    // increasing the count by 1
    count += 1;
    console.log(count)
}

let id = setTimeout(increaseCount, 3000);

// clearTimeout
clearTimeout(id);
console.log('setTimeout is stopped.');
```

Output

```
setTimeout is stopped.
```

In the above program, the `setTimeout()` method is used to increase the value of count after 3 seconds. However, the `clearTimeout()` methc stops the function call of the `setTimeout()` method. Hence, the count value is not increased.

Note: You generally use the `clearTimeout()` method when you need to cancel the `setTimeout()` method call before it happens.

You can also pass additional arguments to the `setTimeout()` method. The syntax is:

```
setTimeout(function, milliseconds, parameter1, ....paramenterN);
```

When you pass additional parameters to the `setTimeout()` method, these parameters (`parameter1` , `parameter2` , etc.) will be passed to tl specified **function**.

For example,

```
// program to display a name
function greet(name, lastName) {
    console.log('Hello' + ' ' + name + ' ' + lastName);
}

// passing argument to setTimeout
setTimeout(greet, 1000, 'John', 'Doe');
```

Output

```
Hello John Doe
```

In the above program, two parameters `John` and `Doe` are passed to the `setTimeout()` method. These two parameters are the arguments that w be passed to the function (here, `greet()` function) that is defined inside the `setTimeout()` method.

JavaScript setInterval()

The `setInterval()` method repeats a block of code at every given timing event.

The commonly used syntax of JavaScript setInterval is:

```
setInterval(function, milliseconds);
```

Its parameters are:

- **function** - a function containing a block of code
- **milliseconds** - the time interval between the execution of the function

The `setInterval()` method returns an **intervalID** which is a positive integer.

Example 1: Display a Text Once Every 1 Second

```
// program to display a text using setInterval method
function greet() {
    console.log('Hello world');
}
```

```
setInterval(greet, 1000);
```

Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
....
```

In the above program, the `setInterval()` method calls the `greet()` function every **1000** milliseconds(**1** second).

Hence the program displays the text Hello world once every **1** second.

Note: The `setInterval()` method is useful when you want to repeat a block of code multiple times. For example, showing a message at a fixe interval.

Example 2: Display Time Every 5 Seconds

```
// program to display time every 5 seconds
function showTime() {

    // return new date and time
    let dateTime= new Date();

    // return the time
    let time = dateTime.toLocaleTimeString();

    console.log(time)
}

let display = setInterval(showTime, 5000);
```

Output

```
"5:15:28 PM"
"5:15:33 PM"
"5:15:38 PM"
....
```

The above program displays the current time once every **5** seconds.

`new Date()` gives the current date and time. And `toLocaleTimeString()` returns the current time. It will be covered in later topics.

JavaScript clearInterval()

As you have seen in the above example, the program executes a block of code at every specified time interval. If you want to stop this function call, the you can use the `clearInterval()` method.

The syntax of `clearInterval()` method is:

```
clearInterval(intervalID);
```

Here, the `intervalID` is the return value of the `setInterval()` method.

Example 3: Use clearInterval() Method

```
// program to stop the setInterval() method after five times

let count = 0;

// function creation
let interval = setInterval(function(){

    // increasing the count by 1
```

```
        count += 1;

        // when count equals to 5, stop the function
        if(count === 5){
            clearInterval(interval);
        }

        // display the current time
        let dateTime= new Date();
        let time = dateTime.toLocaleTimeString();
        console.log(time);

    }, 2000);
```

Output

```
4:47:41 PM
4:47:43 PM
4:47:45 PM
4:47:47 PM
4:47:49 PM
```

In the above program, the `setInterval()` method is used to display the current time every **2** seconds. The `clearInterval()` method stops th function call after **5** times.

You can also pass additional arguments to the `setInterval()` method. The syntax is:

```
setInterval(function, milliseconds, parameter1, ....paramenterN);
```

When you pass additional parameters to the `setInterval()` method, these parameters (`parameter1` , `parameter2` , etc.) will be passed to th specified **function**.

For example,

```
// program to display a name
function greet(name, lastName) {
    console.log('Hello' + ' ' + name + ' ' + lastName);
}

// passing argument to setInterval
setInterval(greet, 1000, 'John', 'Doe');
```

Output

```
Hello John Doe
Hello John Doe
Hello John Doe
....
```

In the above program, two parameters `John` and `Doe` are passed to the `setInterval()` method. These two parameters are the arguments th will be passed to the function (here, `greet()` function) that is defined inside the `setInterval()` method.

Note: If you only need to execute a function one time, it's better to use the setTimeout() method.

Concurrency in JavaScript

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it not: Multiple computers in a network. Multiple applications running on one computer. Multiple processors in a computer (today, often multiple process cores on a single chip).

> Note : This topic will be covered in detail in Frontend module.

Interview Questions

> What is your understanding of the Event Loop concept in JavaScript?

The Event Loop is a mechanism used by JavaScript to handle asynchronous events. It is a continuous loop that checks for events and then processe them accordingly. This allows JavaScript to handle multiple events at the same time and makes it possible for things like animations and user input to b processed without blocking the main thread of execution.

> What will the following code output?

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() { alert(i); }, 1000 + i);
}
```

Answer

The goal of the code above is to alert the numbers 0, 1, and 2 each after 1, 1.1, and 1.2 seconds, respectively. The problem though, is that if you run th above code in your console, you actually get the number **3** alerted 3 times after 1, 1.1, and 1.2 seconds. This is because of an issue with JavaScri closures

A JavaScript **closure** is when an inner function has access to its outer enclosing function's variables and properties. In the code above, the following lir of code:

```
setTimeout(function() { alert(i); }, 1000 + i);
```

uses a variable i which is declared outside of itself. The variable i is actually declared within the for loop and the inner function accesses it. So when th for loop is done running, each of the inner functions refers to the same variable i, which at the end of the loop is equal to 3. Our goal is for each inn function to maintain its reference to the variable i without the value of it being altered. We'll solve this using an IIFE , or an immediately-invoked functic expression.

```
for (var i = 0; i < 3; i++) {
  setTimeout(function(i_local) {
    return function() { alert(i_local); }
  }(i), 1000 + i);
}
```

We pass the variable i into the outer function as a local variable named i_local, where we then return a function that will alert the i_local for us. Th should now correctly alert the numbers 0, 1, and 2 in the correct order.