

# 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

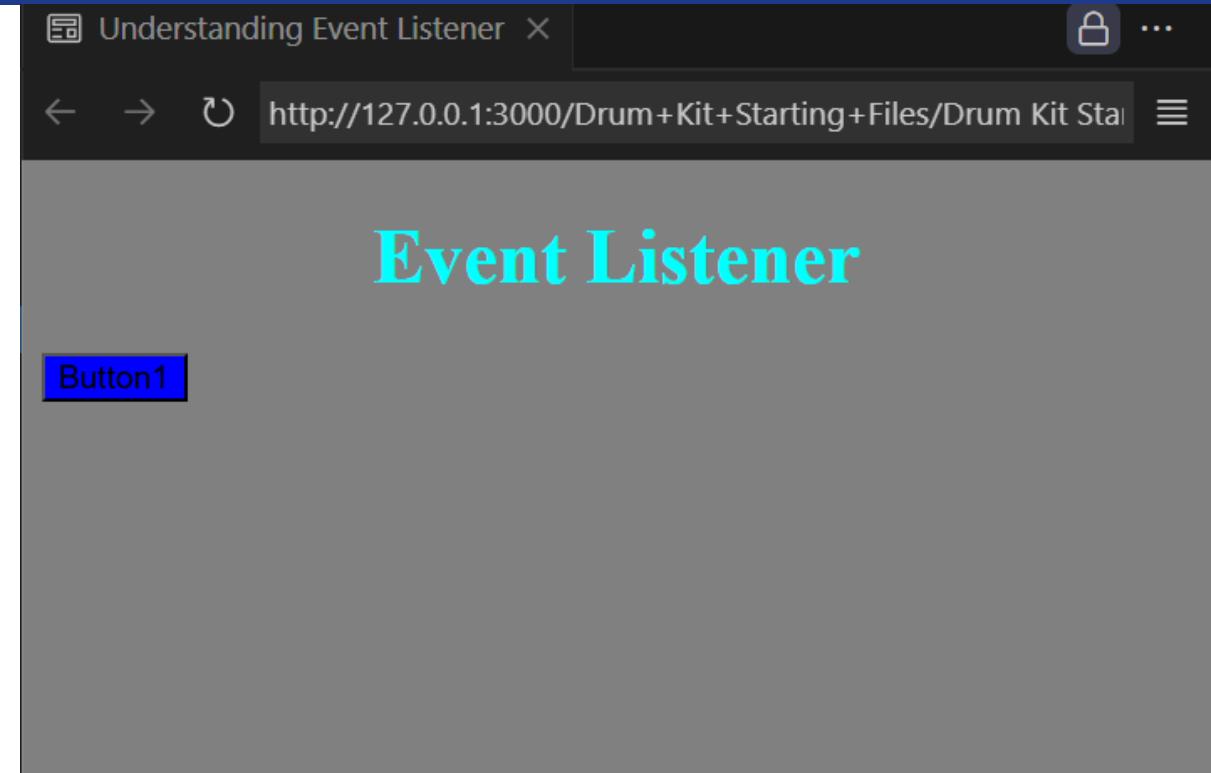
**Muhammad Naveed Shaikh**  
Department of Computer Engineering  
[naveedshaikh@cuiatd.edu.pk](mailto:naveedshaikh@cuiatd.edu.pk)



# 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.

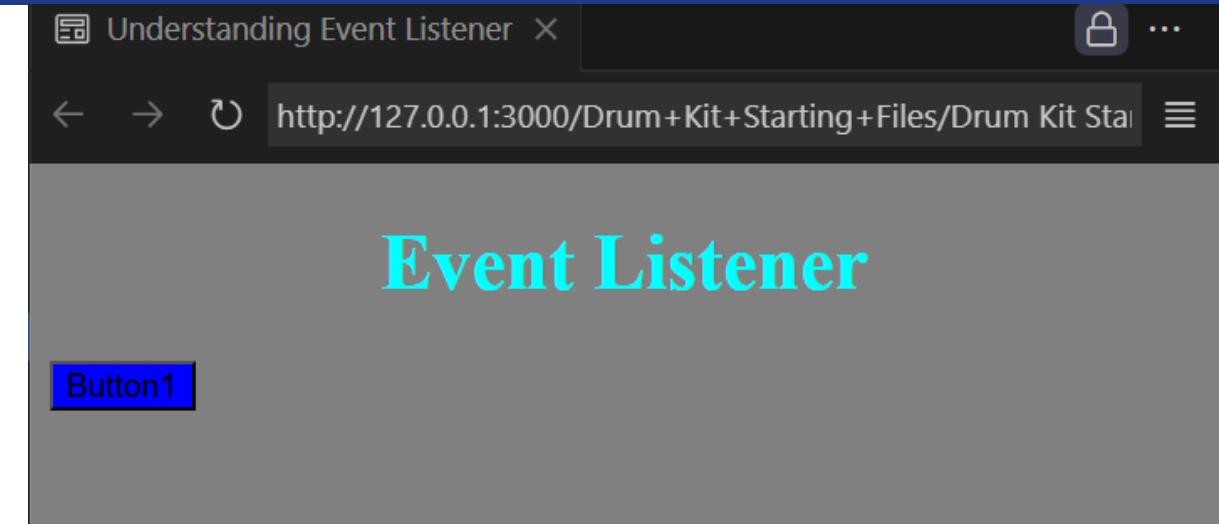
```
target.addEventListener(type, listener);
```

- 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

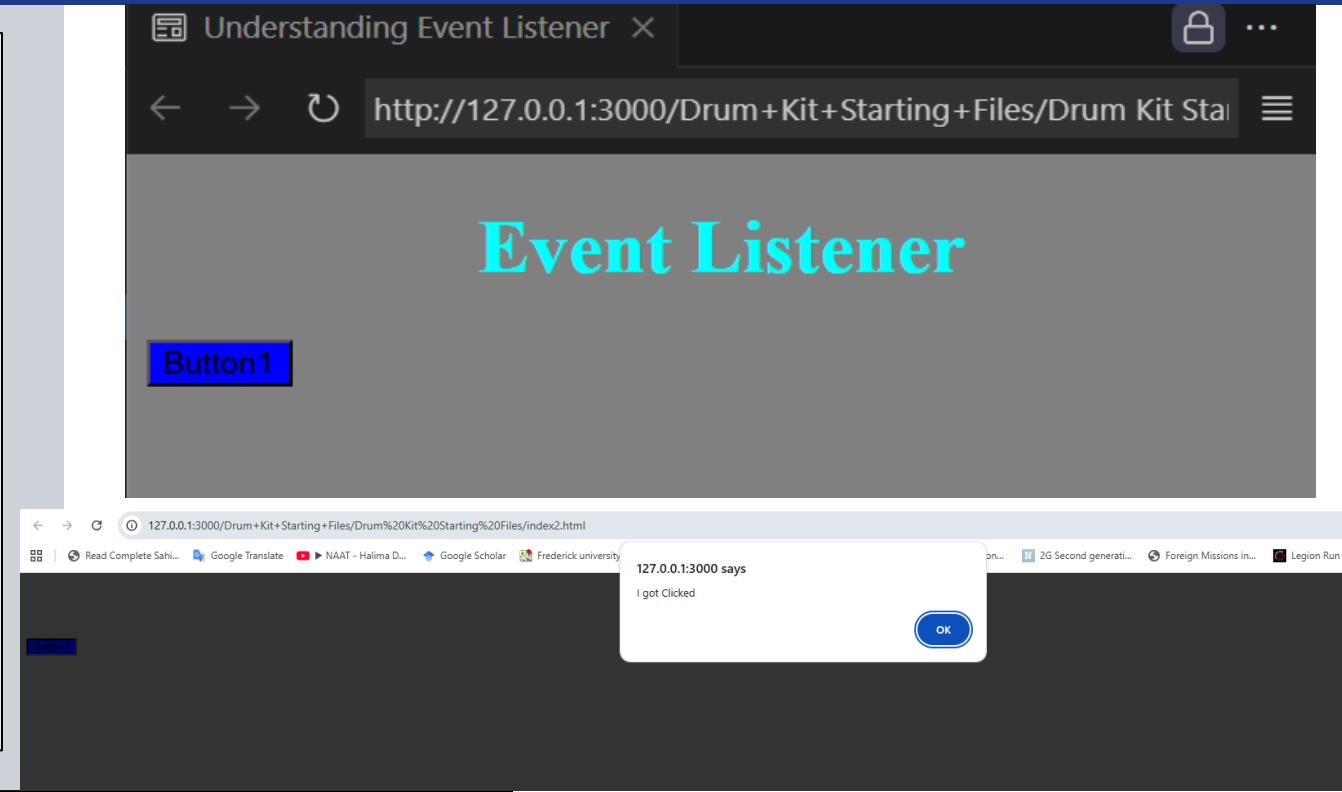
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <Button>Button1</Button>
<script src="index.js"></script>
</body>
</html>
```



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

# Understanding Event Listener

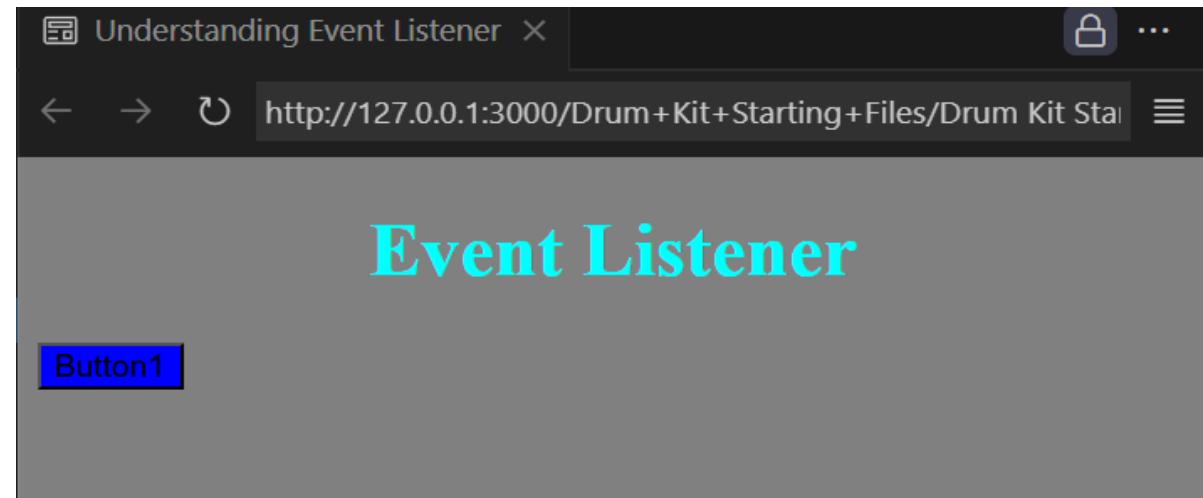
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Understanding Event Listener</title>
  <link rel="stylesheet" href="style2.css">
</head>
<body>
  <H1>Event Listener</H1>
  <Button>Button1</Button>
<script src="index.js"></script>
</body>
</html>
```



```
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>
```

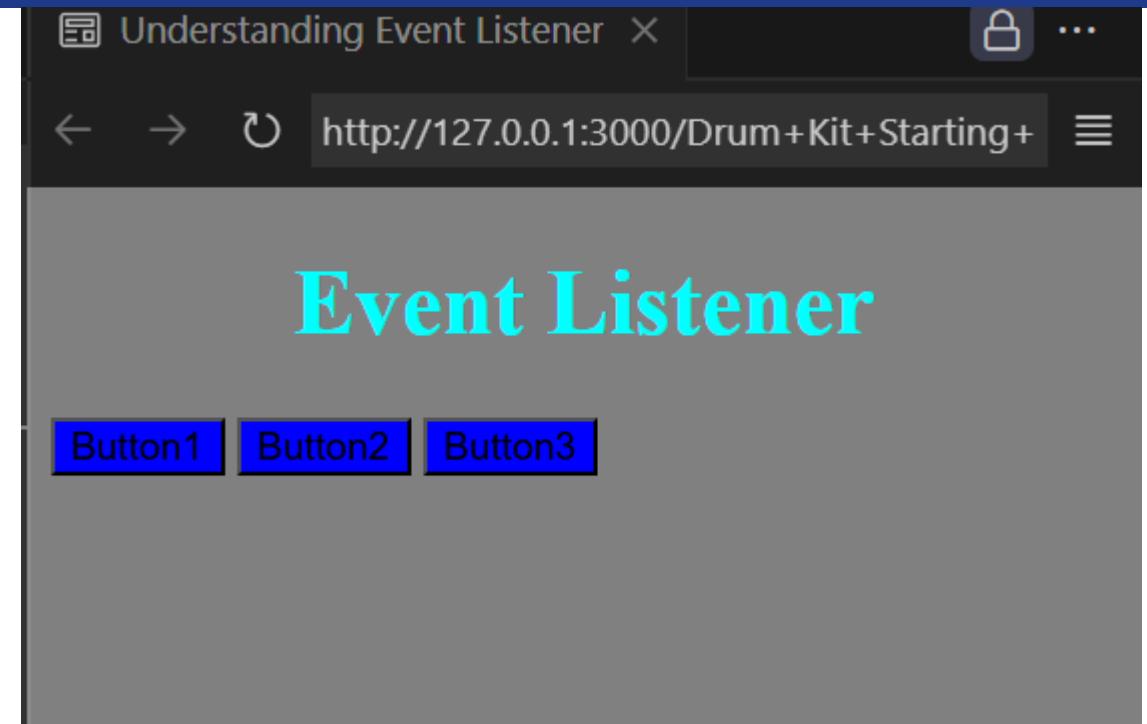


```
document.querySelector("button").addEventListener("click", function () {
  alert("I got Clicked");
})
```

```
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>
```

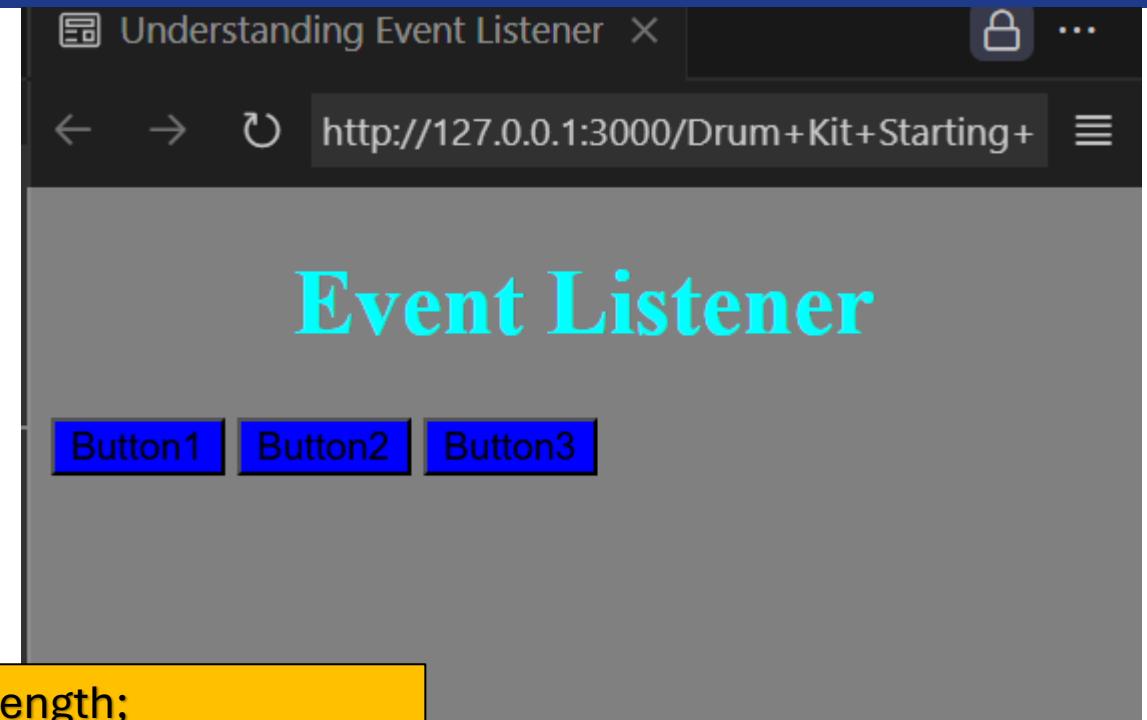


```
document.querySelector("button").addEventListener("click", function () {  
    alert("I got Clicked");  
})
```

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

## Dealing with multiple Event Listeners

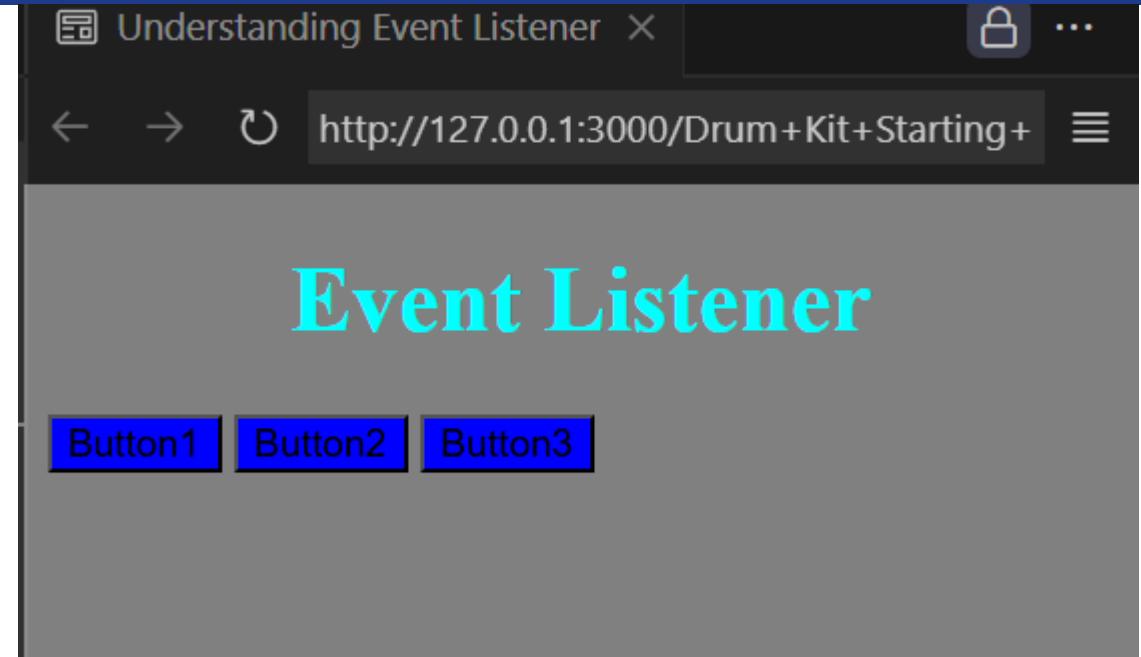
```
<body>  
  
<h1>Event Listener</h1>  
<button>Button1</button>  
<button>Button2</button>  
<button>Button3</button>  
<script src="index2.js" charset="utf-8">  
  
</script>  
</body>
```



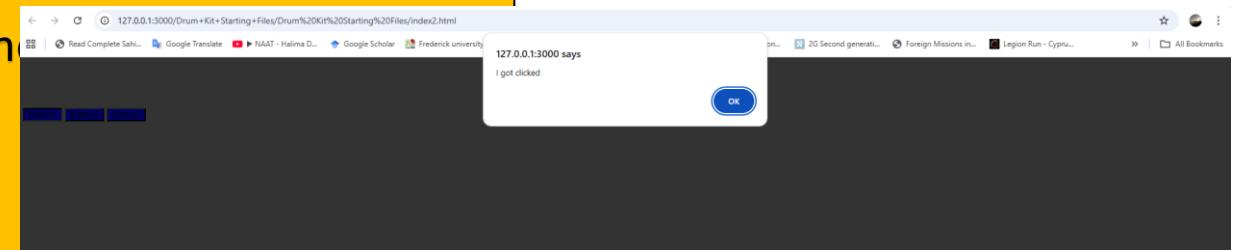
```
numberOfButtons = document.querySelectorAll("button").length;  
for (var i=0;i<numberOfButtons;i++) {  
document.querySelectorAll("button")[i].addEventListener("click", function (){  
    alert("I got Clicked");  
})  
}
```

# 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("click",function(){  
    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:

The screenshot shows a browser developer tools interface with the following details:

- Left Panel (Code View):** Shows two tabs: "index.js" and "VM913 X". The "VM913 X" tab is active, displaying the following code:

```
1 | debugger;
2 | calculator(2,3,add);
```

The first line, "debugger;", is highlighted with a yellow background.
- Right Panel (Debugger Tools):** A sidebar with the following sections:
  - Debugger paused** (highlighted in yellow)
  - Threads
  - Watch
  - Breakpoints
  - Pause on uncaught exceptions
  - Pause on caught exceptions
  - Scope
    - Global (selected) Window
  - Call Stack
    - (anonymous) VM913:1 (selected)
    - XHR/fetch Breakpoints
    - DOM Breakpoints
    - Global Listeners
    - Event Listener Breakpoints
    - CSP Violation Breakpoints

## Debugging Step-by-Step

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

The screenshot shows a browser developer tools debugger interface. On the left, there are two tabs: "index.js" and "VM913". The "VM913" tab is active, displaying the following code:

```
1 debugger;
2 calculator(2,3,add);
```

The first line, "debugger;", is highlighted with a yellow background. On the right, a sidebar titled "Debugger paused" is open, showing various debugging options and information. The "Scope" section is expanded, showing "Global" and "Window". The "Call Stack" section is also expanded, showing a single entry: "(anonymous) VM913:1". Other sections like "Threads", "Watch", and "Breakpoints" are listed but not expanded.

# Debugging Step-by-Step

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

The screenshot shows a browser developer tools debugger interface. On the left, the code editor displays `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 line `operator(num1,num2);` is highlighted with a yellow background.

The right panel shows the debugger controls and the state of the variables:

- Debugger status: `Debugger paused`
- Threads: None
- Watch: None
- Breakpoints:
  - `debugger;` at VM913:1
  - `calculator` at VM928:2
- Pause options:
  - 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`
- Other sections: 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 shows a browser's developer tools debugger. On the left, the code editor displays `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 second line, `return (num1+num2)`, is highlighted with a yellow background. In the top right corner of the code editor, there is a yellow box with the text `(i) Debugger paused`. To the right of the code editor is the debugger sidebar, which includes the following sections:

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

# Debugging Step-by-Step

- Type in code
- And, click

The screenshot shows a browser developer tools debugger interface. On the left is the code editor window titled "index.js X" with tabs "VM913" and "VM928". The code contains three functions: "add", "subtract", and "calculator". The "add" function is highlighted with a yellow background. The "calculator" function call `calculator(2, 3)` has been stepped into, as indicated by the yellow highlighting of its opening brace. The right side of the interface is the debugger sidebar, which includes a toolbar with icons for step operations, a status message "Debugger paused", and sections for "Threads", "Watch", "Breakpoints", "Pause on uncaught exceptions", "Pause on caught exceptions", and "Scope". The "Scope" section is expanded, showing the "Local" environment with variables `num1` (2), `num2` (3), and `this` (Window). The "Global" environment is also listed under "Scope". The "Call Stack" section shows the current stack trace: `add` at index.js:2, `calculator` at index.js:8, and `(anonymous)` at VM928:2. Other sections like "XHR/fetch Breakpoints", "DOM Breakpoints", "Global Listeners", "Event Listener Breakpoints", and "CSP Violation Breakpoints" are also present.

```
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 }
```

Debugger paused

Threads

Watch

Breakpoints

Pause on uncaught exceptions

Pause on caught exceptions

Scope

Local

Return value: 5

this: Window

num1: 2

num2: 3

Global Window

Call Stack

add index.js:2

calculator index.js:8

(anonymous) 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, click

The screenshot shows a browser developer tools debugger interface. On the left, the code editor displays `index.js` with the following content:

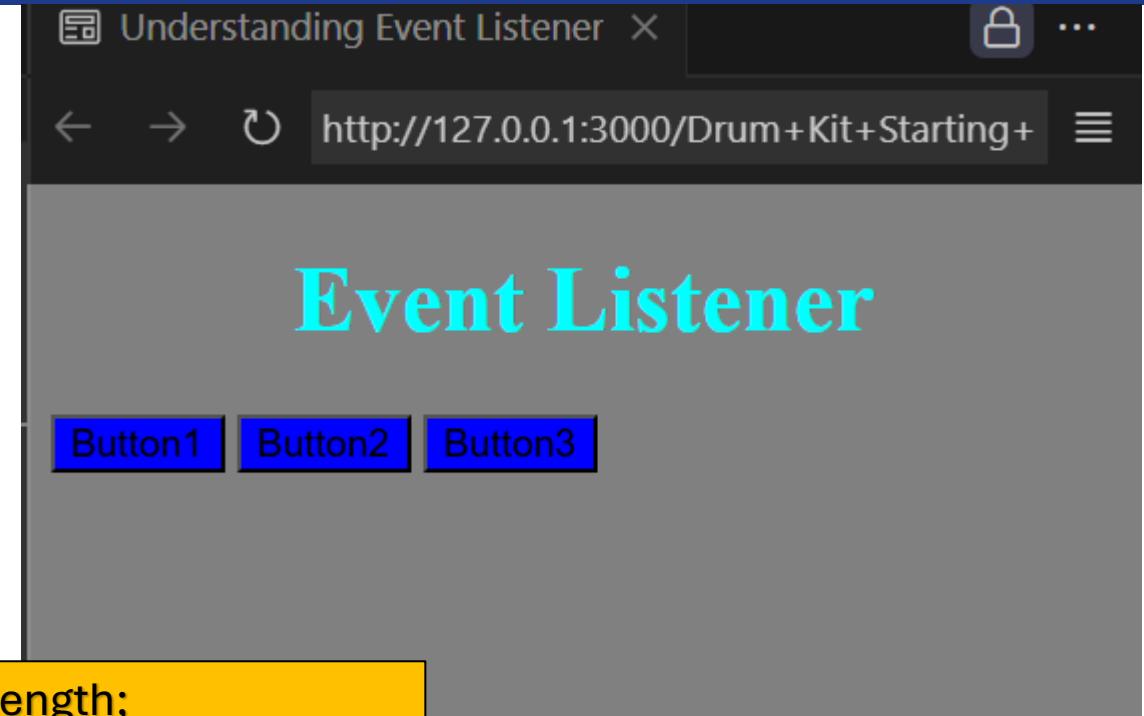
```
1 function add(num1,num2){  
2     return (num1+num2)  
3 };  
4 function subtract(num1,num2) {  
5     return (num1-num2);  
6 };  
7 function calculator(num1,num2,operator) { num1 = 2, num2 = 3, operator = f add(num1,num2)  
8     return operator(num1,num2);  
9 }
```

The line `operator = f add(num1,num2)` is highlighted with a yellow background. The right side of the interface shows the debugger sidebar with the following details:

- Debugger paused** status indicator.
- Threads**: No threads listed.
- Watch**: No items listed.
- Breakpoints**: No breakpoints listed.
- Pause on uncaught exceptions**: Unchecked checkbox.
- Pause on caught exceptions**: Unchecked checkbox.
- Scope** section:
  - Local**:
    - Return value: 5
    - this: Window
    - num1: 2
    - num2: 3
    - operator: f add(num1,num2)
  - Global: Window
- Call Stack**:
  - calculator index.js:8
  - (anonymous) VM928:2
- XHR/fetch Breakpoints**
- DOM Breakpoints**
- Global Listeners**
- Event Listener Breakpoints**
- CSP Violation Breakpoints**

## 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>
```

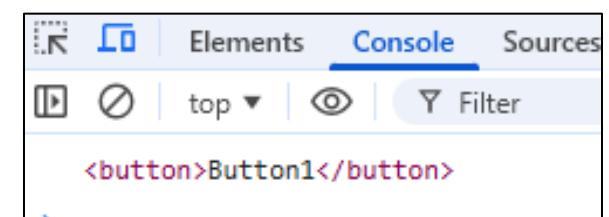
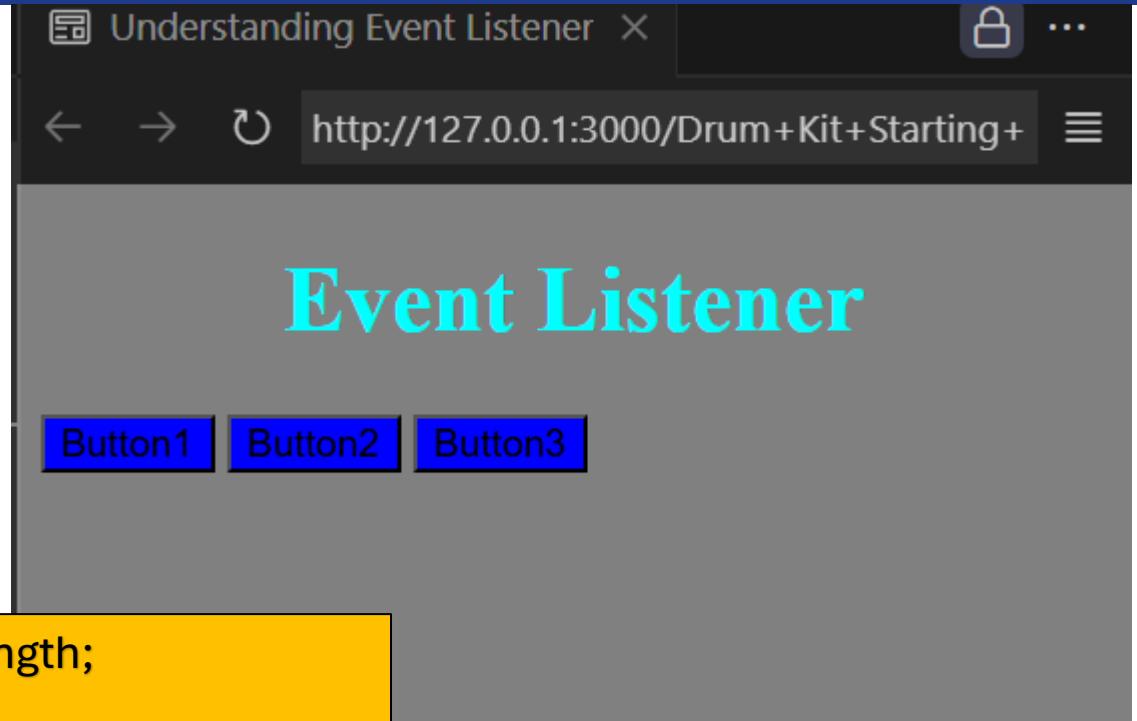


```
numberOfButtons = document.querySelectorAll("button").length;  
for (var i=0;i<numberOfButtons;i++) {  
document.querySelectorAll("button")[i].addEventListener("click", function (){  
  console.log(this);  
})  
}
```

# 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>
```

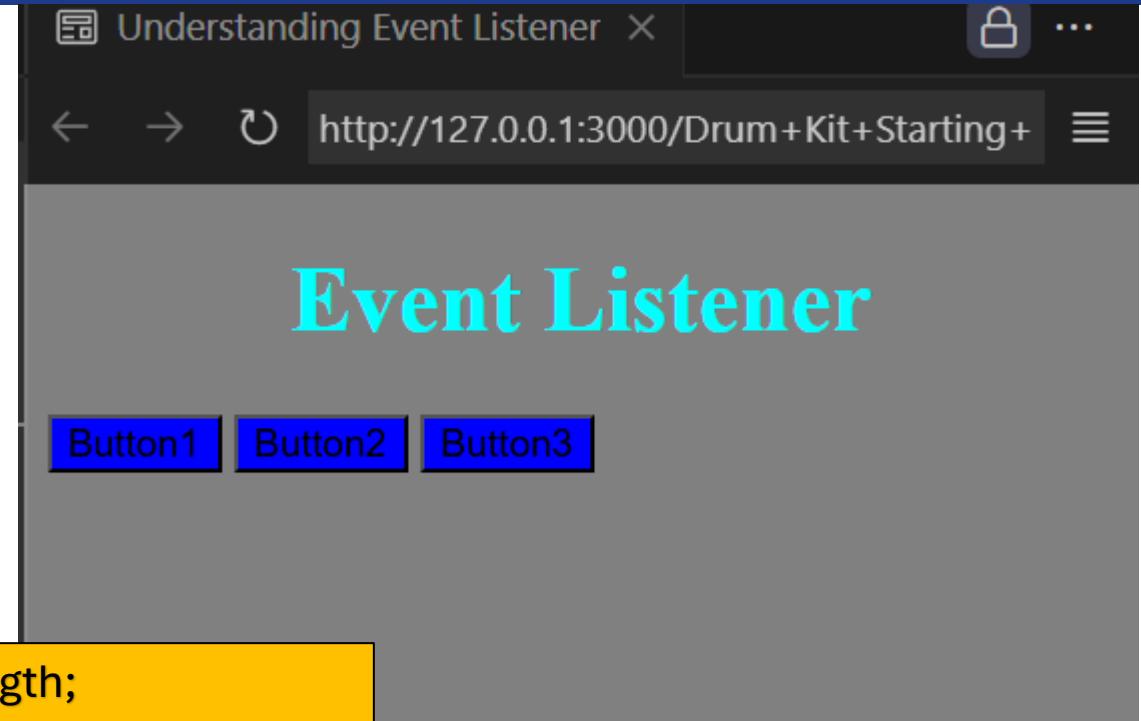
```
numberOfButtons = document.querySelectorAll("button").length;  
for (var i=0;i<numberOfButtons;i++) {  
document.querySelectorAll("button")[i].addEventListener("click", function (){  
    console.log(this);  
})  
}
```



This is the output in console if Button1 is pressed

## JavaScript this Keyword

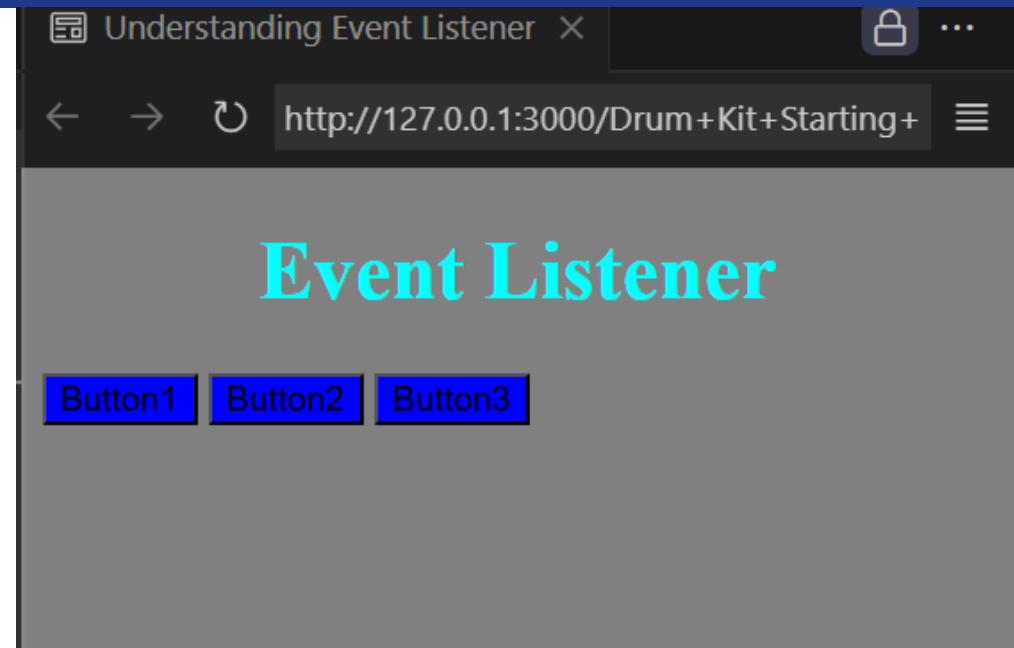
```
<body>  
  
<h1>Event Listener</h1>  
<button>Button1</button>  
<button>Button2</button>  
<button>Button3</button>  
<script src="index2.js" charset="utf-8">  
  
</script>  
</body>
```



```
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>
```



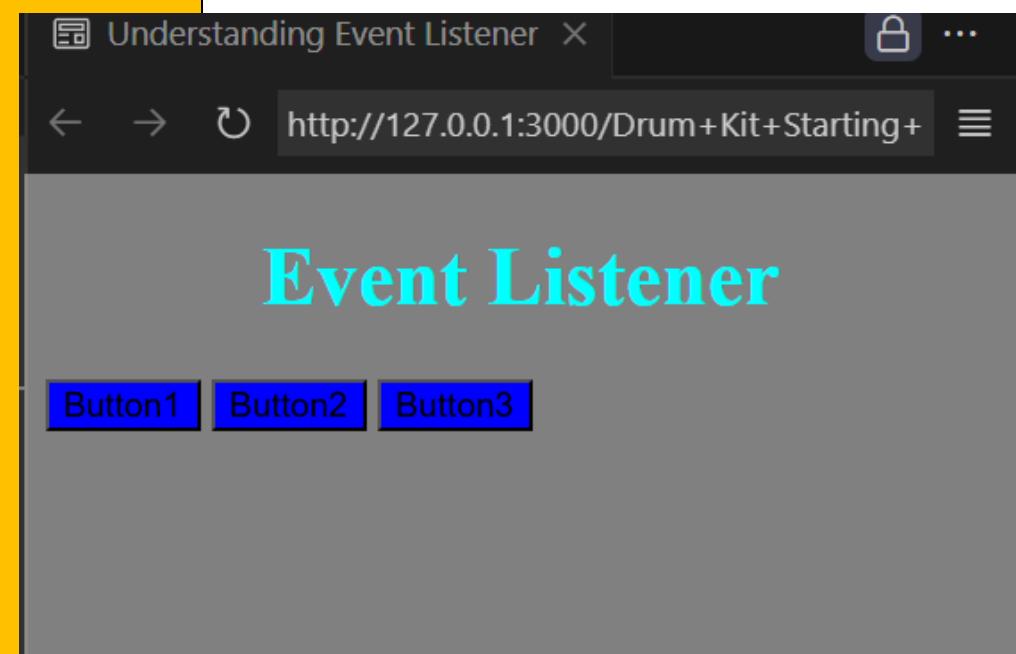
```
numberOfButtons = document.querySelectorAll("button").length;  
for (var i=0;i<numberOfButtons;i++) {  
document.querySelectorAll("button")[i].addEventListener("click", function (){  
    console.log(this.innerHTML);})  
}
```

This is the output in console

```
Button2  
Button3  
Button2  
Button1
```

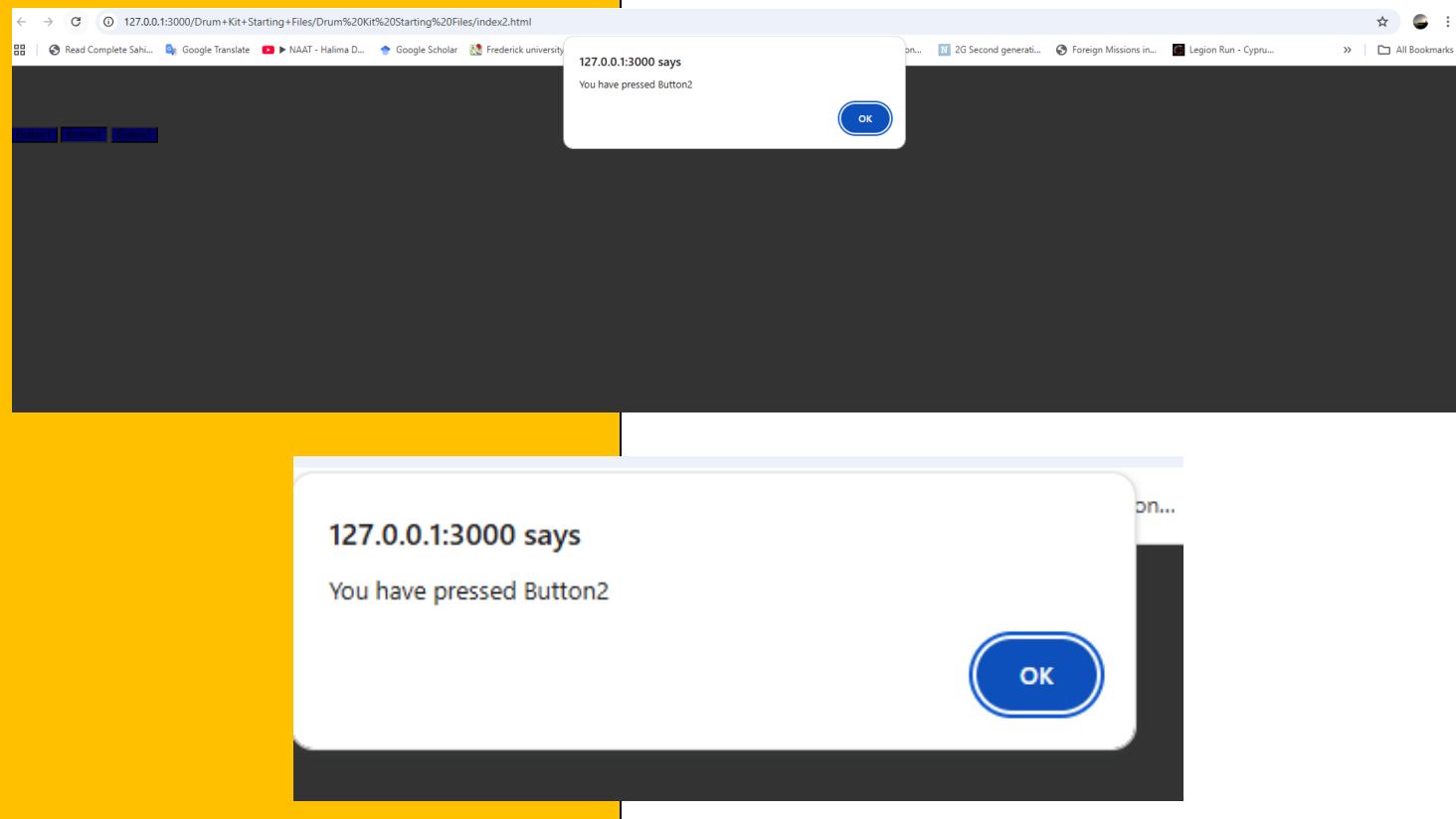
## Switch statement

```
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} ⓘ  
    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);  
    }  
};
```

➤ `student.greet()`  

---

`Hello Abid`

---

- `this` ➔ refers to the same object

## Constructor Function with Method ()

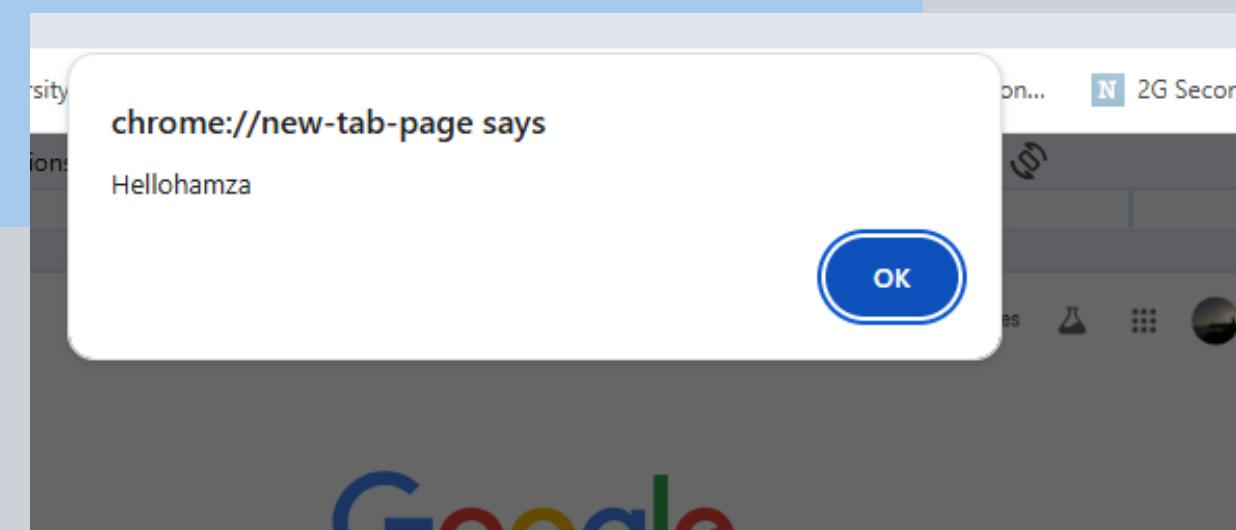
```
function StudentCreator(name, age, semester) {  
    this.name = name;  
    this.age = age;  
    this.semester = semester;  
    this.greet =function() { alert("Hello"+this.name);  
    }  
}
```

```
> 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);
```