COMP 206:
WEB PROGRAMMING

LECTURE #2

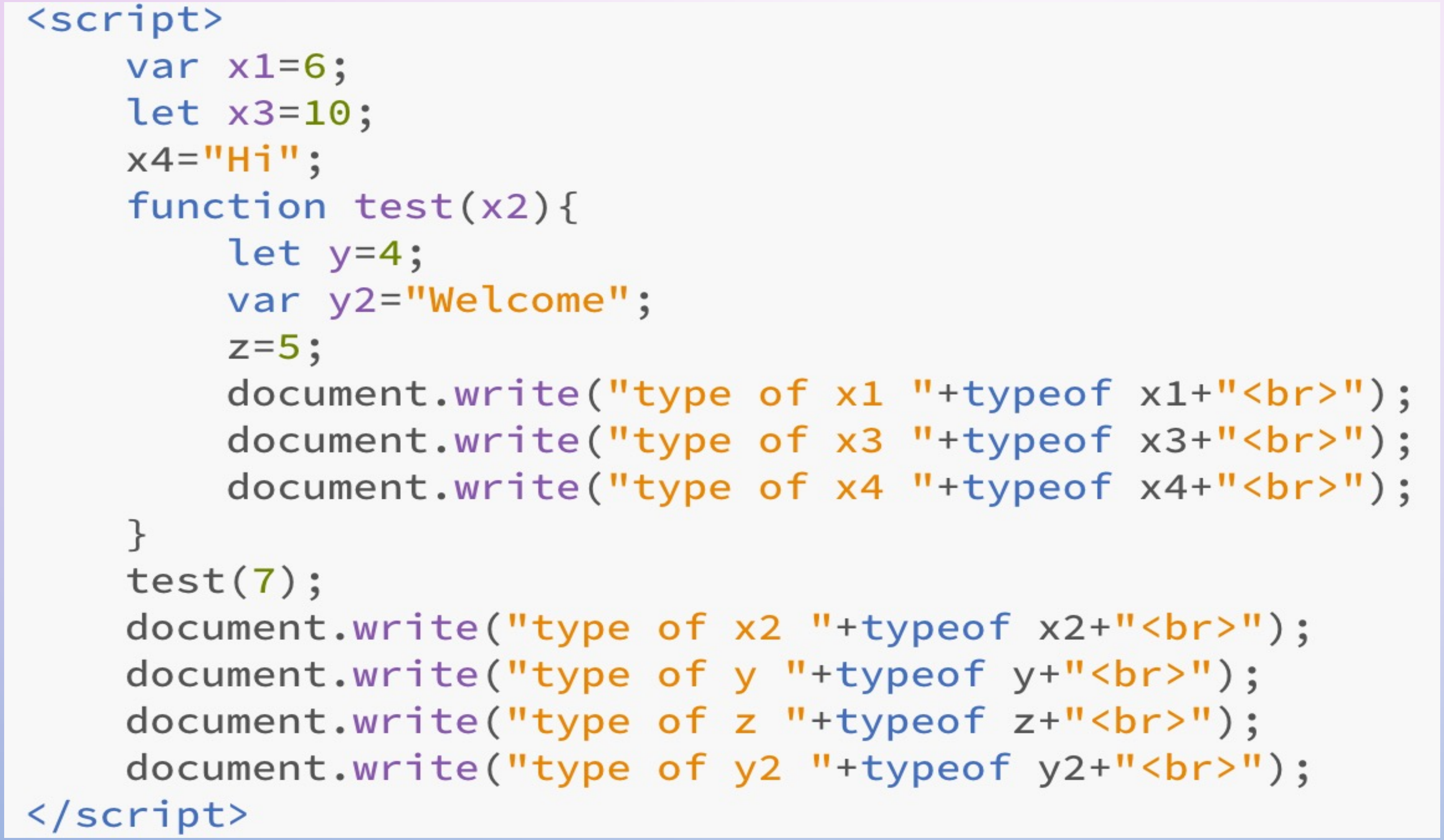
Dr. Nashwa Abdelghaffar

nashwa.abdelghaffar@sci.asu.edu.eg

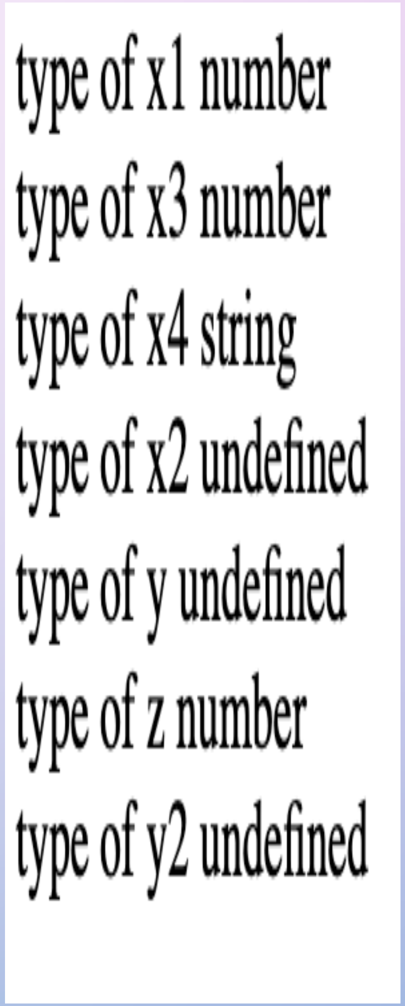
OUTLINE

- Variables Scope and Hoisting
- Functions
- Functions Hoisting
- Objects and classes
- Events

VARIABLES SCOPE



```
<script>
    var x1=6;
    let x3=10;
    x4="Hi";
    function test(x2){
        let y=4;
        var y2="Welcome";
        z=5;
        document.write("type of x1 "+typeof x1+"<br>");
        document.write("type of x3 "+typeof x3+"<br>");
        document.write("type of x4 "+typeof x4+"<br>");
    }
    test(7);
    document.write("type of x2 "+typeof x2+"<br>");
    document.write("type of y "+typeof y+"<br>");
    document.write("type of z "+typeof z+"<br>");
    document.write("type of y2 "+typeof y2+"<br>");
</script>
```



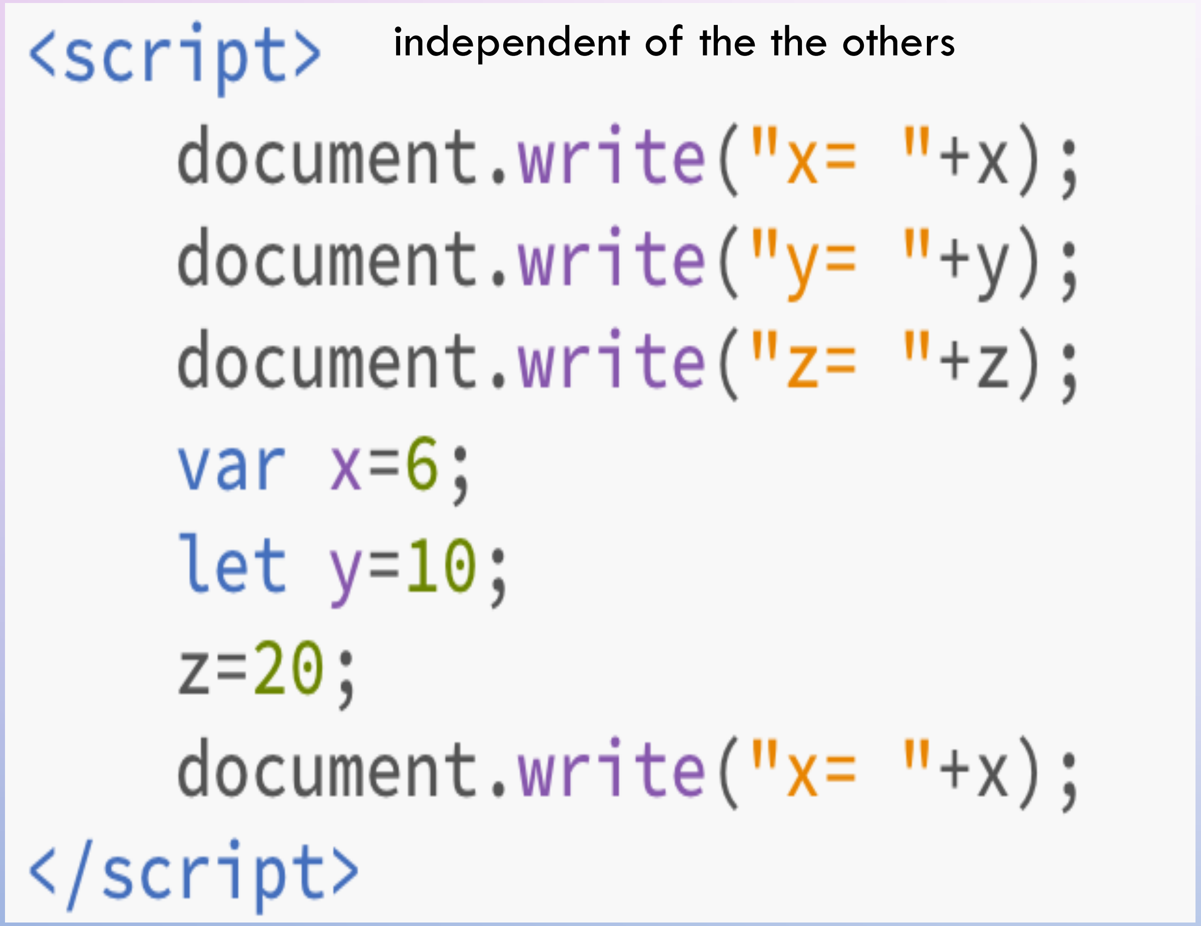
```
type of x1 number
type of x3 number
type of x4 string
type of x2 undefined
type of y undefined
type of z number
type of y2 undefined
```

VARIABLE HOISTING

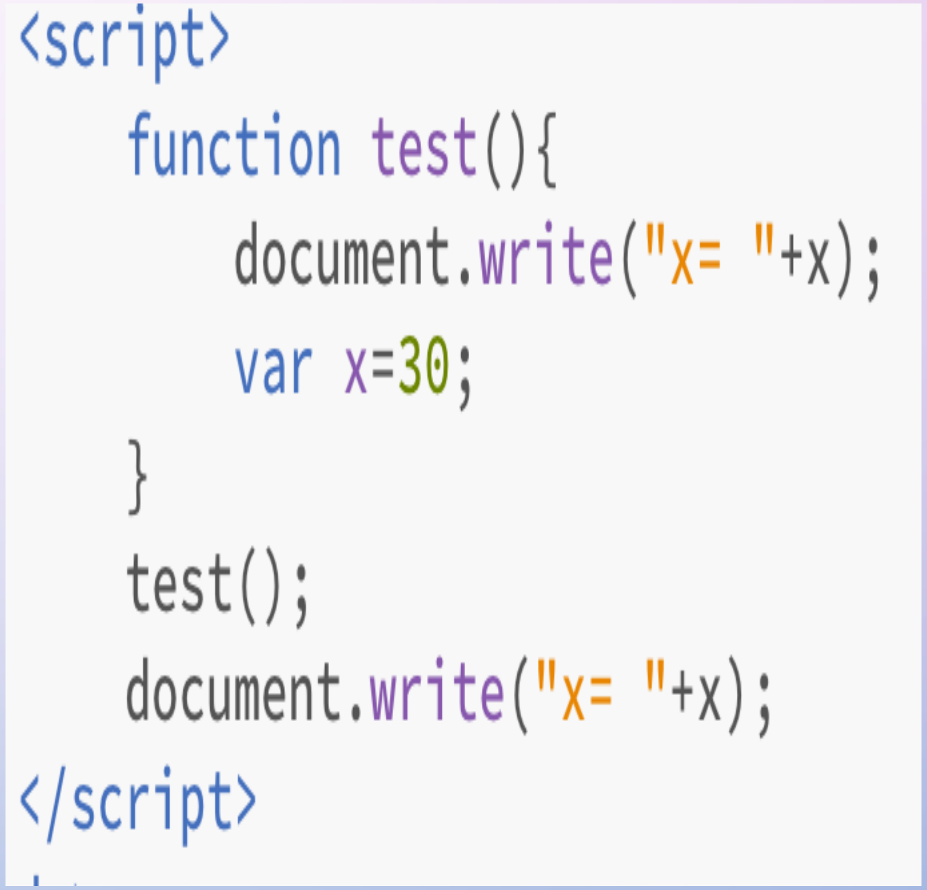
- Hoisting is a concept or behavior in JavaScript where the declaration of a function, variable, or class goes to the top of the scope they were defined in.
- You can declare variables in JavaScript with the **var**, **let**, and **const** variables. And these variable declarations would be hoisted but behave differently.
- With variables declared **var**, the variable declaration is hoisted but with a default value of ”*undefined*”. The actual value is initialized when the declaration line is executed.
- With variables declared **let** and **const**, the variable declaration is hoisted but with having no default value. The following error ”*cannot access variablename before initialization*" occurs.

VARIABLE HOISTING (EXAMPLE)

Consider each write statement is independent of the the others



```
<script>
    document.write("x= "+x);
    document.write("y= "+y);
    document.write("z= "+z);
    var x=6;
    let y=10;
    z=20;
    document.write("x= "+x);
</script>
```



```
<script>
    function test(){
        document.write("x= "+x);
        var x=30;
    }
    test();
    document.write("x= "+x);
</script>
```

FUNCTIONS

The general syntax for a function:

```
function function-name([parameter[,…]]){
Statements
}
```

There is a general naming convention for functions. The first letter of each word in a name is capitalized except for the very first letter, which is lower case. This convention is commonly referred to as CamelCase.

ANONYMOUS FUNCTIONS

- A function can be defined using a const function expression:

```
const x=function(a,b){return a+b;}
```

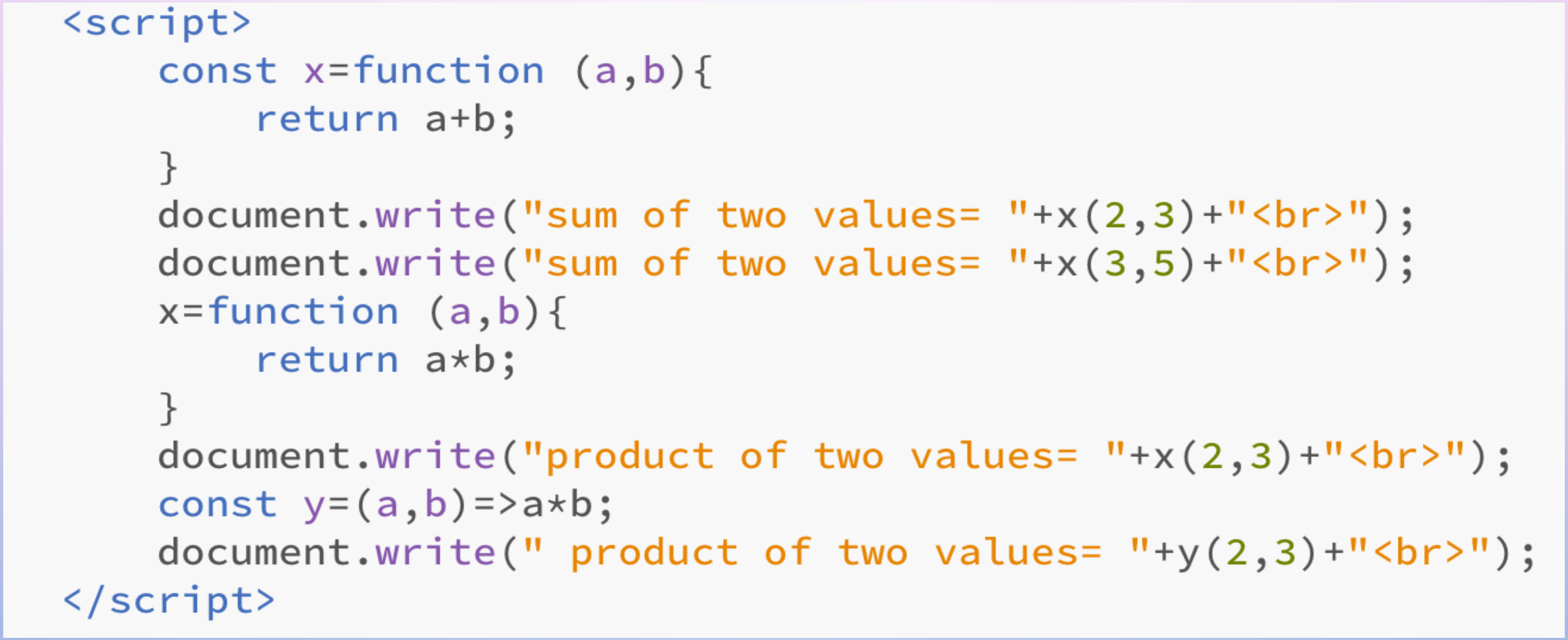
- After the function expression has been stored in a variable. The variable can then be used as a function.

```
let z=x(2,3);
```

- Another short syntax for writing function expression is called arrow functions.
- In this syntax, the function keyword, the return keyword, and the curly brackets can be omitted, if the function is a single statement.

```
const x=(a,b)=>a+b;
```

EXAMPLE



```
<script>
    const x=function (a,b){
        return a+b;
    }
    document.write("sum of two values= "+x(2,3)+"<br>");
    document.write("sum of two values= "+x(3,5)+"<br>");
    x=function (a,b){
        return a*b;
    }
    document.write("product of two values= "+x(2,3)+"<br>");
    const y=(a,b)=>a*b;
    document.write(" product of two values= "+y(2,3)+"<br>");
</script>
```

DEFAULT PARAMETERS FUNCTIONS

```
function add(a,b=10){
return a+b;
}
add(5); //calling a function
```

- A function definition can also be dealt with an indefinite number of arguments as an array.

```
function add(… args){
let sum=0;
for(let i of args) sum+=i;
return sum;
}
```

Rest Parameters

for…of loop

- The block of code inside of the **`for...of`** loop will be executed once for each argument value.

ARGUMENTS OBJECT

- Functions in JS have a built-in object called the **`arguments`** object. It contain an array of the arguments used when the function is invoked.
- The `arguments` array gives the flexibility to handle a variable number of arguments.

```
function display(){
 for(j=0;j<display.arguments.length;j++)
  document.write(display.arguments[j]+” <br>”);
}
```

Calling a function: `display(“Dog”,”Cat”,”Hamster”);`

FUNCTION HOISTING

- Function hoisting is about calling functions before the line of its declaration.
- This is done by hoisting the declared function to the top of the scope it is defined in.

```
printHello();
function printHello(){
  document.write(“Hello”);
}
```

- Hoisting does not occur on function expressions.

NON-PRIMITIVE DATATYPE (OBJECTS)

- A **primitive value** is a value that has no properties or methods. For example, **3.14** is a primitive value
- A **primitive data type** is data that has a primitive value.
- An **object** groups data together with the functions needed to manipulate it.
- The data associated with an object is called **properties**, while the functions it used are called **methods**.
- All object instances have the same **properties and methods**, but the property **values** differ from instance to another, and methods are performed **at different times**.

CREATING OBJECTS

- There are 2 ways to create objects: by object literal and by creating instance of object directly (using `new` keyword)
- Objects are variables that can contain many values. Values are written as **name : value** pairs.
- A single object can be created using object literal.

```
 const person = {firstname:"Ahmed", lastname:"Mohammed",
age:50};
```

OR

```
const person{}
person.firstname="Ahmed"; person.lastname="Mohammed";
person.age=50;
```

OBJECTS (CONT.)

- Objects are mutable: They are addressed by reference, not by value.

```
const x = person;
```

- The object `x` is not a copy of `person`. It is a person.
- Both `x` and `person` are the same object.
- Any changes to `x` will also change `person`, because `x` and `person` are the same object.

```
x.age=30;  //The age of person will also be changed
```

ACCESSING PROPERTIES

- The syntax for accessing the property of an object is:

```
1. objectname.property      // person.age
2. objectname[”property"]  // person["age"]
3. objectname[expression] // x = "age"; person[x]
   for (let x in person) {
          txt += person[x];
     }
```

USING NEW KEYWORD

```
const person = new object();
person.firstname = ”Ahmed";
person.lastname = ”Ali";
person.age = 50;
```

For readability, simplicity, and execution speed, use the object literal method.

NESTED OBJECTS

```
myobj = {
  name:"Ahmed",
  age:30,
  cars: {
    car1:"Ford",
    car2:"BMW",
    car3:"Volvo"
  }
}
```

- Nested objects can be accessed using one of the following ways:

```
1 myobj.cars.car2; 2 myobj.cars["car2"]; 3 myobj["cars"]["car2"];
 let p1 = "cars"; let p2 = "car2"; myobj[p1][p2]; 4
```

OBJECT METHODS

Methods are functions stored as object properties.

```
const person = {
  firstname: ″Ahmed",
  lastname: ″Ali",
  age: 50,
  fullname: function() {
    return this.firstname + " " + this.lastname;
  }
};
```

This refers to an object and depends on the object being invoked

ACCESSING METHODS

```
objectname.methodname()
name = person.fullname();
```

- Accessing the `fullname` **property**, without `()`, it will return the **function definition**.
- Adding methods after object declaration:

```
 person.name = function () {
  return (this.firstname + " " + this.lastname);
 };
```

DISPLAYING OBJECTS

Some common solutions to display JavaScript objects are:

- Displaying the object properties by name
- Displaying the object properties in a loop
- Displaying the object using **`Object.values()`**

By a name

```
document.getElementById("demo").innerHTML =
person.firstname + "," + person.lastname + "," +
person.age;
```

DISPLAYING OBJECTS

In a loop

```
let txt = "";
for (let x in person) {
txt += person[x] + " ";
};
document.getElementById("demo").innerHTML = txt;
```

Using Object.values()

```
const myarray = Object.values(person);
document.getElementById("demo").innerHTML =
myarray;
```

CLASSES

- The examples from the previous slides are limited. They only create single objects.
- Sometimes we need a "**blueprint**" for creating many objects of the same "type".
- The way to create an "object type", is to use an **object constructor function**.
- Objects of the same type are created by calling the constructor function with the `new` keyword:

CREATING CLASSES

- Classes are in fact "special functions", and just as you can define function expressions and function declarations, a class can be defined in two ways: a class expression or a class declaration.
- Class Declaration:

```
class Rectangle {
   constructor(height, width){
      this.height = height; this.width = width;
   }
}
const r=new Rectangle(20,30);
```

CREATING CLASSES (CONT.)

Class Expression:

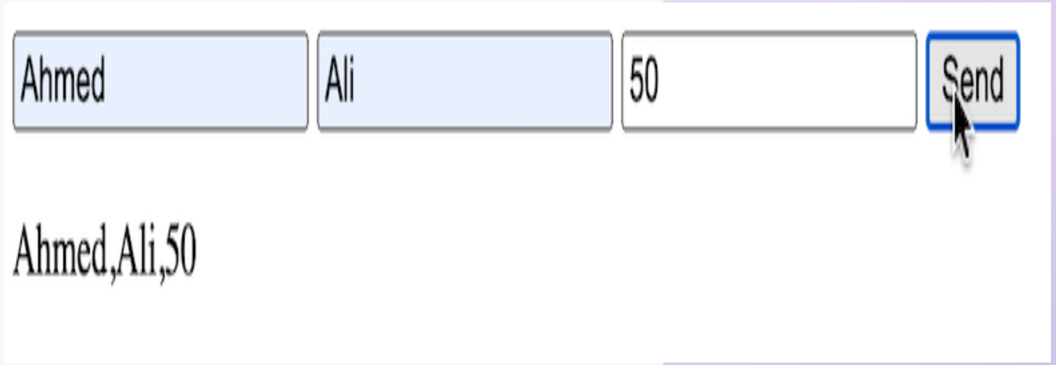
```
const Rectangle = class {
constructor(height, width) { this.height = height;
this.width = width; }
};
const r=new Rectangle(20,30);
```

```
const Rect = class Rectangle {
constructor(height, width){ this.height = height;
this.width = width;}
};
const r=new Rect(20,30);
```

EXAMPLE

```
<body>
    <form>
    <input id="fname" type="text" placeholder="First Name" name="fname">
    <input id="lname" type="text" placeholder="Last Name" name="lname">
    <input id="age"   type="number" placeholder="Age" name="age">
    <button  type="button" id ="send" onclick="process()">Send</button>
    </form>
    <p id="demo"></p>
    <script>
        class Person {
            constructor(firstname,lastname, age){
                this.firstname =firstname;
                this.lastname = lastname;
                this.age = age;
            }
        }
        function process(){
            const per=new Person();
            per.firstname=document.getElementById("fname").value;
            per.lastname=document.getElementById("lname").value;
            per.age=parseInt(document.getElementById("age").value);
            var myarray = Object.values(per);
            document.getElementById("demo").innerHTML = myarray;

        }
    </script>
</body>
```



```
Ahmed,Ali,50
```

EVENTS

- Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs.
- Events are fired inside the browser window and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window.
- The following are different types of events that can occur:

  The user selects, clicks, or hovers the cursor over a certain element,
  the user resizes or closes the browser window, a web page finishes loading, a form is submitted, or an error occurs.

EVENTS HANDLING

- To react to an event, you attach an event handler to it. This is a block of code that runs when the event fires.
- Event handlers are sometimes called event listeners.
- The listener listens out for the event happening, and the handler is the code that is run in response to it happening.
- There are different ways of registering event handlers: for example, event handler properties, inline event handlers, or registering event handler.

EVENTS HANDLING (CONT.)

- The HTML **button** element will fire an event when the user **clicks** the button.
- There are many different events that can be fired by a `button` element.
- **focus** and **blur**—when the button is focused and unfocused
- **dblclick** —when the button is double-clicked.
- **mouseover** and **mouseout** —when the mouse pointer hovers over the button, or when the pointer moves off the button, respectively.

EVENTS HANDLER PROPERTIES

- Objects (such as buttons) that can fire events also usually have properties whose name is on followed by the name of the event.

For example, elements have a property `onclick`. This is called an event handler property.

To listen for the event, you can assign the handler function to the property.

```
const btn = document.querySelector("button");

btn.onclick = () =>
{document.getElementById(“p”).innerHTML = “hello”; };
```

The `Document` method **`querySelector()`** returns the first `element` within the document that matches the specified selector

INLINE EVENT HANDLERS

```
<button onclick=”change()">Press me</button>

function change() {

document.getElementById(“p”).innerHTML = “hello”;

}
```

- Assigning event handlers using HTML event handler attributes is considered a bad practice and should be avoided as much as possible for the following reasons:

1. The event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.
2. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

REGISTERING EVENT LISTENERS

```
const btn = document.querySelector("button");

btn.addEventListener("click", () => {document.
getElementById(“p”).innerHTML = “hello”; });
```

- An `addEventListener()` function is used to make the HTML `button` element fires an event when the user clicks the button.
- The function passes two parameters:

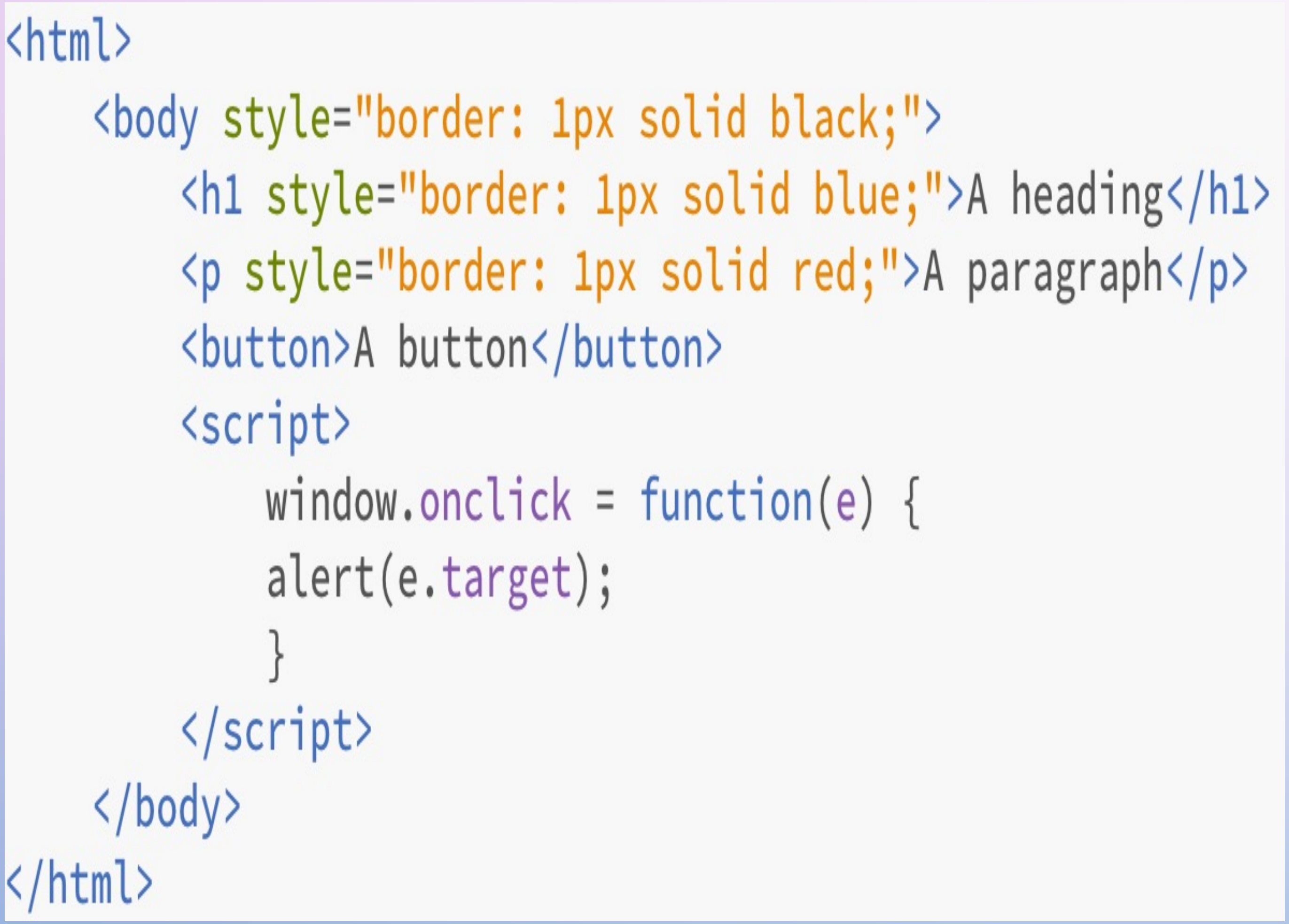
1. the string `"click"`, to indicate that we want to listen to the click event.
2. a function to call when the event happens.

- `addEventListener()` method can be used to add more than one handler for a single event.

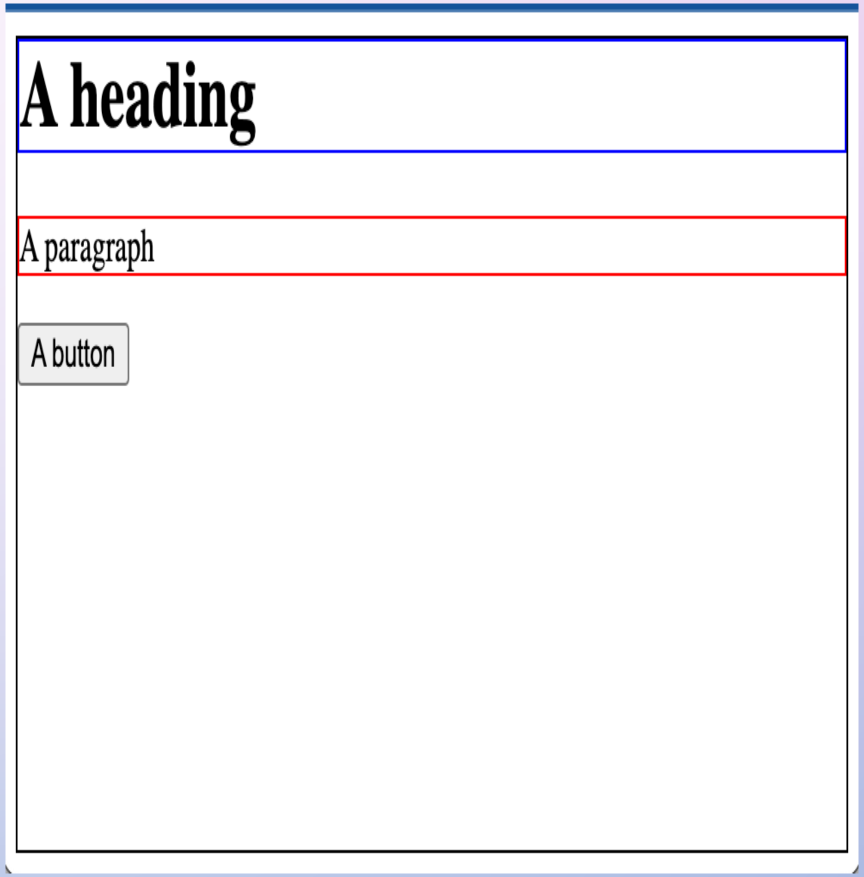
EVENT OBJECTS

- Event handler functions are passed an argument called an *event object*, which holds information about the event that was fired.
- Event objects store information about the event target, the event type, and associated listeners in properties and methods.
- Conventional names are e, event, or even evt, following the fact that the parameter represents an event.

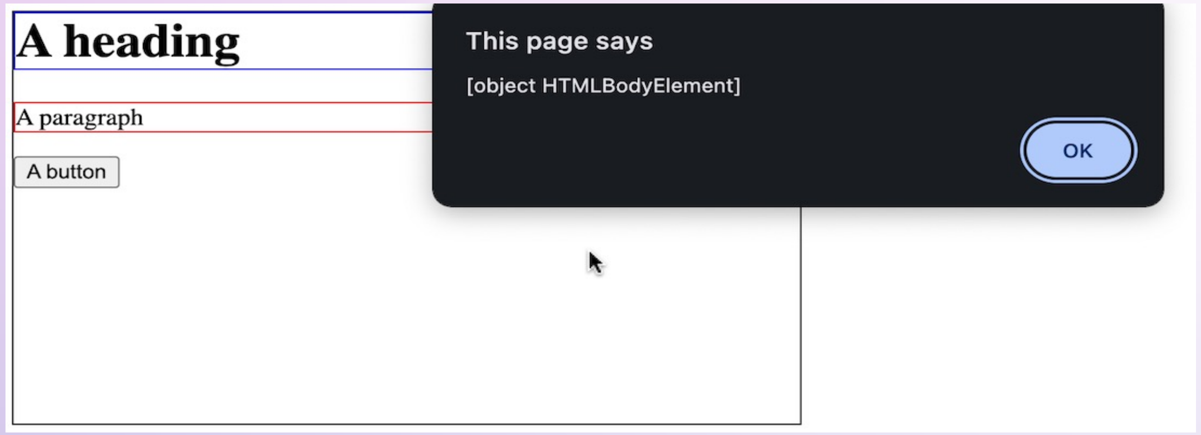
EXAMPLE



```
<html>
    <body style="border: 1px solid black;">
        <h1 style="border: 1px solid blue;">A heading</h1>
        <p style="border: 1px solid red;">A paragraph</p>
        <button>A button</button>
        <script>
            window.onclick = function(e) {
            alert(e.target);
            }
        </script>
    </body>
</html>
```



EXAMPLE (CONT.)



A heading

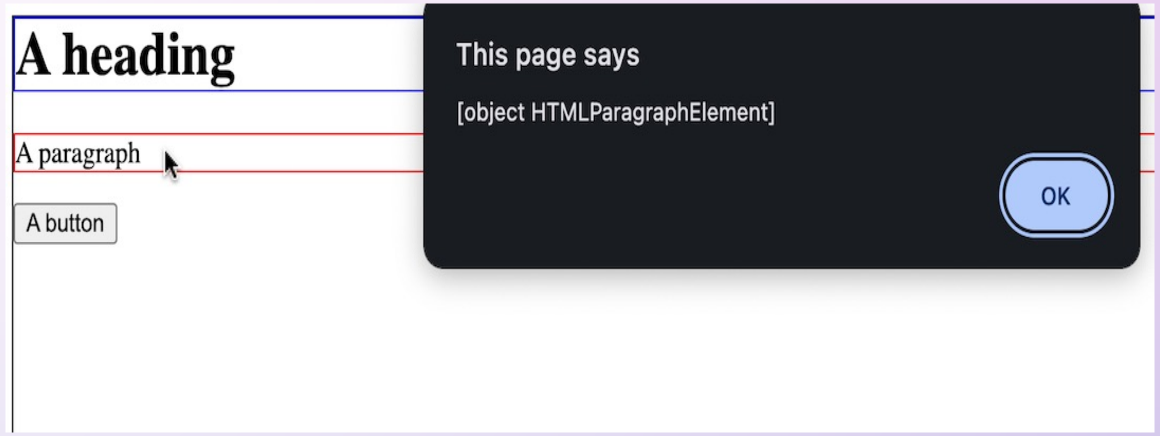
A paragraph

A button

This page says

[object HTMLBodyElement]

OK



A heading

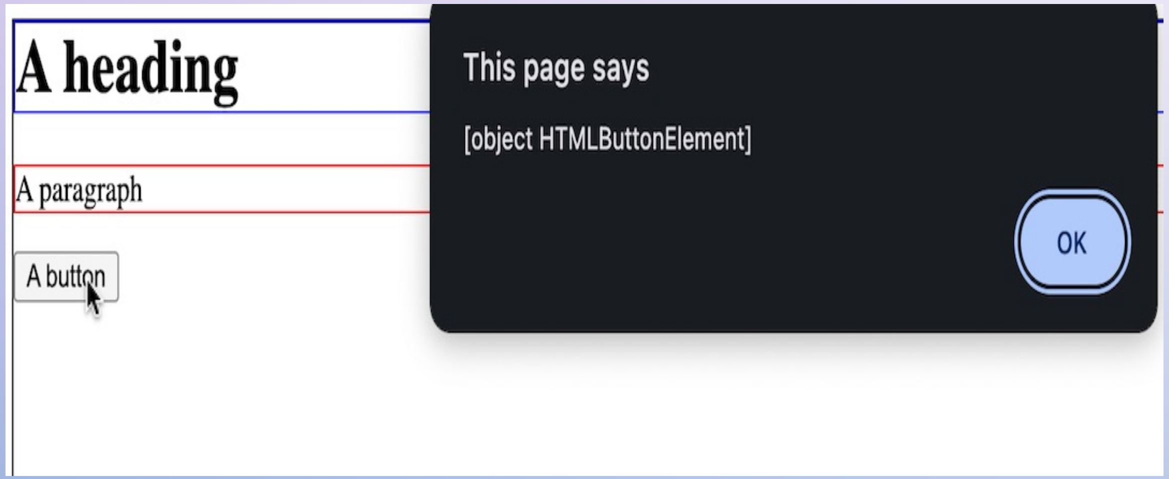
A paragraph

A button

This page says

[object HTMLParagraphElement]

OK



A heading

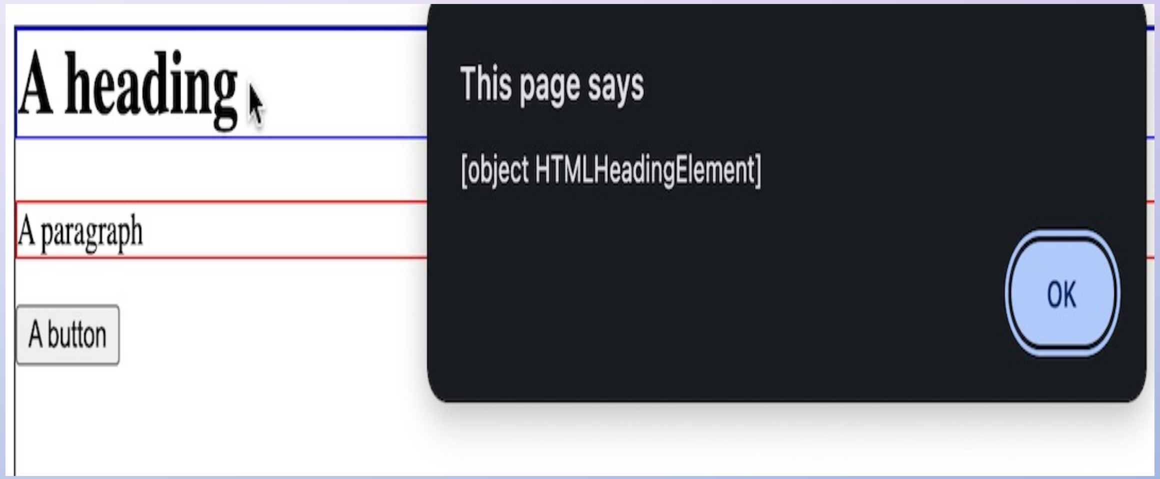
A paragraph

A button

This page says

[object HTMLButtonElement]

OK



A heading

A paragraph

A button

This page says

[object HTMLHeadingElement]

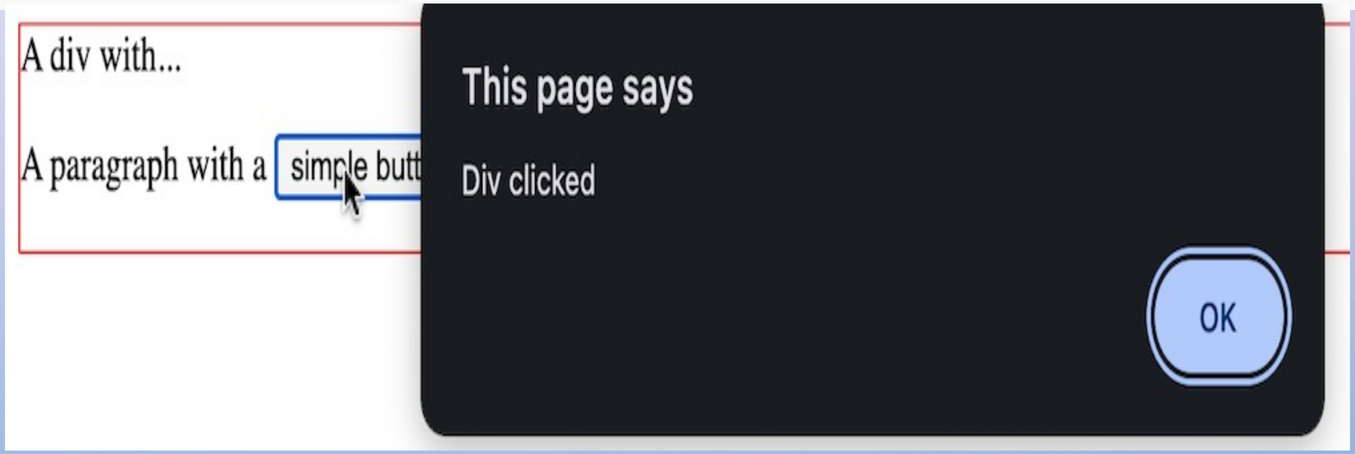
OK

EVENT PROPAGATION

- Event propagation is the process of transmitting an event across a series of objects.
- Propagation can be done in either of two ways: bubbling and capturing.
- An event can be in either of the following 4 states, also known as its phases:
    1. The 'none' phase is when event hasn't yet been fired on any object.
    2. The 'capturing' phase is when the event is being propagated in the capturing stage.
    3. The 'bubbling' phase is when the event is being propagated in the bubbling stage.
    4. The 'target' phase is when the event is fired on its target.

EXAMPLE

```html
<body>
    <div style="border: 1px solid red;">
        A div with...
        <p>A paragraph with a <button>simple button</button></p>
    </div>
    <script>
        var divElement = document.querySelector('div');
        divElement.onclick = function() {
        alert('Div clicked');
        }
    </script>
</body>
```

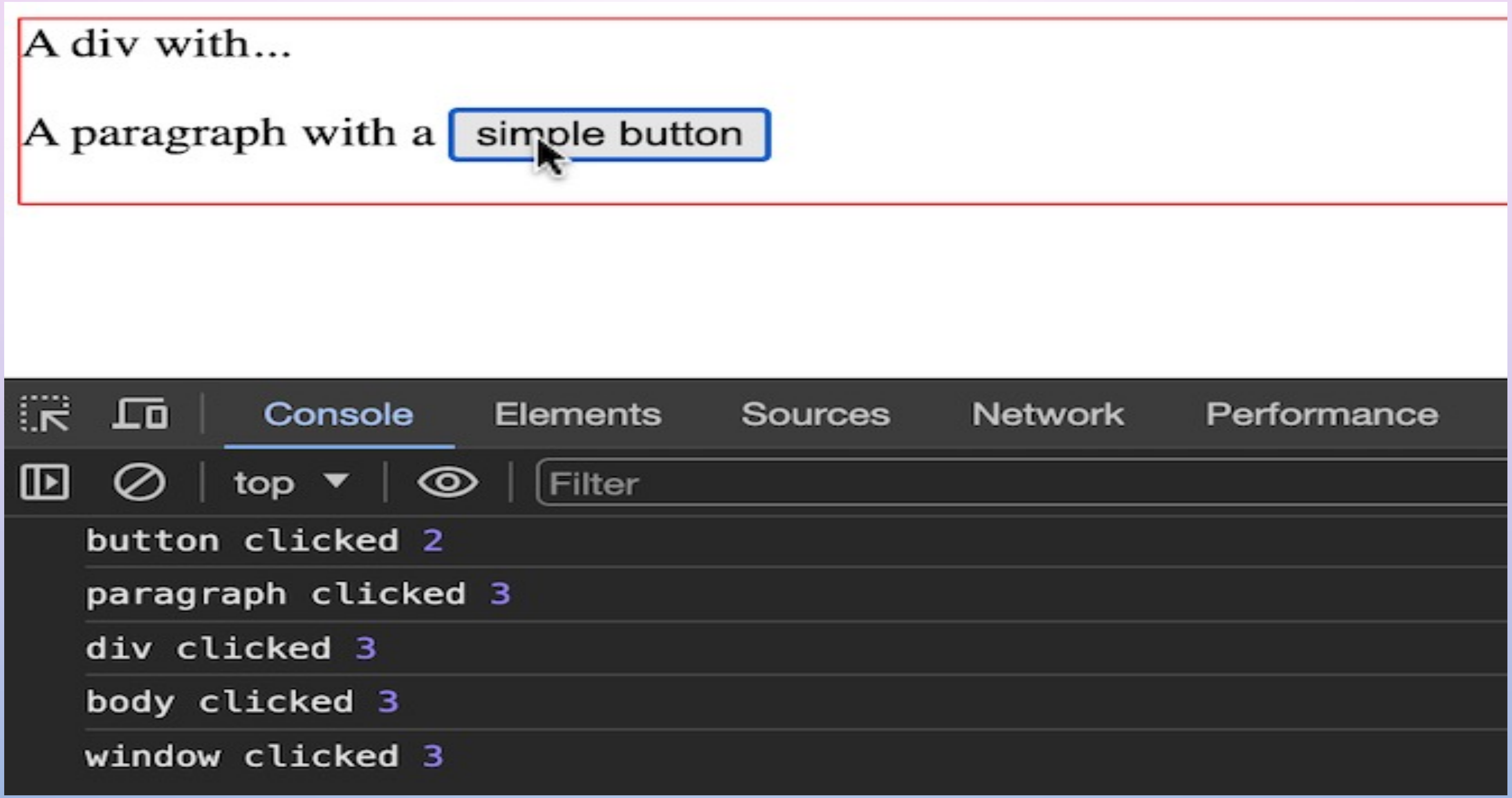

A div with...

A paragraph with a simple butt

This page says

Div clicked

BUBBLING PROPAGATION


A div with...

A paragraph with a simple button

```
button clicked 2
paragraph clicked 3
div clicked 3
body clicked 3
window clicked 3
```