Event Propagation

Event propagation in JavaScript defines the order in which events are handled in the DOM. It consists of three phases:

1. **Capturing Phase (Event Capturing)**:
   - The event starts from the root of the DOM tree and propagates down to the target element.
   - Handlers set for the capturing phase are executed during this phase.
2. **Target Phase**:
   - The event reaches the target element.
   - Handlers on the target element are executed.
3. **Bubbling Phase (Event Bubbling)**:
   - The event propagates back up from the target element to the root of the DOM tree.
   - Handlers set for the bubbling phase are executed during this phase.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        #outer{
            background-color: blue;
            width: 600px;
            height: 600px;
        }
        #middle{
            background-color: wheat;
            width: 400px;
            height: 400px;
            margin: 100px;
        }
        #inner{
            background-color: red;
            width: 200px;
            position: absolute;
            display: flex;
            justify-content: center;
            align-items: center;
            margin: 100px;
            height: 200px;
        }
    </style>
</head>
```

```
<body>
    <div id="outer">
        <div id="middle">
            <div id="inner">Click Me</div>
        </div>
    </div>
    <script>
        var innerTag = document.getElementById('inner');
        innerTag.addEventListener('mouseover',function(event){
            console.log("Inner tag is triggered");
            console.log(event)
            event.stopPropagation()
        },)
        var middleTag = document.getElementById('middle');
        middleTag.addEventListener('click', function(event){
            console.log('Middel tag is triggered')
            event.stopPropagation()

        },)
        var outerTag = document.getElementById('outer');
        outerTag.addEventListener('click', function(){
            console.log('Outer tag is triggered')
        }, )
    </script>
</body>
</html>
```

Event Delegation

Event delegation is a technique to handle events efficiently by leveraging event bubbling. Instead of adding event listeners to individual child elements, you add a single event listener to a common ancestor and handle events for specific child elements using the event object.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div>
        <ul id="list">
            <li>Item 1</li>
            <li>Item 2</li>
            <li>Item 3</li>
        </ul>
    </div>
    <button id="button-tag">Add Item</button>
    <script>
        var ulTag = document.getElementById('list');
        ulTag.addEventListener('click',function(event){
            console.log(event.target)
            console.log("Li tag is triggered")
        })
        var buttonTag = document.getElementById('button-tag');
        buttonTag.addEventListener('click',function(){
            var newElement = document.createElement('li');
             newElement.textContent = "Item " + (document.querySelectorAll('#list li').length +
1);
            ulTag.appendChild(newElement);
        })
    </script>
</body>
</html>
```

Introduction to Error Handling

Error handling in JavaScript is crucial for writing robust and reliable code. JavaScript provides mechanisms to handle runtime errors and create custom error messages to ensure your applications run smoothly even when unexpected issues arise.

Types of Errors

1. **Syntax Errors**:
   - Occur when there is a mistake in the syntax of the code.
   - Example: console.log("Hello World"
2. **Runtime Errors** (Exceptions):
   - Occur during the execution of the code.
   - Example: let x = y + 1; // ReferenceError: y is not defined
3. **Logical Errors**:
   - Occur when the code runs without throwing any error but produces an incorrect result.
   - Example: Incorrect calculation or using wrong conditions in a loop.

Error Handling Mechanisms

1. **try...catch Statement**:
   - try: The block of code to be tested for errors.
   - catch: The block of code that handles the error.
   - finally: Optional block executed after try and catch, regardless of the outcome

```
var a = 10;
var b = 3;
try{

   console.log(a+b);
}catch(error){
   console.log(error.message);
}finally{
   console.log("Finally");
}

var x = 6;
console.log(x)
```

throw Statement:

- Used to create custom errors.
- Can throw any expression, but commonly used with Error objects

```

function divide(a, b) {
   if (b === 0) {
     throw new Error("Division by zero is not allowed.");
   }
   return a / b;
 }

 try {
   let result = divide(4, 0);
 } catch (error) {
   console.error(error.message);
 }
```

Interview Questions and Answers

What is the purpose of the try...catch statement in JavaScript?

Answer: The try...catch statement is used to handle exceptions (runtime errors) in JavaScript. The code that may throw an error is placed inside the try block, and the error handling code is placed inside the catch block. An optional finally block can be used for cleanup tasks.

How do you create a custom error in JavaScript?

Answer: Custom errors can be created by extending the built-in Error class. This allows you to create more specific error types.

What is the purpose of the finally block in a try...catch statement?

Answer: The finally block is executed after the try and catch blocks, regardless of whether an error was thrown or not. It is typically used for cleanup activities such as closing files, releasing resources, or resetting states.

Explain the difference between throw and return statements in error handling.

Answer: The throw statement is used to create and raise an exception, which can be caught and handled by a try...catch block. The return statement, on the other hand, is used to exit a function and optionally return a value to the function caller. throw interrupts the normal control flow to handle errors, while return is part of the normal function execution.

How can you handle multiple types of errors in a single catch block?

Answer: You can handle multiple types of errors in a single catch block by checking the error's type or properties inside the catch block.

```
try {
  // Code that may throw different types of errors
 } catch (error) {
   if (error instanceof TypeError) {
     console.error("Type Error: ", error.message);
   } else if (error instanceof ReferenceError) {
     console.error("Reference Error: ", error.message);
   } else {
     console.error("General Error: ", error.message);
   }
 }
```

Interview Questions:

What is event propagation in JavaScript?

Answer: Event propagation is the process by which an event travels through the DOM tree. It includes three phases: capturing, target, and bubbling.

Explain the capturing phase in event propagation.

Answer: The capturing phase is the phase in which the event starts from the root and travels down to the target element. Handlers for the capturing phase are executed during this phase.

What is the target phase in event propagation?

Answer: The target phase is when the event reaches the target element, and the event handler on the target element is executed.

Describe the bubbling phase in event propagation.

Answer: The bubbling phase is the phase in which the event travels back up from the target element to the root. Handlers for the bubbling phase are executed during this phase.

What is event delegation and why is it useful?

Answer: Event delegation is a technique where a single event listener is added to a common ancestor of multiple child elements. It leverages event bubbling to handle events for specific child elements. It is useful for reducing the number of event listeners and for handling dynamically added elements.

How can you stop event propagation?

Answer: You can stop event propagation by using the stopPropagation() method on the event object. This will prevent the event from bubbling up (or capturing down) the DOM tree.

```
document
     .getElementById("inner")
     .addEventListener("click", function (event) {
       event.stopPropagation();
       alert("Inner Div Clicked!");
     });
```

What is the difference between preventDefault() and stopPropagation()?

Answer: preventDefault() prevents the default action associated with the event (e.g., following a link or submitting a form), while stopPropagation() stops the event from propagating through the DOM (bubbling or capturing).

How would you implement event delegation to handle clicks on dynamically added list items?

Answer: You add an event listener to a common ancestor (like a ul element) and use the event object to determine which child element was clicked

Can you explain the difference between true and false in the third argument of addEventListener?

Answer: The third argument of addEventListener specifies whether the event handler should be executed in the capturing phase (true) or the bubbling phase (false). By default, it is false, meaning the event handler is executed in the bubbling phase.