

CSC-336

Web Technologies

Lecture 11

Topics:

- Event Listener, Anonymous Functions, Higher Order Functions
- JS Object, Switch, Methods , this operator
- Constructor functions, Call Back Functions

Muhammad Naveed Shaikh

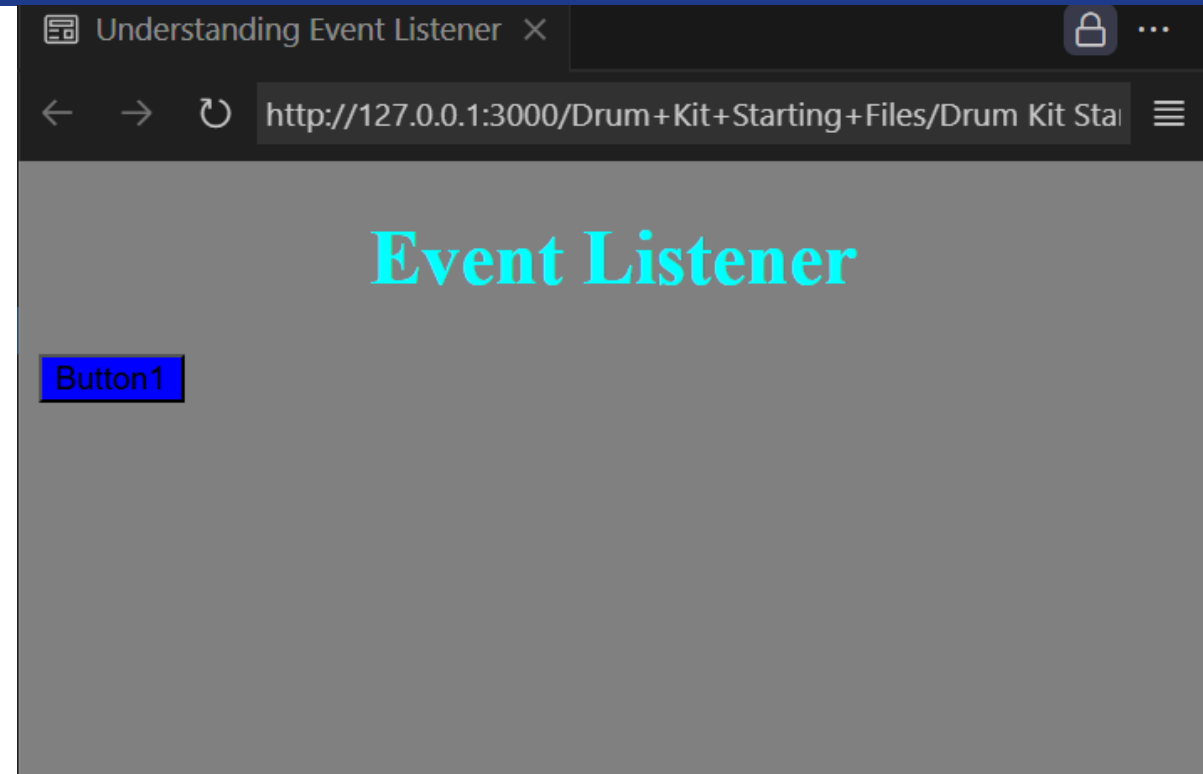
Department of Computer Engineering
naveedshaikh@cuiatd.edu.pk



Understanding Event Listener

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <BUtton>Button1</BUtton>
</body>
</html>
```

- UI is ready.
- But interaction is missing!
- We need to add an event listener for click actions.”



Event Listeners

- The `addEventListener()` method is used to attach an event handler to a specified element (or event target). This allows you to execute a function whenever a certain type of event occurs on that element.

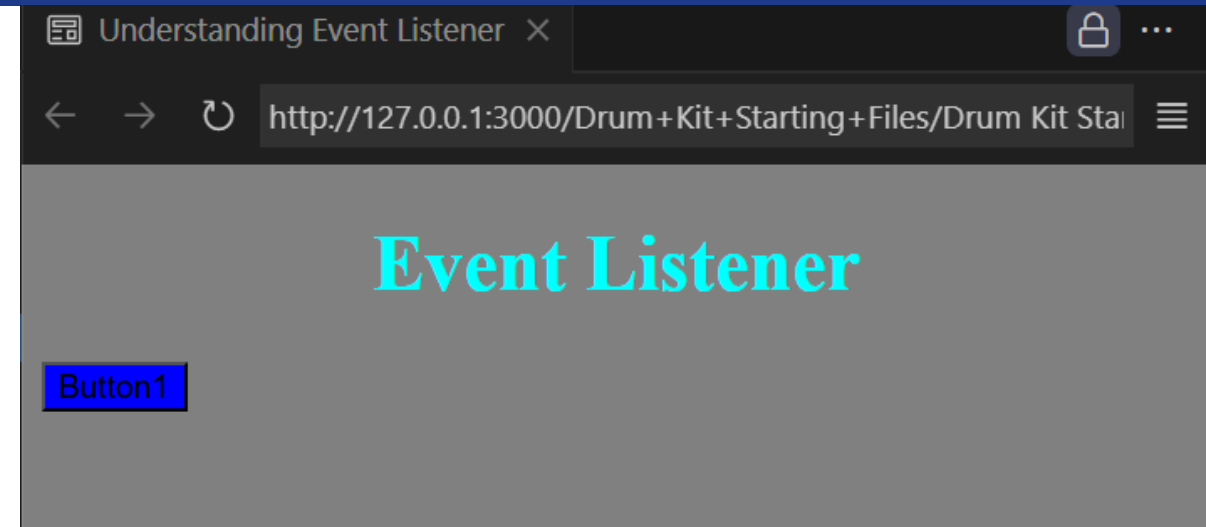
`target.addEventListener(type, listener);`

- Type: A **case-sensitive string** representing the event type to listen for (e.g., "click", "mouseover", "keydown").
- Listener: The function or object that will be called when the event occurs.
 - This can be:
 - A function: `function(event) { /* code */ }`
 - An object with a `handleEvent()` method
 - null, which effectively does nothing.

```
var obj = {  
  handleEvent: function(event) {  
    console.log("Event handled");  
  }  
};  
element.addEventListener("click", obj);
```

Understanding Event Listener

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <BUtton>Button1</BUtton>
<script src="index.js"></script>
</body>
</html>
```

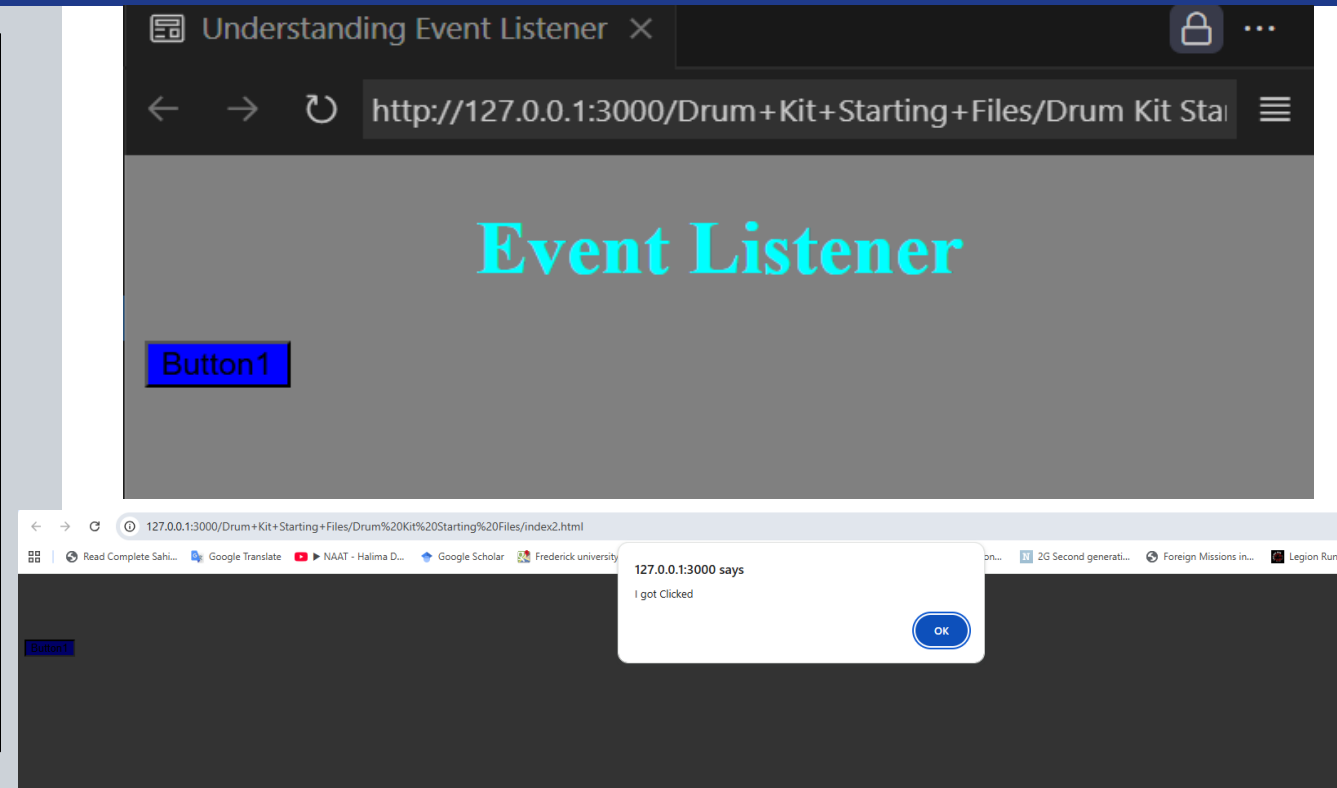


```
document.querySelector("button").addEventListener("click",handleClick)
function handleClick() {
  alert("I got Clicked");
}
```

Understanding Event Listener

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <BUtton>Button1</BUtton>
  <script src="index.js"></script>
</body>
</html>
```

```
document.querySelector("button").addEventListener("click",handleClick)
function handleClick() {
  alert("I got Clicked");
}
```

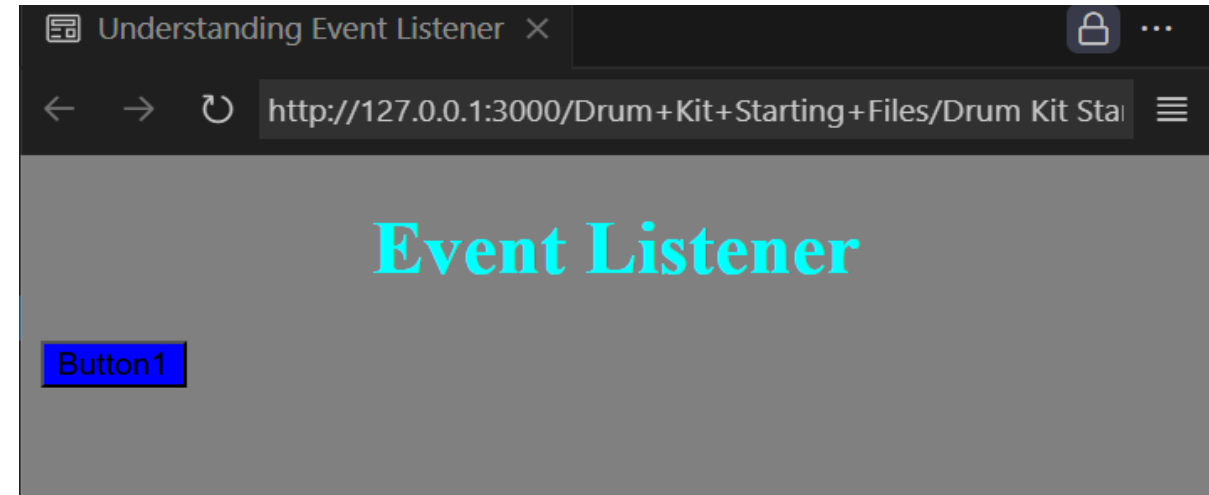


Anonymous Functions

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <BUtton>Button1</BUtton>
<script src="index.js"></script>
</body>
</html>
```

```
document.querySelector("button").addEventListener("click", function () {
  alert("I got Clicked");
})
```

```
document.querySelector("button").addEventListener("click", handleClick)
function handleClick() {
  alert("I got Clicked");
}
```



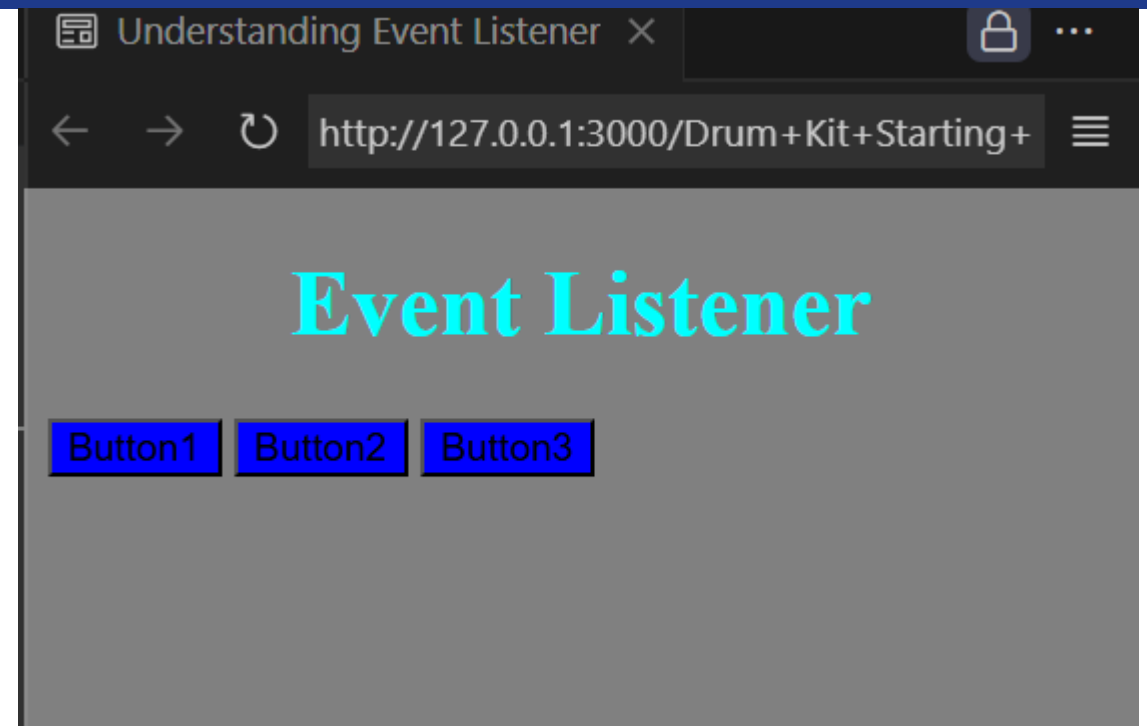
Dealing with multiple Event Listeners

```
<body>

  <h1>Event Listener</h1>
  <button>Button1</button>
  <button>Button2</button>
  <button>Button3</button>
  <script src="index2.js" charset="utf-8">

    </script>
</body>
```

```
document.querySelector("button").addEventListener("click", function () {
  alert("I got Clicked");
})
```



Can you modify this JS code to apply it to multiple buttons

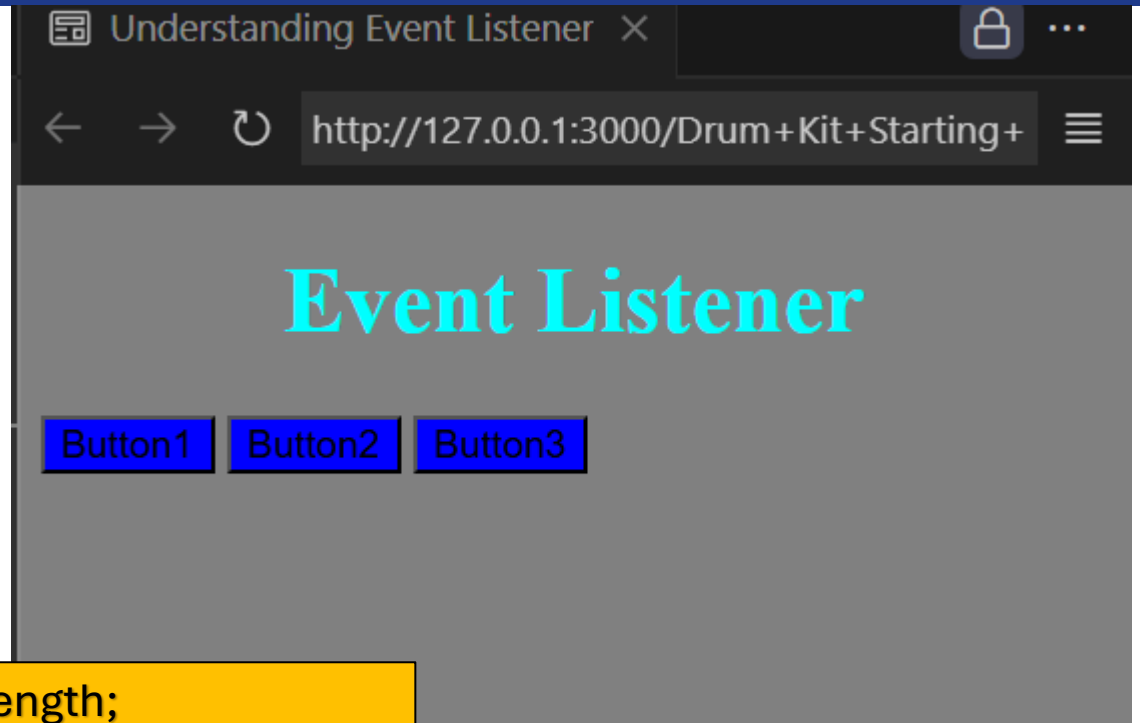
Dealing with multiple Event Listeners

```
<body>

  <h1>Event Listener</h1>
  <button>Button1</button>
  <button>Button2</button>
  <button>Button3</button>
  <script src="index2.js" charset="utf-8">

    </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
  document.querySelectorAll("button")[i].addEventListener("click", function (){
    alert("I got Clicked");
  })
}
```



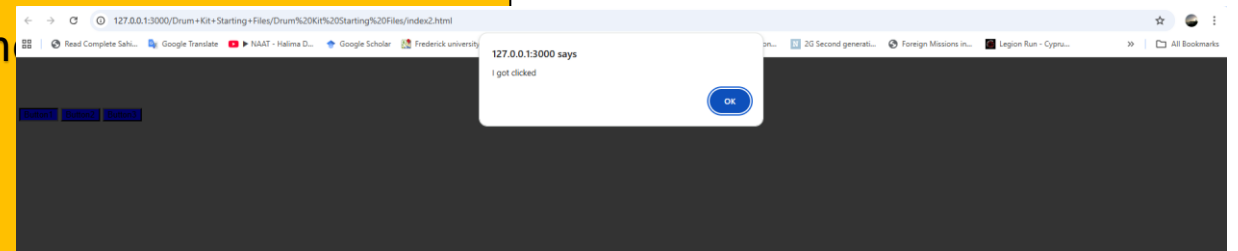
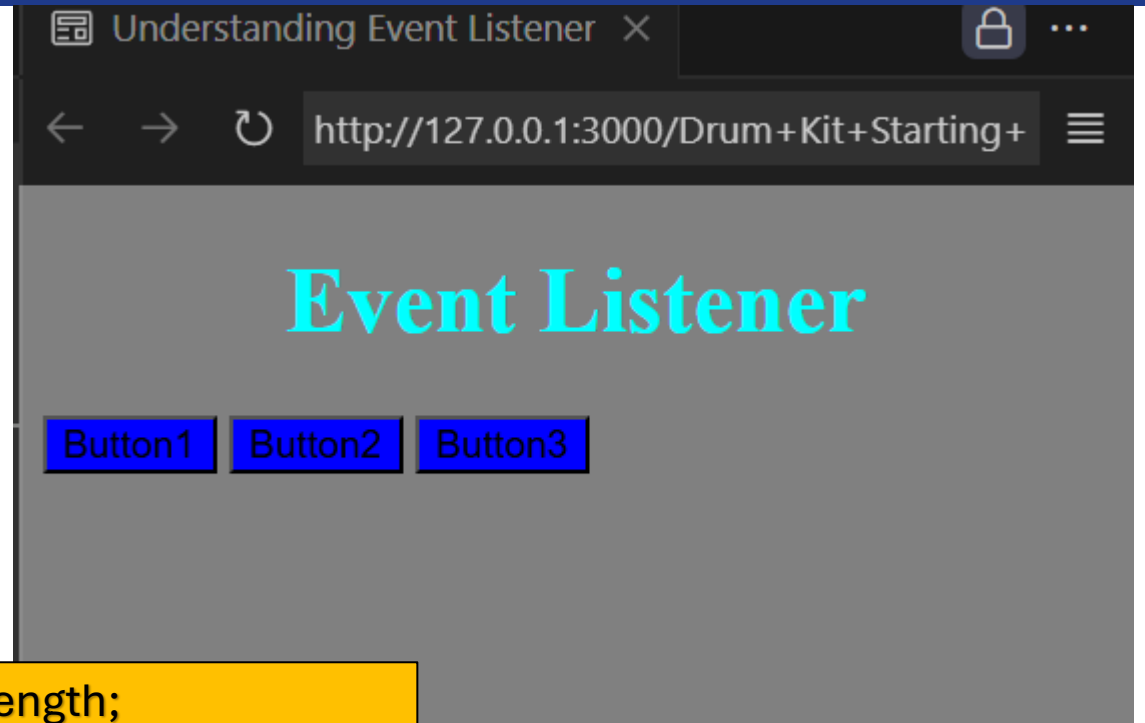
Dealing with multiple Event Listeners

```
<body>

  <h1>Event Listener</h1>
  <button>Button1</button>
  <button>Button2</button>
  <button>Button3</button>
  <script src="index2.js" charset="utf-8">

    </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
  document.querySelectorAll("button")[i].addEventListener(
    alert("I got Clicked");
  })
}
```



Two Inputs of addEventListener()

- Event Type → What to listen for (e.g., "click")
- Event Handler → Function to run (e.g., handleClick)
- `element.addEventListener("click", handleClick);`
- Passing function as input to another function
 - Higher Order Functions

Why We Pass Functions into Event Listeners?

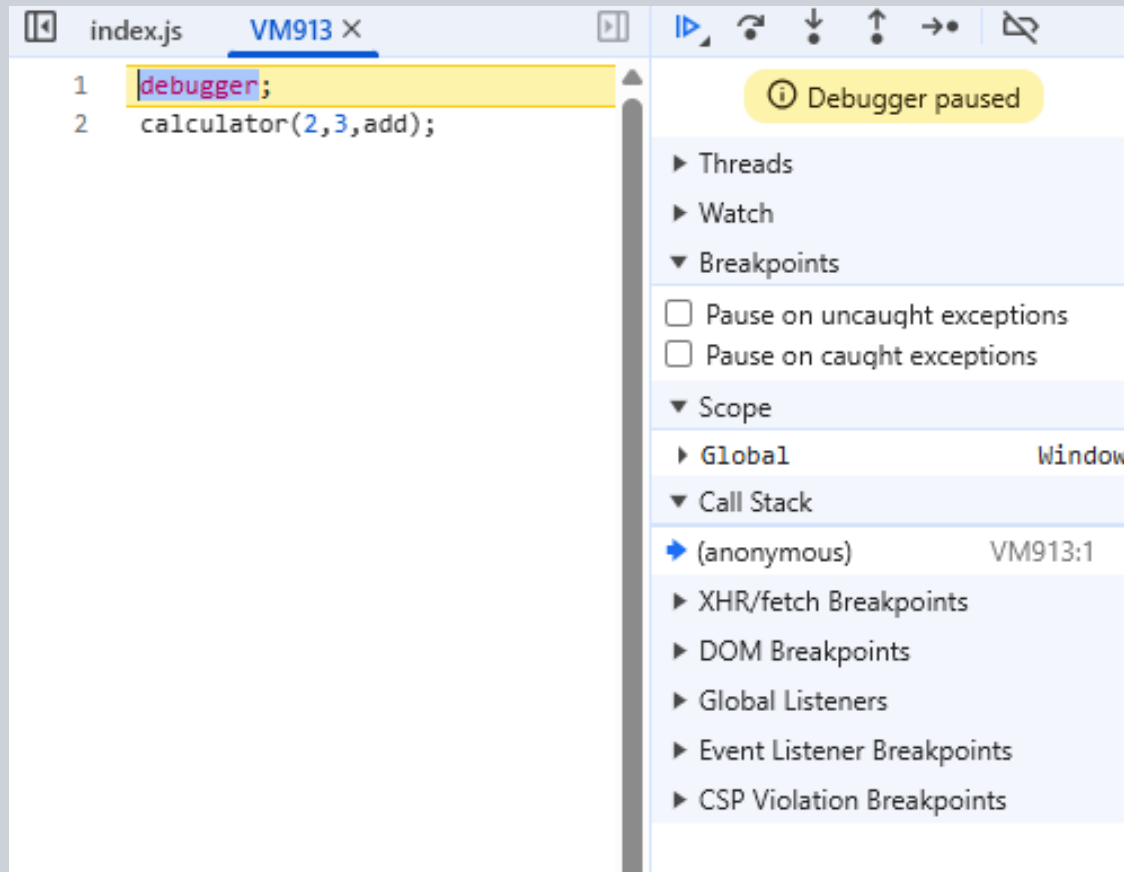
- Functions can be passed as inputs to `.addEventListener()`
- The function becomes an event handler
- It runs only when the event actually happens (e.g., user click)
- Helps reuse code for different elements and events
- Makes websites interactive and efficient

Example of Higher Order Functions

- `calculator(2, 3, add);` `// returns 5`
- `calculator(3, 4, multiply);` `// returns 12`
- ```
function add(a,b) {
 return a+b; }
```
- ```
function multiply (a,b) {  
    return a*b; }
```
- ```
function calculator (a,b, operator) {
 return operator (a,b);}
```

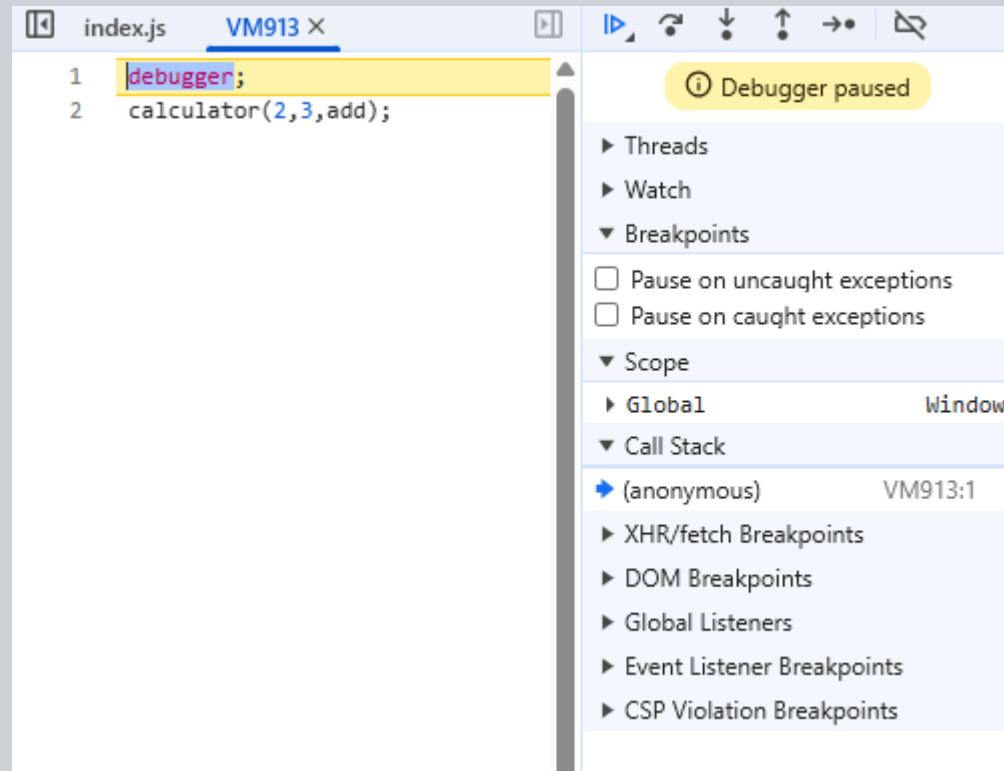
# Debugging Step-by-Step

- Type in Debugger in console
  - And, call the function that you want to debug:



# Debugging Step-by-Step

- Type in Debugger in console
  - And, call the function that you want to debug:



# Debugging Step-by-Step

- Type in Debugger in console
  - And, call the function that you want to debug:

The screenshot displays a web browser's developer console with a JavaScript debugger. The left pane shows the source code of `index.js` with the following content:

```
1 function add(num1,num2){
2 return (num1+num2)
3 };
4 function subtract(num1,num2) {
5 return (num1-num2);
6 };
7 function calculator(num1,num2,operator) { num1 = 2, num2 = 3, operator = f add(num1,num2)
8 return operator(num1,num2);
9 }
```

The debugger is paused at line 8. The right sidebar shows the 'Debugger paused' state with the following sections:

- Threads**
- Watch**
- Breakpoints**
- ☐ Pause on uncaught exceptions
- ☐ Pause on caught exceptions
- Scope**
  - Local**
    - `this`: Window
    - `num1`: 2
    - `num2`: 3
    - `operator`: `f add(num1,num2)`
  - Global**: Window
- Call Stack**
  - `calculator` at `index.js:8`
  - `(anonymous)` at `VM928:2`
- XHR/fetch Breakpoints**
- DOM Breakpoints**
- Global Listeners**
- Event Listener Breakpoints**
- CSP Violation Breakpoints**

# Debugging Step-by-Step

- Type in Debugger in console

- And,

The screenshot displays a web browser's developer console with a JavaScript debugger. The main pane shows the source code of `index.js` with the following content:

```
1 function add(num1,num2){ num1 = 2, num2 = 3
2 return (num1+num2)
3 };
4 function subtract(num1,num2) {
5 return (num1-num2);
6 };
7 function calculator(num1,num2,operator) {
8 return operator(num1,num2);
9 }
```

The debugger is paused at line 2, where the `return` statement is highlighted. The right sidebar provides additional context:

- Debugger paused**: A yellow banner at the top of the sidebar.
- Threads**, **Watch**, and **Breakpoints**: Collapsible sections for further debugging options.
- Scope**: A section showing the current execution context.
  - Local**: Contains the variables `num1: 2` and `num2: 3`.
  - Global**: Points to the `Window` object.
- Call Stack**: A list of function calls that led to the current state.
  - add**: The current function, located at `index.js:2`.
  - calculator**: The function that called `add`, located at `index.js:8`.
  - (anonymous)**: The initial call, located at `VM928:2`.
- XHR/fetch Breakpoints**, **DOM Breakpoints**, **Global Listeners**, **Event Listener Breakpoints**, and **CSP Violation Breakpoints**: Additional debugging categories.



# Debugging Step-by-Step

- Type in
- And, c

The screenshot shows a web browser's developer console with a JavaScript debugger. The main pane displays the source code of `index.js` with the following content:

```
1 function add(num1,num2){ num1 = 2, num2 = 3
2 return (num1+num2)
3 };
4 function subtract(num1,num2) {
5 return (num1-num2);
6 };
7 function calculator(num1,num2,operator) {
8 return operator(num1,num2);
9 }
```

The debugger is paused at line 2, `return (num1+num2)`. The right sidebar provides details about the current execution state:

- Debugger paused**: A yellow banner at the top of the sidebar.
- Threads**: A section with a right-pointing arrow.
- Watch**: A section with a right-pointing arrow.
- Breakpoints**: A section with a downward-pointing arrow.
- Pause on uncaught exceptions**: An unchecked checkbox.
- Pause on caught exceptions**: An unchecked checkbox.
- Scope**: A section with a downward-pointing arrow.
- Local**: A section with a downward-pointing arrow showing the return value and variables:
  - Return value: 5
  - `this`: Window
  - `num1`: 2
  - `num2`: 3
- Global**: A section with a right-pointing arrow, showing the `Window` object.
- Call Stack**: A section with a downward-pointing arrow showing the sequence of function calls:
  - `add` at `index.js:2` (highlighted with a blue arrow)
  - `calculator` at `index.js:8`
  - `(anonymous)` at `VM928:2`
- XHR/fetch Breakpoints**: A section with a right-pointing arrow.
- DOM Breakpoints**: A section with a right-pointing arrow.
- Global Listeners**: A section with a right-pointing arrow.
- Event Listener Breakpoints**: A section with a right-pointing arrow.
- CSP Violation Breakpoints**: A section with a right-pointing arrow.

# Debugging Step-by-Step

- Type in Debugger in console
- And, c

The screenshot shows a web browser's developer console with a JavaScript debugger. The code in the console is as follows:

```
1 function add(num1,num2){
2 return (num1+num2)
3 };
4 function subtract(num1,num2) {
5 return (num1-num2);
6 };
7 function calculator(num1,num2,operator) { num1 = 2, num2 = 3, operator = f add(num1,num2)
8 return operator(num1,num2);
9 }
```

The debugger is paused at line 8. The right sidebar shows the 'Debugger paused' state and the 'Scope' panel with the following variables:

- Local
  - Return value: 5
  - this: Window
  - num1: 2
  - num2: 3
  - operator: f add(num1,num2)
- Global: Window

The 'Call Stack' panel shows the following calls:

- calculator index.js:8
- (anonymous) VM928:2

The 'XHR/fetch Breakpoints', 'DOM Breakpoints', 'Global Listeners', 'Event Listener Breakpoints', and 'CSP Violation Breakpoints' panels are also visible.

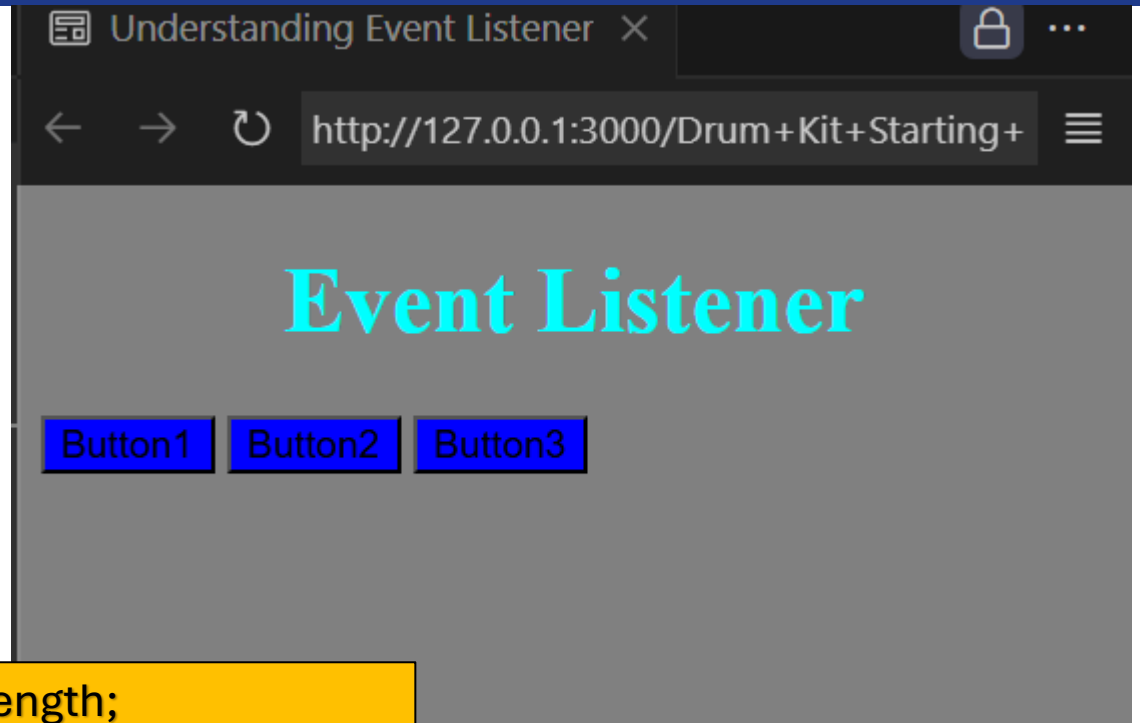
# JavaScript this Keyword

```
<body>

 <h1>Event Listener</h1>
 <button>Button1</button>
 <button>Button2</button>
 <button>Button3</button>
 <script src="index2.js" charset="utf-8">

 </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
 document.querySelectorAll("button")[i].addEventListener("click", function (){
 console.log(this);
 })
}
```



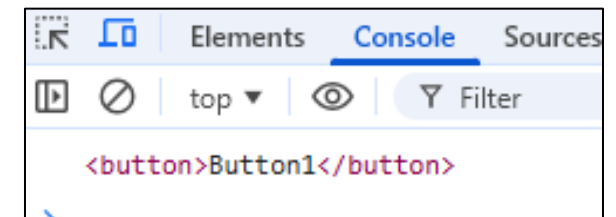
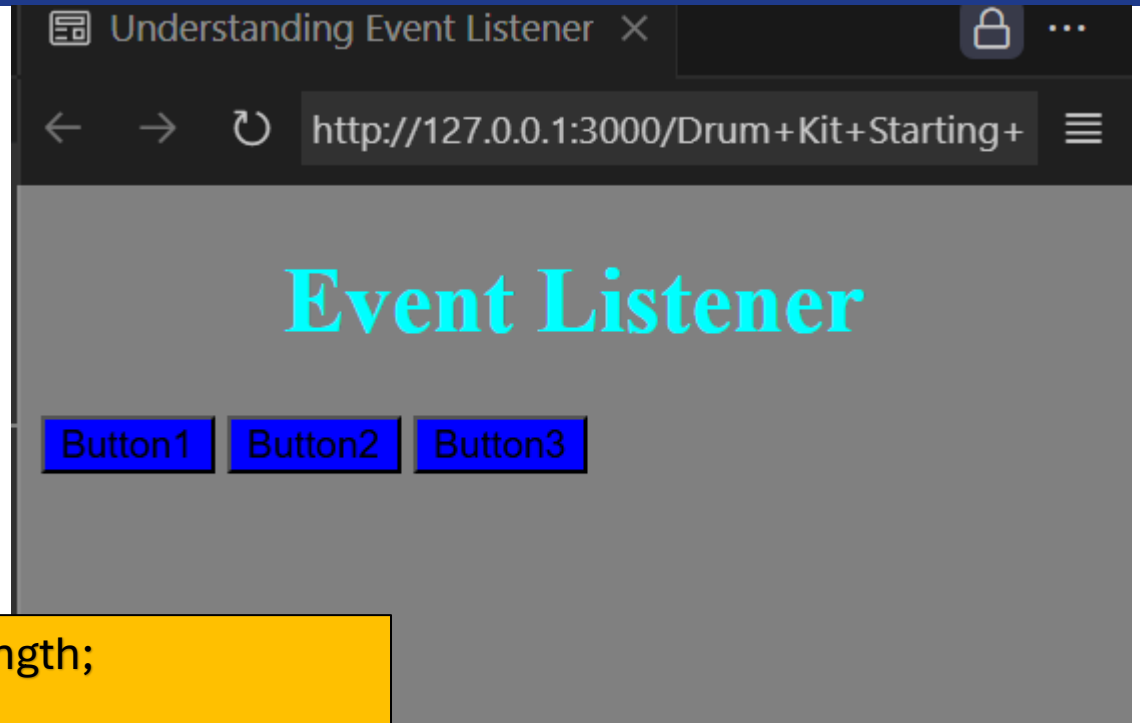
# JavaScript this Keyword

```
<body>

 <h1>Event Listener</h1>
 <button>Button1</button>
 <button>Button2</button>
 <button>Button3</button>
 <script src="index2.js" charset="utf-8">

 </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
 document.querySelectorAll("button")[i].addEventListener("click", function (){
 console.log(this);
 })
}
```



This is the output in console if Button1 is pressed

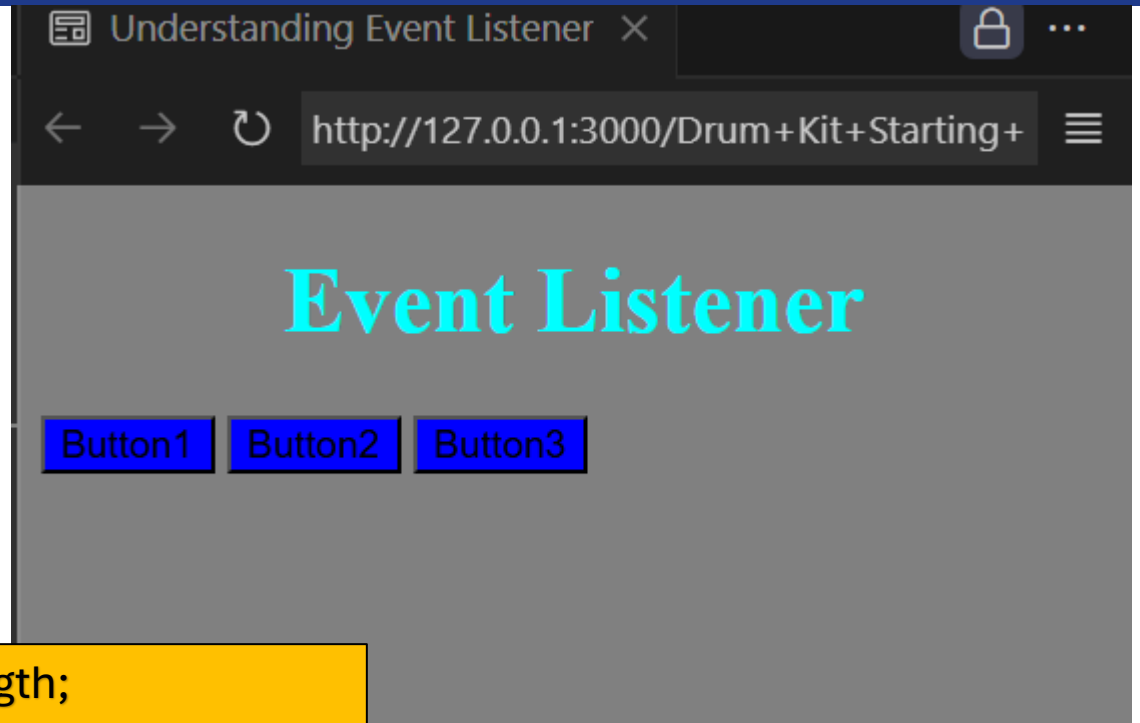
# JavaScript this Keyword

```
<body>

 <h1>Event Listener</h1>
 <button>Button1</button>
 <button>Button2</button>
 <button>Button3</button>
 <script src="index2.js" charset="utf-8">

 </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
 document.querySelectorAll("button")[i].addEventListener("click", function (){
 console.log(this.innerHTML);})
}
```



# JavaScript this Keyword

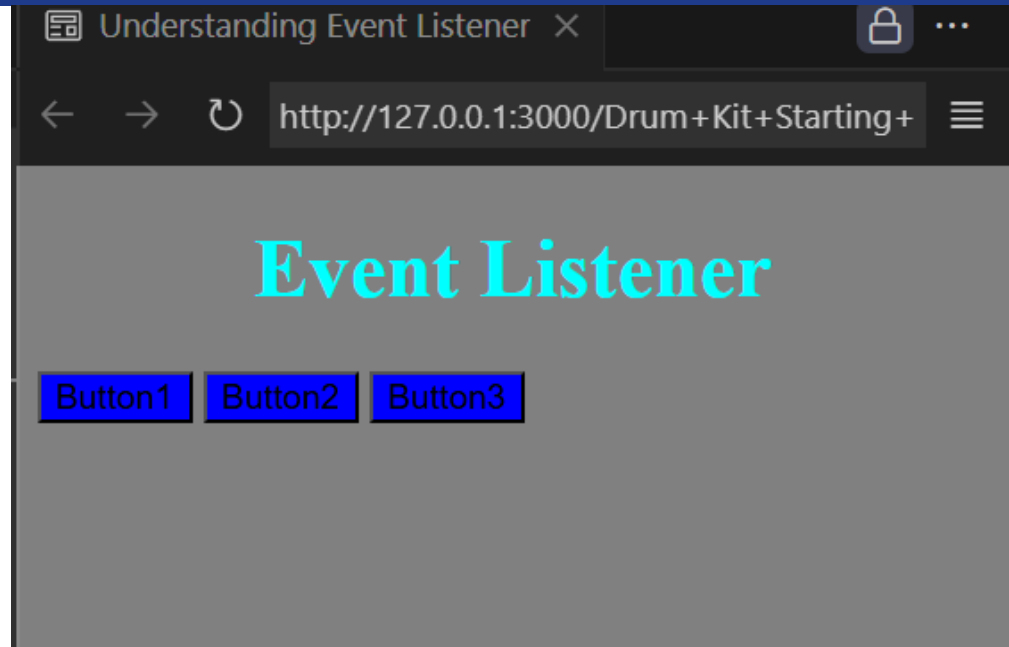
```
<body>

 <h1>Event Listener</h1>
 <button>Button1</button>
 <button>Button2</button>
 <button>Button3</button>
 <script src="index2.js" charset="utf-8">

 </script>
</body>
```

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
 document.querySelectorAll("button")[i].addEventListener("click", function (){
 console.log(this.innerHTML);})
}
```

This is the output in console



Button2

Button3

Button2

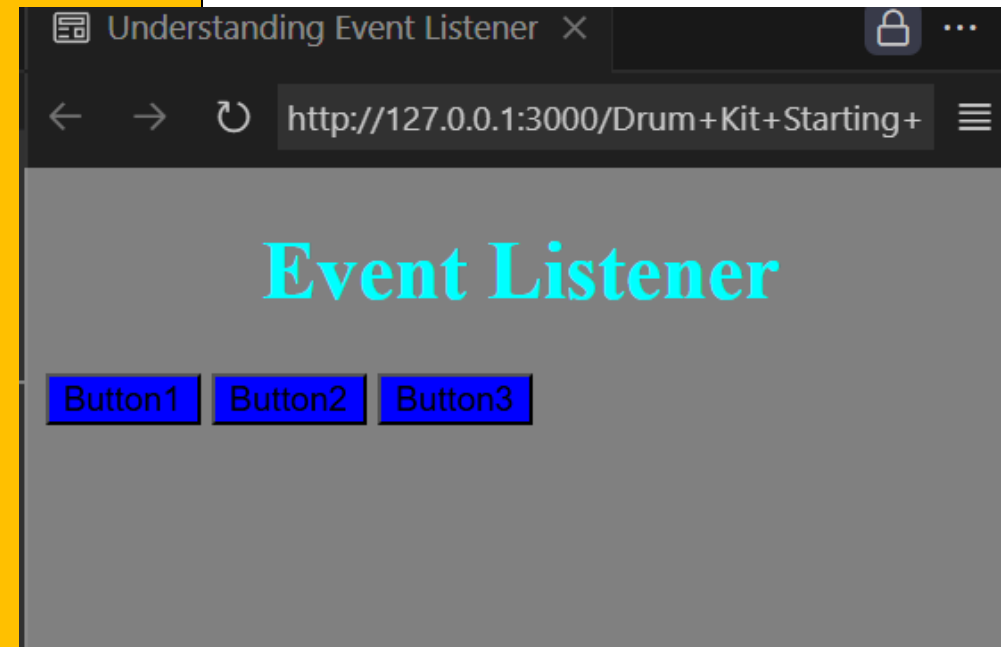
Button1

# Switch statement

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
document.querySelectorAll("button")[i].addEventListener("click", function (){
var selectedButton = this.innerHTML;
switch (selectedButton) {
case "Button1":
 alert("You have pressed "+selectedButton);
 break;
case "Button2":
 alert("You have pressed "+selectedButton);
 break;

default:

 break;
}
})
}
```

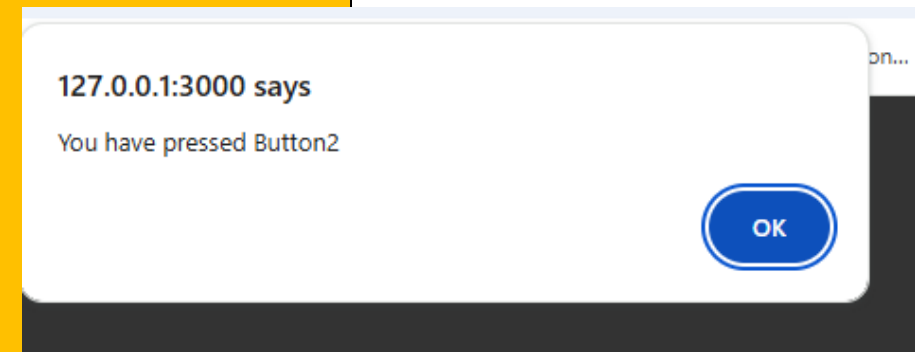
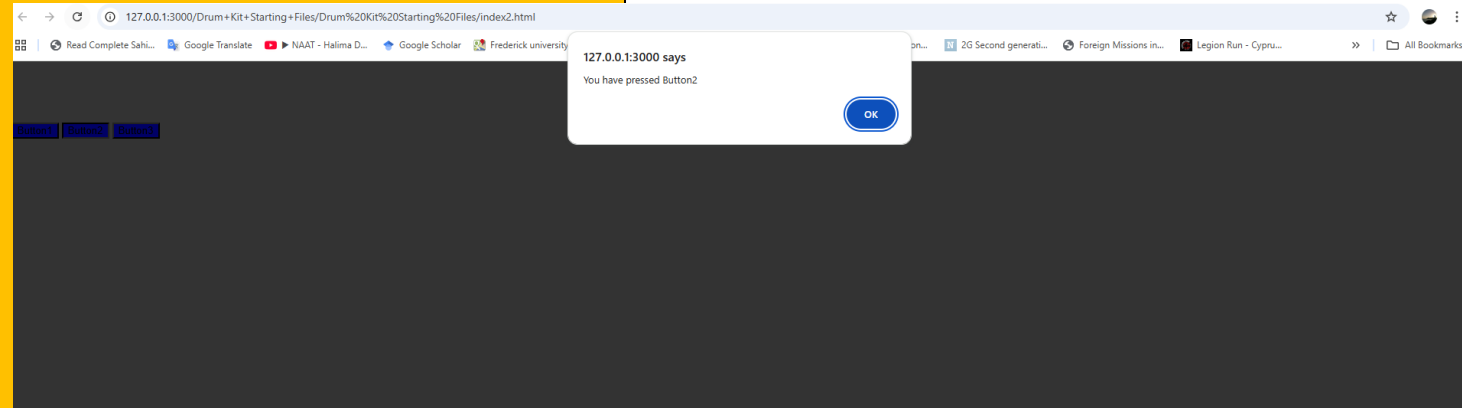


# Switch statement

```
numberOfButtons = document.querySelectorAll("button").length;
for (var i=0;i<numberOfButtons;i++) {
document.querySelectorAll("button")[i].addEventListener("click", function (){
var selectedButton = this.innerHTML;
switch (selectedButton) {
case "Button1":
 alert("You have pressed "+selectedButton);
 break;
case "Button2":
 alert("You have pressed "+selectedButton);
 break;

default:

 break;
}
})
}
```





# JavaScript Objects

Organizing Data the Smart Way

## Way 1: Variables (Not Recommended)

```
var studentName = 'Abid';
var studentAge = 21;
var studentSemester = 7;
var studentHobbies = ['Cycling', 'Driving', 'Hiking'];
var studentDepartment = 'Computer Engineering';
var studentUniversity = 'CUI ATD';
```

- Hard to manage
- Scattered & difficult to update

## Way 2: Object Literal (Recommended)

```
var student = {
 name: 'Abid',
 age: 21,
 semester: 7,
 hobbies: ['Cycling', 'Driving', 'Hiking'],
 department: 'Computer Engineering',
 university: 'CUI ATD'
};
```

- All related data together
- Simple & clean structure

## What is an Object?

- Collection of key-value pairs
- Represents real-world entities
- Stores properties + behaviors/methods

Example: Student, Car, Mobile Phone

## Why Use Objects?

- Group related data
- Avoid too many variables
- Easy to manage & reuse
- Foundation of OOP & Web Apps

## Accessing Object Properties

- `student.name` // Dot notation
- `student['age']` // Bracket notation

```
var student = {
 name: 'Abid',
 age: 21,
 semester: 7,
 hobbies: ['Cycling', 'Driving', 'Hiking'],
 department: 'Computer Engineering',
 university: 'CUI ATD'
};
```

```
> student.name
< 'Abid'

> student['name']
< 'Abid'
```

## Accessing Object Properties

Brackets allow dynamic property names

- **Bracket notation** allows you to access object properties **using variables**, not just fixed text.

```
var student = {
 name: 'Abid',
 age: 21,
 semester: 7,
 hobbies: ['Cycling', 'Driving', 'Hiking'],
 department: 'Computer Engineering',
 university: 'CUI ATD'
};
```

```
> var abc = "name";
← undefined
```

# Accessing Object Properties

Brackets allow dynamic property names

- **Bracket notation** allows you to access object properties **using variables**, not just fixed text.

```
var student = {
 name: 'Abid',
 age: 21,
 semester: 7,
 hobbies: ['Cycling', 'Driving', 'Hiking'],
 department: 'Computer Engineering',
 university: 'CUI ATD'
};
```

```
> var abc = "name";
```

```
< undefined
```

```
> student.abc
```

```
< undefined
```



# Accessing Object Properties

Brackets allow dynamic property names

- **Bracket notation** allows you to access object properties **using variables**, not just fixed text.

```
var student = {
 name: 'Abid',
 age: 21,
 semester: 7,
 hobbies: ['Cycling', 'Driving', 'Hiking'],
 department: 'Computer Engineering',
 university: 'CUI ATD'
};
```

```
> var abc = "name";
```

```
< undefined
```

```
> student.abc
```

```
< undefined
```

```
> student[abc]
```

```
< 'Abid'
```

## Updating Object Properties

Objects are mutable

- **Mutable** means:
  - Objects can be **changed after they are created.**
- This includes:
  - ✓ Adding new properties
  - ✓ Updating existing properties
  - ✓ Deleting properties

## Updating Object Properties ( Adding New Property )

Objects are mutable

- **Mutable** means:
  - Objects can be **changed after they are created**.
- This includes:
  - ✓ Adding new properties
  - ✓ Updating existing properties
  - ✓ Deleting properties

```
> student
< {name: 'Abid', age: 20, hobbies: Array(3), department: 'Computer Engineering'}

> student.cgpa =3.0
< 3

> student
< {name: 'Abid', age: 20, hobbies: Array(3), department: 'Computer Engineering', cgpa: 3}
```

## Updating Object Properties ( Updating Existing Property )

Objects are mutable

- **Mutable** means:
  - Objects can be **changed after they are created**.
- This includes:
  - ✓ Adding new properties
  - ✓ Updating existing properties
  - ✓ Deleting properties

```
> student
```

```
< ▶ {name: 'Abid', age: 21, hobbies: Array(3), department: 'Computer Engineering'}
```

```
> student.age = 20;
```

```
< 20
```

```
> student
```

```
< ▶ {name: 'Abid', age: 20, hobbies: Array(3), department: 'Computer Engineering'}
```

## Updating Object Properties ( Deleting Property )

Objects are mutable

- **Mutable** means:
  - Objects can be **changed after they are created**.
- This includes:
  - ✓ Adding new properties
  - ✓ Updating existing properties
  - ✓ Deleting properties

```
> student
```

```
< ▶ {name: 'Abid', age: 20, hobbies: Array(3), department: 'Computer Engineering', cgpa: 3}
```

```
> delete student.department
```

```
< true
```

```
> student
```

```
< ▶ {name: 'Abid', age: 20, hobbies: Array(3), cgpa: 3}
```

## Nested Objects

```
var student = {
 name: 'Abid',
 address: {
 city: 'Abbottabad',
 country: 'Pakistan'
 }
};
```

```
> student.address.city
```

```
< 'Abbottabad'
```

```
> student.address.country
```

```
< 'Pakistan'
```

## Creating Multiple Objects

- `var student1 = { name: 'Abid', age: 21 };`
- `var student2 = { name: 'Ali', age: 22 };`
- ✓ Good but not scalable

## Constructor Function

```
function StudentCreator(name, age, semester) {
 this.name = name;
 this.age = age;
 this.semester = semester;
}
```

- `var s1 = new StudentCreator('Abid',21,7);`
- `var s2 = new StudentCreator('hamza',22,6);`



## Constructor Function

```
function StudentCreator(name, age, semester) {
 this.name = name;
 this.age = age;
 this.semester = semester;
}
```

```
> var s1= new StudentCreator("hamza",20,6)
< undefined
> s1
< StudentCreator {name: 'hamza', age: 20, semester: 6} i
 age: 20
 name: "hamza"
 semester: 6
 ▶ [[Prototype]]: Object
```

- `var s1 = new StudentCreator('Abid',21,7);`
- `var s2 = new StudentCreator('hamza',22,6);`

## Object Methods

- This object has:
- A property → name
- A function → greet() (this is called a **method**)

```
var student = {
 name: 'Abid',
 greet() {
 console.log('Hello ' + this.name);
 }
};
```

```
> student.greet()
```

- this → refers to the same object

## Object Methods

- This object has:
- A property → name
- A function → greet() (this is called a **method**)

```
var student = {
 name: 'Abid',
 greet() {
 console.log('Hello ' + this.name);
 }
};
```

- this → refers to the same object

```
> student.greet()
Hello Abid
```

## Constructor Function with Method ()

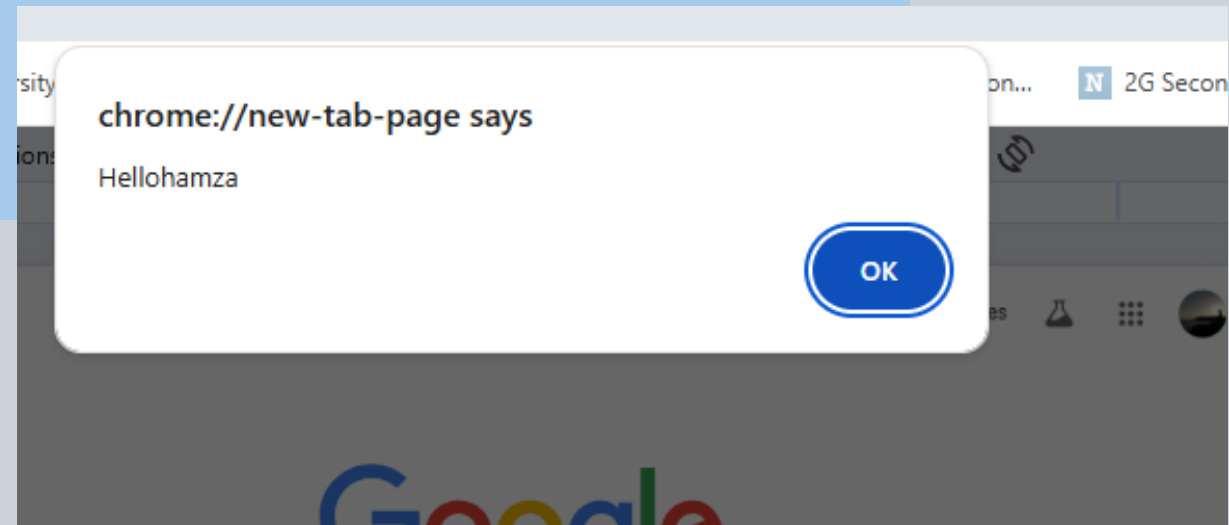
```
function StudentCreator(name, age, semester) {
 this.name = name;
 this.age = age;
 this.semester = semester;
 this.greet = function() { alert("Hello" + this.name);
 }
}
```

```
> var s1= new StudentCreator("hamza",20,6)
← undefined
> s1.greet()
```

## Constructor Function with Method ()

```
function StudentCreator(name, age, semester) {
 this.name = name;
 this.age = age;
 this.semester = semester;
 this.greet = function() { alert("Hello"+this.name);
 }
}
```

```
> var s1= new StudentCreator("hamza",20,6)
← undefined
> s1.greet()
← undefined
```



# Callback Functions

- A **function passed as an argument** to another function
- It **runs later** when the event or task completes
- Helps in **asynchronous** programming and event handling
- Common in:
  - Buttons click
  - keydown events
  - Timers
  - AJAX responses
- Here, the anonymous function is a *callback*, executed only when click happens.

```
button.addEventListener("click", function() {
 console.log("Button clicked!");
});
```

## Why Use Callback Functions?

- Make code **responsive** to user actions
- Allow **reuse** of functions
- Avoid running code too early
- Foundation of **jQuery** and **JavaScript events**
- Used for:
  - Animations
  - API calls
  - Timing (setTimeout)

```
setTimeout(() => {
 console.log("Runs after 2 seconds");
}, 2000);
```