

DOM Methods:

- Table of DOM Methods:

Method	What it does
x.innerHTML	Gets the content of x
x.attributes	Gets the attribute nodes of x
x.style	Gets the CSS of x
x.parentNode	Gets the parent node of x
x.children	Gets the child/children nodes of x
x.appendChild	Inserts a child node to x
x.removeChild	Removes a child node from x

- **Note:** When you use these DOM methods, you are manipulating/changing the DOM. For example, when you do x.style, you are manipulating the CSS of the DOM element x, but you are not making any changes to the css file itself.

Events:

- All **events** occur on HTML elements in the browser.
- DOM events are sent to notify code of interesting things that have taken place. Each event is represented by an object which is based on the Event interface, and may have additional custom fields and/or functions used to get additional information about what happened. Events can represent everything from basic user interactions to automated notifications of things happening in the rendering model.
- JavaScript is used to define the action that needs to be taken when an event occurs. I.e. When [HTML Event] , do [JS Action]
- To show that actions originate from HTML elements, we can put attributes inside of elements that 'listen' for events.
- E.g. `<div onclick="alert('Clicked!')">...</div>`
In this case, a user clicking on this <div> will run some JS to show an alert dialog.
- E.g. Consider the code and output below

```

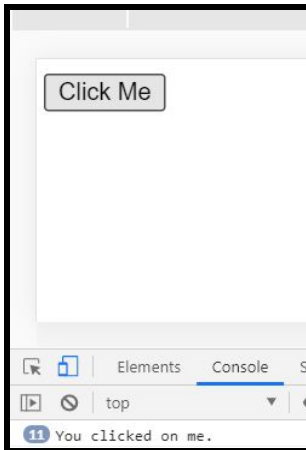
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="test.css">
  </head>
  <body>

    <!-- When the button is clicked, a message will be printed in the console log.-->
    <button onclick="test()"> Click Me </button>

    <script type="text/javascript" src="test.js"> </script>
  </body>
</html>

```

```
function test(){  
    console.log("You clicked on me.")  
}
```

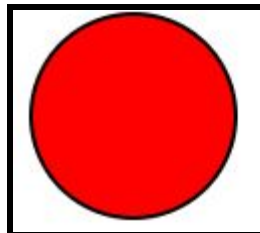
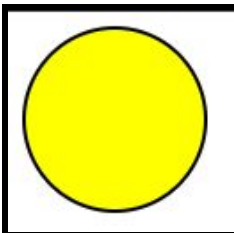


- E.g. Consider the code and output below

```
<!DOCTYPE html>  
<html>  
  <head>  
    <link rel="stylesheet" type="text/css" href="test.css">  
  </head>  
  <body>  
  
    <!-- When the user's mouse is hovering over the yellow circle, it changes color to red.-->  
    <div id="circle" onmouseover="changeColor(this)"></div>  
  
    <script type="text/javascript" src="test.js"> </script>  
  </body>  
</html>
```

```
div{  
  border: 2px solid black;  
  border-radius: 50%;  
  width: 100px;  
  height: 100px;  
  background-color: yellow;  
}
```

```
function changeColor(shape){  
  shape.style.backgroundColor = "red";  
}
```



- An **event listener** in JS programmatically sets an event attribute on an HTML element. Syntax: **element.addEventListener(event, functionToExecuteWhenEventOccurs)**
Note: **functionToExecuteWhenEventOccurs** is called a **callback function**.
As a result, you can rewrite the above as **element.addEventListener(event, callback)**
- A **callback** is a function that is designated to be 'called back' at an appropriate time. In the case of events, it will be 'called back' when the event occurs. It can be an anonymous function, or a function defined outside of the event listener.
- E.g.
button.addEventListener('click', function() { alert('Clicked') }); OR
function alertClick() { alert('Clicked') }
button.addEventListener('click', alertClick);
- Note that in the second way, we wrote alertClick and not alertClick(). You do not put parentheses if you are using addEventListener in the second way.
- E.g. Consider the code and output below

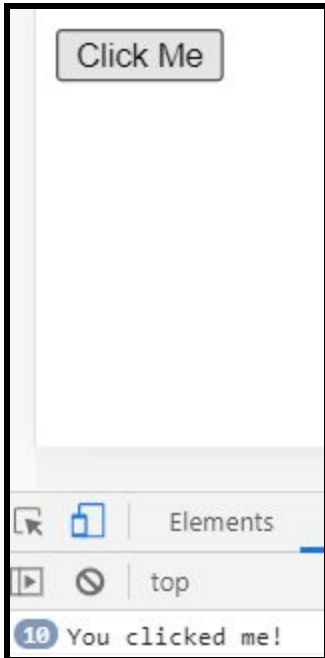
```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="test.css">
  </head>
  <body>

    <!-- When the button is clicked, a message shows up in the console log. -->
    <button id="button"> Click Me </button>
    <!-- When the user's mouse is hovering over the yellow circle, it changes color to red.
    <div id="circle" onmouseover="changeColor(this)"></div> -->

    <script type="text/javascript" src="test.js"> </script>
  </body>
</html>
```

```
document.getElementById("button").addEventListener("click", printToConsoleLog)

function printToConsoleLog(){
    console.log("You clicked me!")
}
```



- E.g. Consider the code and output below

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="test.css">
  </head>
  <body>

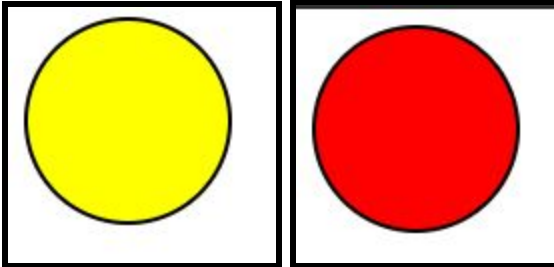
    <!-- When the user's mouse is hovering over the yellow circle, it changes color to red. -->
    <div id="circle"></div>

    <script type="text/javascript" src="test.js"> </script>
  </body>
</html>
```

```
div{
  border: 2px solid black;
  border-radius: 50%;
  width: 100px;
  height: 100px;
  background-color: yellow;
}
```

```
document.getElementById("circle").addEventListener("mouseover", changeColor)

function changeColor(){
  const shape = document.getElementById("circle")
  shape.style.backgroundColor = "red";
}
```



- **Note:** EventListener and HTML attributes have different ways of writing about the same event. For example, in HTML attributes, onclick is used but in EventListener, click is used. They both mean the same thing.
- All events that occur create a JS Object with information about that event.
Event.target gives information about the event origin element.
Event.type gives information about the type of event.
- Events can be passed to the callback function as an argument.
 E.g.

```
function myCallback(e) {
  // find information about e.
  // execute proper code.
}
```

- E.g. Consider the code and output below

```
document.getElementById("circle").addEventListener("click", changeColor)

function changeColor(e){
  const shape = document.getElementById("circle")
  shape.style.backgroundColor = "red";
  console.log(e)
  console.log(e.target)
  console.log(e.type)
}
```

```
► MouseEvent {isTrusted: true, screenX: 113, screenY: 258, clientX: 73, clientY: 65, ...}
  <div id="circle" style="background-color: red;"></div>
  click
```

- Some common events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

- Some event modifiers:
 1. **Event.preventDefault():**
 - Prevents default action of an event.
E.g. click on form submit, clicking on link
 2. **Event.stopPropagation():**
 - Prevents child elements from propagating events to parent elements in some cases.
I.e. Prevents propagation of the same event from being called.
Propagation means bubbling up to parent elements or capturing down to child elements.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="test.css">
  </head>
  <body>

    <a id="link" href="https://www.google.com/"> Google </a>

    <script type="text/javascript" src="test.js"> </script>
  </body>
</html>
```

```
document.getElementById("link").addEventListener("click", function(e){
  e.preventDefault()
})
```




No matter how many times I click on the link, nothing happens. This is because `preventDefault()` is preventing the link from taking me to Google.com.

- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="test.css">
  </head>
  <body>

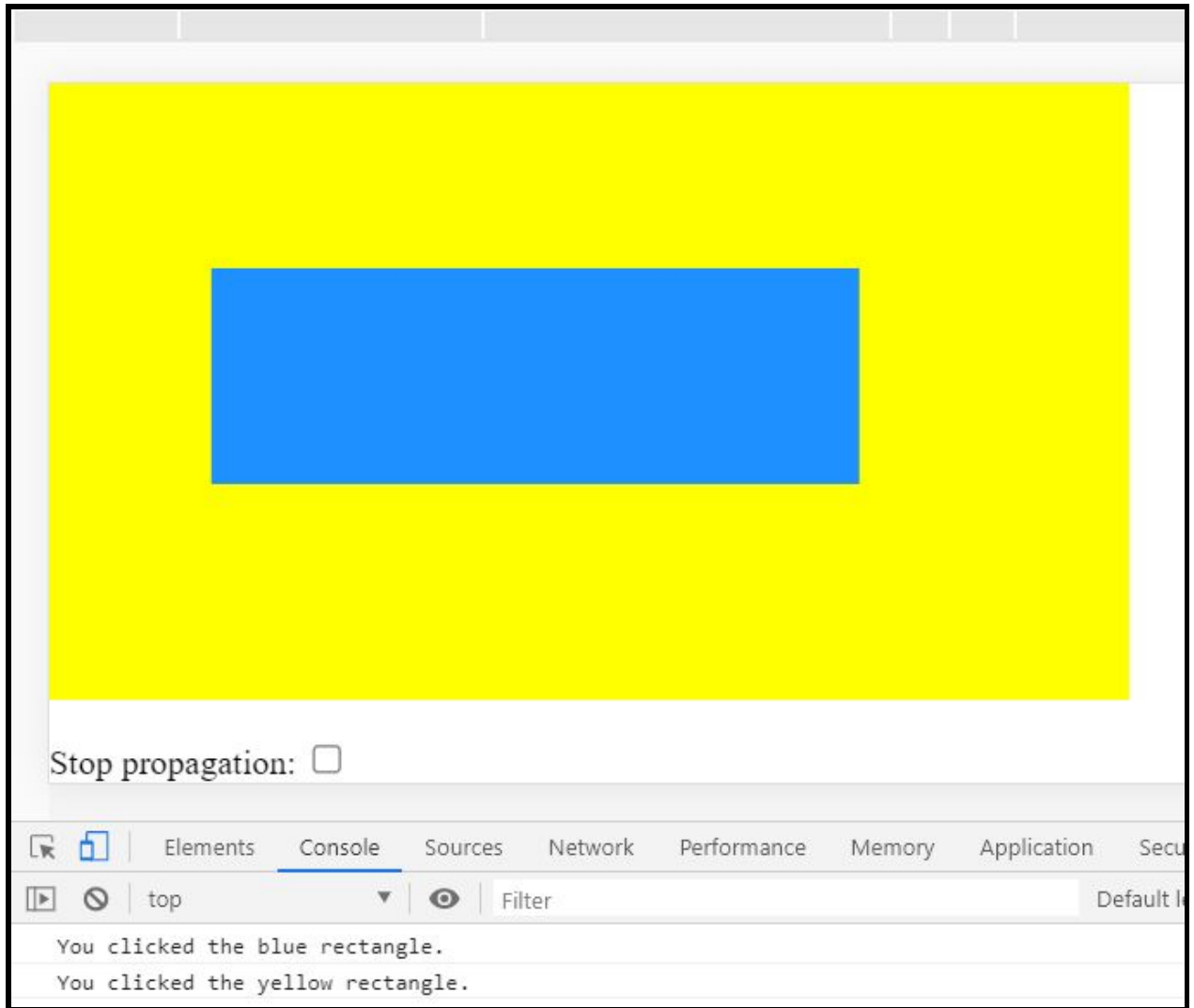
    <div id="rectangle_1">
      <div id="rectangle_2"></div>
    </div>

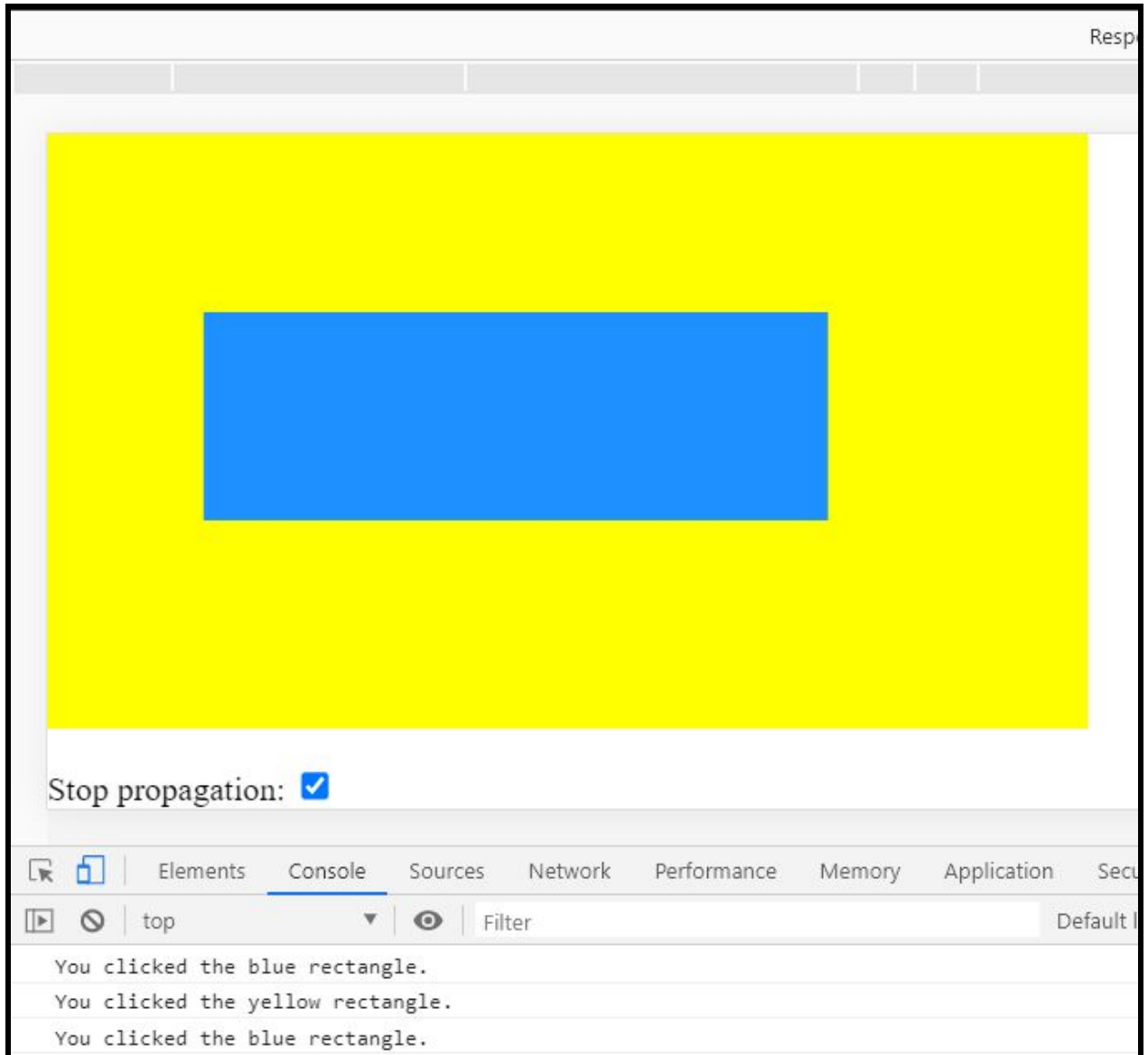
    <br>
    Stop propagation:
    <input type="checkbox" id="check">

    <script type="text/javascript" src="test.js"> </script>
  </body>
</html>
```

```
document.getElementById("rectangle_2").addEventListener("click", function(e){
  console.log("You clicked the blue rectangle.")
  if (document.getElementById("check").checked) {
    e.stopPropagation();
  }
})

document.getElementById("rectangle_1").addEventListener("click", function(){
  console.log("You clicked the yellow rectangle.")
})
```





When I click the “Stop propagation” checkbox, when I click on the blue rectangle, the second function doesn’t run.

Non-blocking JS:

- **Block code** is code that runs one instruction after another, and makes next instructions wait.
- **Non-blocking code** allows JS to continue executing instructions while we wait for some blocking code to complete. This feature of JS that allows for non-blocking code is an example of **asynchronous programming**.
- JavaScript must be compiled and interpreted. It needs a runtime environment. E.g. In Chrome, the JS runtime is the V8 Engine. Furthermore, Javascript is an event-driven language. It uses something called the **event loop** to keep track of all of these events. The event loop is a way of scheduling events one after the other. It often gets help from the platform the engine is running on (the browser). It is important to note that JS is single-threaded, but that browsers are multi-threaded. Non-blocking JS functions aren’t built into the V8 runtime. They live in the platform JS is running on, the

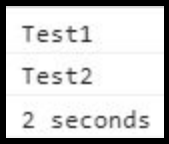
browser. For example, Chrome contains the instructions for `setTimeout`, a non-blocking function in JS.

- E.g. Consider the code and output below:

```
console.log("Test1")

setTimeout(function(){
  console.log('2 seconds')
},2000)

console.log("Test2")
```



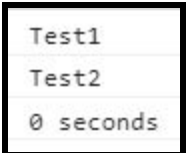
Test1
Test2
2 seconds

- E.g. Consider the code and output below:

```
console.log("Test1")

setTimeout(function(){
  console.log('0 seconds')
},0)

console.log("Test2")
```



Test1
Test2
0 seconds

- Here is a link with a demonstration for the second example:

https://youtu.be/X_uQ80bFZk

- What's happening is this:

- Recall that JS is single-threaded. So, the call stack can only run one instruction at a time.
- Furthermore, whenever there's asynchronous code, it goes to the web api, which is in the browser. In the video above, `setTimeout()` goes to the web api. Afterwards, the asynchronous code goes to the callback queue, where it waits until the stack is empty. Hence, all asynchronous code must wait for all the synchronous code to be executed before they go to the call stack. This is done to give more consistency and to make things more predictable. This is why, even though we set `setTimeout()` to 0 seconds, it still runs after the 2 `console.log()` lines.