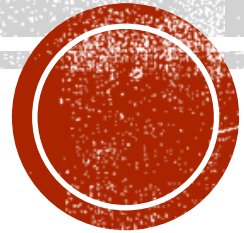
ES6+



Lecturer: Suraj Pandey, CCT College

- JavaScript was developed by Brendan Eich, a developer at Netscape Communications Corporation, in 1995.JavaScript started life with the name Mocha, and was briefly named LiveScript before being officially renamed to JavaScript. It is a scripting language that is executed by the browser, i.e. on the client's end. It is used in conjunction with HTML to develop responsive webpages

ECMASCRIPT VERSIONS

- There are nine editions of ECMA-262 which are as follows:

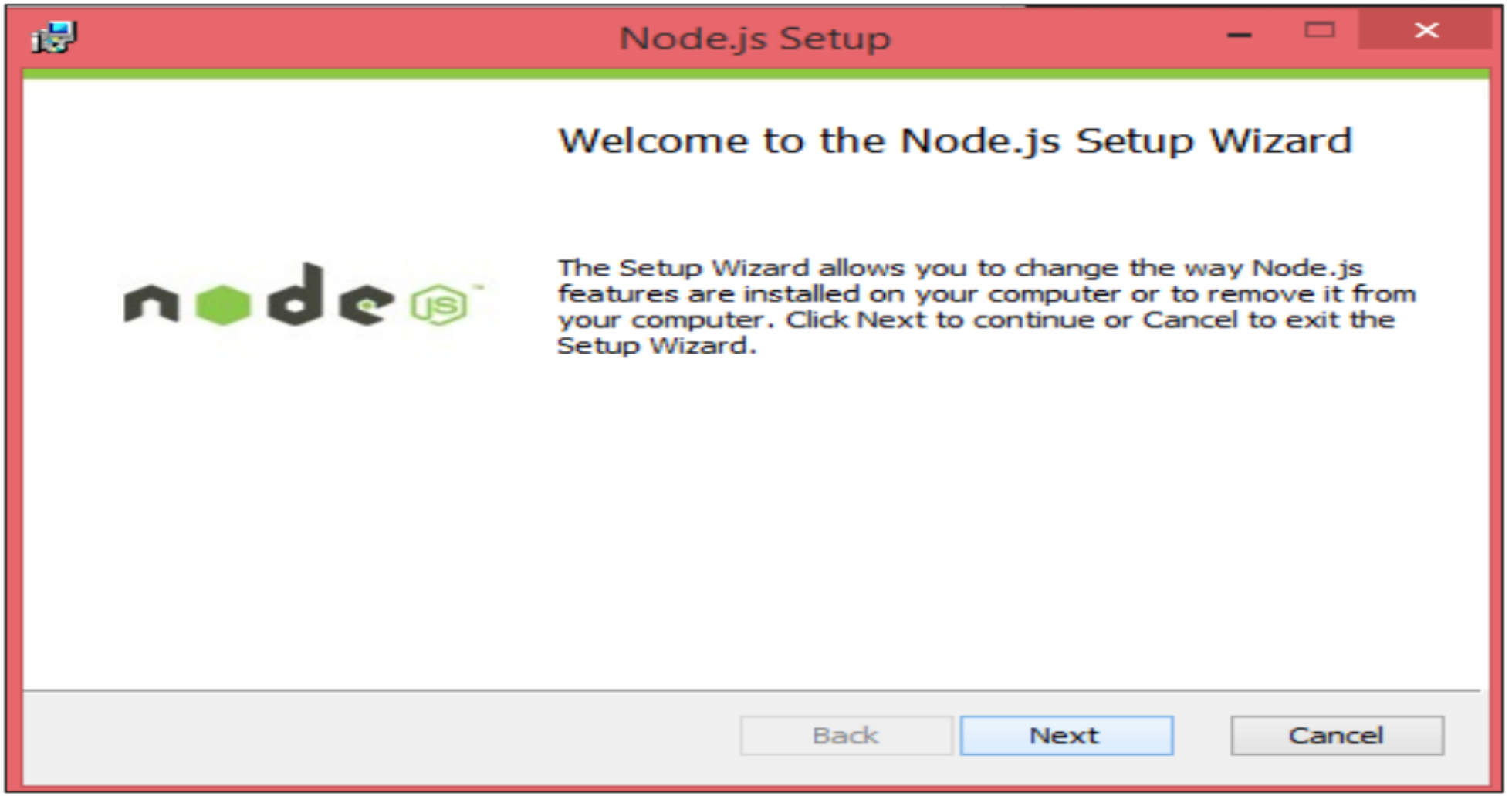
Edition	Name	Description
1	ECMAScript 1	First Edition released in 1997
2	ECMAScript 2	Second Edition released in 1998, minor changes to meet ISO/IEC 16262 standard
3	ECMAScript 3	Third Edition released in 1999 with language enhancements
4	ECMAScript 4	Fourth Edition release plan was dropped, few features added later in ES6 & other complex features dropped
5	ECMAScript 5	Fifth Edition released in 2009
5.1	ECMAScript 5.1	5.1 Edition released in 2011, minor changes to meet ISO/IEC 16262:2011 standard
6	ECMAScript 2015/ES6	Sixth Edition released in 2015, see ES6 chapters for new features
7	ECMAScript 2016/ES7	Seventh Edition released in 2016, see ES7 chapters for new features
8	ECMAScript 2017/ES8	Eight Edition released in 2017, see ES8 chapters for new features
9	ECMAScript 2018/ES9	Ninth Edition released in 2018, see ES9 chapters for new features

SETTING UP OF THE ENVIRONMENT FOR ES6.

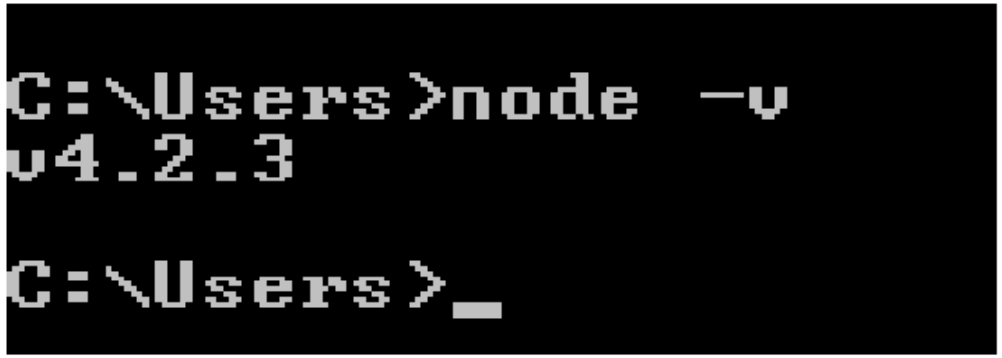
- LocalEnvironment Setup JavaScript can run on any browser, any host, and any OS. You will need the following to write and test a JavaScript program standard:
- Text Editor
- The text editor helps you to write your source code. Examples of few editors include Windows Notepad, Notepad++, Emacs, vim or vi etc. Editors used may vary with the operating systems. The source files are typically named with the extension .js.
- Installing Node.js
- Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute the code. You may download Node.js source code or a pre-built installer for your platform. Node is available at https://nodejs.org/en/download

Installation on Windows

Download and run the **.msi installer** for Node.



To verify if the installation was successful, enter the command ***node –v*** in the terminal window.



ES6 – SYNTAX

- Syntax defines the set of rules for writing programs. Every language specification defines its own syntax. A JavaScript program can be composed of:
- Variables: Represents a named memory block that can store values for the program.
- Literals: Represents constant/fixed values.
- Operators: Symbols that define how the operands will be processed.
- Keywords: Words that have a special meaning in the context of a language.
- The following table lists some keywords in JavaScript. Some commonly used keywords are listed in the following table.

break	as	any	Switch
case	if	throw	Else
var	number	string	Get
module	type	instanceof	Typeof
finally	for	enum	Export
while	void	this	New
null	super	Catch	let
static	return	True	False

- Modules: Represents code blocks that can be reused across different programs/scripts.
- Comments: Used to improve code readability. These are ignored by the JavaScript engine.
- Identifiers: These are the names given to elements in a program like variables, functions, etc. The rules for identifiers are:

1. Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
2. Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
3. Identifiers cannot be keywords. They must be unique.
4. Identifiers are case sensitive.
5. Identifiers cannot contain spaces.

The following table illustrates some valid and invalid identifiers.

Examples of valid identifiers	Examples of invalid identifiers
firstName first_name num1 $result	Var# first name first-name 1number

- **Whitespaceand Line Breaks**
- ES6 ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.
- **JavaScript is Case-sensitive**
- JavaScript is case-sensitive. This means that JavaScript differentiates between the uppercase and the lowercase characters.
- **Semicolons are Optional**
- Each line of instruction is called a statement. Semicolons are optional in JavaScript.

- Example
- console.log("hello world")
- console.log("We are learning ES6")
  - A single line can contain multiple statements. However, these statements must be separated by a semicolon.

COMMENTS IN JAVASCRIPT

- Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like the author of the code, hints about a function/construct, etc. Comments are ignored by the compiler. JavaScript supports the following types of comments:
- **Single-line comments** (//): Any text between a // and the end of a line is treated as a comment.
- **Multi-line comments** (/* */): These comments may span multiple lines

Example

```
//this is single line comment


/* This is a
Multi-line comment
*/
```


Your First JavaScript Code

Let us start with the traditional "Hello World" example".

```
var message="Hello World"

console.log(message)
```

The program can be analyzed as:

- Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.

- Line 2 prints the variable's value to the prompt. Here, the console refers to the terminal window. The function log () is used to display the text on the screen.

Executing the Code

We shall use Node.js to execute our code.

Step 1: Save the file as Test.js

Step 2: Right-click the Test.js file under the working files option in the project-explorer window of the Visual Studio Code.

Step 3: Select Open in Command Prompt option.

Step 4: Type the following command in Node's terminal window.

```
node Test.js
```

The following output is displayed on successful execution of the file.

```
Hello World
```


ES6+ FEATURES

1. let and const

Before ES6, JavaScript had only one way to declare variables: var. However, var comes with some quirks, such as function-scoping and the potential for unexpected behavior due to hoisting. ES6 introduced let and const, which provide block-scoping and more predictable behavior.

- **let**: Used to declare variables that can be reassigned. It's block-scoped, meaning it's only accessible within the block where it's declared.

```
let count = 10;
if (count > 5) {
  let message = "Count is greater than 5";
  console.log(message); // Works fine
}
console.log(message); // Error: message is not defined
```


- **const**: Used to declare constants. These cannot be reassigned after their initial assignment, and like let, const is block-scoped.

```
const pi = 3.14;
pi = 3.14159; // Error: Assignment to constant variable
```


2. Arrow Functions

Arrow functions provide a shorter syntax for writing functions and also offer more predictable behavior with the `this` keyword. They are especially useful for writing concise, anonymous functions.

```
// Traditional function expression
const add = function(a, b) {
  return a + b;
};
```

```
// Arrow function
const add = (a, b) => a + b;

console.log(add(2, 3)); // Output: 5
```


PROMISES

- Promises in JavaScript are a way to handle asynchronous operations. They allow you to write code that can run in the future once a task completes, without blocking the execution of other code. Here's a simple example to illustrate how promises work:

Promises provide a cleaner way to handle asynchronous operations, avoiding callback hell. A promise represents a value that may be available now, or in the future, or never.

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data fetched");
  }, 2000);
});
```

```
fetchData.then(data => console.log(data)); // Output: Data fetched
```


OBJECTS

- Objects in ECMAScript 6 (ES6) and beyond (often referred to as ES6+) have introduced several new features and enhancements that make working with objects more intuitive and powerful. Here are some of the key features and improvements:

- **Object Literals**: Simplified syntax for creating objects.
- **Computed Property Names**: Dynamically defining property names using expressions.
- **Method Definitions**: Defining methods directly in object literals.
- **Destructuring**: Extracting properties from objects into variables.

- **Shorthand Property Names:** This feature allows you to use shorthand syntax when the property name is the same as the variable name.

```
const name = "Alice";

const age = 30;

const person = { name, age }; // Equivalent to { name: name, age: age }
```


Computed Property Names: You can use expressions as property names, making your objects more dynamic.

javascript

```
const propName = "age";
const person = {
        name: "Alice",
 [propName]: 30 // Equivalent to { age: 30 } };
```


REGULAR EXPRESSION

Regular expressions (regex or regexp) are patterns used to match character combinations in strings.
In ECMAScript 6 (ES6) and later versions, regular expressions have some new features and improvements.
Here's a detailed overview:

- **Basic Concepts**
- **Pattern Matching**: Regular expressions are used to search, match, and manipulate strings based on specific patterns.
- **Syntax**: Patterns are written using a combination of literal characters and metacharacters

NEW FEATURES IN ES6+

- **Sticky Flag** (y): This flag makes the regex match only from the current position in the string. It's useful for matching the next occurrence of a pattern.
- **Unicode Flag (**u): This flag enables Unicode support in regular expressions, allowing you to match characters from various languages and scripts

EXAMPLE

```
// Using the sticky flag
const regex = /hello/y;
const str = 'hello world';
console.log(regex.test(str)); // true

// Using the Unicode flag
const regex2 = /\u{1F601}/u; // Unicode smiley face
const str2 = '😁';
console.log(regex2.test(str2)); // true
```


Methods and Properties

1. test(): Checks if a pattern matches a string.
2. match(): Searches a string for a pattern and returns an array of matches.
3. replace(): Replaces matched substrings with a replacement string.
4. search(): Searches a string for a pattern and returns the index of the first match.
5. split(): Splits a string into an array based on a pattern

EXAMPLE METHODS

```
const str = 'JavaScript is fun!';
const regex = /fun/;
console.log(regex.test(str)); // true
console.log(str.match(regex)); // ['fun']
console.log(str.replace(regex, 'awesome')); // 'JavaScript is awesome!'
console.log(str.search(regex)); // 16
console.log(str.split(regex)); // ['JavaScript is ', '!']
```


PRACTICAL EXAMPLES OF ES6 REGULAR EXPRESSIONS
VALIDATING A DATE FORMAT (YYYY-MM-DD)

```
let regex = /^(?<year>\d{4})-(?<month>\d{2})-(?
<day>\d{2})$/;
let s = "2024-12-31";
let match = s.match(regex);

if (match) {
    console.log(match.groups.year);
    console.log(match.groups.month);
    console.log(match.groups.day);
}
```

Output

```
2024
12
31
```


MATCHING A VALID EMAIL ADDRESS

```
let regex = /^(?<username>[a-zA-Z0-9._%+-]+)@(?<domain>
[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})$/;
let mail = "user@example.com";
let match = mail.match(regex);

if (match) {
    console.log(match.groups.username);
    console.log(match.groups.domain);
}
```

Output

```
user
example.com
```


EXTRACTING EMOJI FROM A STRING

```
let regex = /\p{Emoji}/gu;
let s = "Hello , how are you? ";

console.log(s.match(regex));
```

Output

```
[ '😊', '😎' ]
```

The regex uses the Unicode property escape \p{Emoji} to match emoji characters in the string.

UNDERSTANDING ASYNCHRONOUS PROGRAMMING

In JavaScript, tasks are processed one after another by default. However, some tasks, like reading files, making network requests, or processing large amounts of data, can take a long time to complete. If these tasks block the main thread, it can make the application unresponsive, which is not good.

Asynchronous programming allows tasks to run at the same time, so the main thread can continue to work on other tasks while waiting for the time-consuming one to finish. This is done by using callback functions, Promises, and, more recently, the await and async keywords. These techniques help developers write code that is efficient and responsive, making the most of available resources.

Promises: Managing Asynchronous Operations

In ECMAScript 6 (ES6), the concept of `Promises` was introduced to make asynchronous programming easier and to tackle scenarios where multiple asynchronous tasks require coordination. A Promise is an object that represents a value that may be accessible presently, in the future, or not at all. To illustrate, consider a basic example of a Promise that mimics retrieving data from a server.

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const data = { id: 1, name: 'John Doe' };
    resolve(data); // Resolving the Promise with the fetched data
  }, 2000);
});

fetchData.then(result => {
  console.log(result); // Output: { id: 1, name: 'John Doe' }
});
```


In this example, the `fetchData` `Promise` represents an asynchronous operation that will be resolved with the fetched data after a 2-second delay. The `.then()` method is used to handle the resolved value.

Promises have three states: `pending`, `fulfilled`, and `rejected`. The `resolve` function transitions the Promise to the `fulfilled` state with the provided value, while the `reject` function transitions it to the `rejected` state with an error.

async and await: A Syntactic Sweetener

Although `Promises` offer an improvement over callback hell in asynchronous programming, things can still get pretty complicated when multiple asynchronous operations are involved. This often leads to complex and hard-to-read code, which can be a real headache for developers. However, the introduction of the `async` and `await` keywords in **ES8** have provided a more synchronous-looking syntax for handling asynchronous code. This has helped to simplify the process and make it more intuitive for developers to work with. With these new features, developers can now write cleaner, more readable code that is easier to debug and maintain.

The `async` Keyword

The `async` keyword is a special identifier used in JavaScript to declare an asynchronous function. This type of function always returns a `Promise`, which is a special object used for handling asynchronous operations in JavaScript. The benefit of using the `async` keyword is that it enables you to write **asynchronous code** that appears similar to **synchronous code**, making it easier to follow and maintain. Essentially, an `async` function allows you to avoid callback hell, which is a common issue that arises when working with asynchronous code. By using the `async/await` syntax, you can write asynchronous code in a more concise and readable way.

```
async function fetchDataAsync() {
  return { id: 1, name: 'Jane Smith' };
}

fetchDataAsync().then(result => {
  console.log(result); // Output: { id: 1, name: 'Jane Smith' }
});
```

In this example, `fetchDataAsync` is an asynchronous function that directly returns an object. Even though it looks like synchronous code, calling `fetchDataAsync()` still returns a Promise.

The await Keyword

When dealing with `async` functions, developers utilize the keyword `await` within a function in order to pause its execution until the `Promise` has been resolved. This technique provides a more streamlined and understandable method for managing Promises. To further clarify, let's examine an example implementation of the await keyword in action.

```
async function fetchAndProcessData() {
  const data = await fetchDataAsync();
  console.log(data); // Output: { id: 1, name: 'Jane Smith' }
}
fetchAndProcessData();
```

In this example, `await fetchDataAsync()` pauses the execution of `fetchAndProcessData` until the Promise returned by `fetchDataAsync` is resolved. This makes the code read as if it's performing synchronous operations, even though it's still asynchronous behind the scenes.

- **3 Fetch**
- Used for making HTTP requests.
- **Example:**

```
fetch("https://api.example.com/data")
.then((response) => response.json())
.then((data) => console.log(data));
```


Practical Example: Fetching Data from an API

Let's fetch data from a public API using `fetch()`, which returns a Promise.

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(response => {
        if (!response.ok) throw new Error("Network response was not ok");
        return response.json(); // Parse JSON data
    })
    .then(data => console.log(data)) // Output fetched data
    .catch(error => console.error("Error:", error)); // Handle errors
```


ERROR HANDLING

- There are three types of errors in programming: Syntax Errors, Runtime Errors, and Logical Errors.
- **Syntax Errors**
- Syntax errors, also called parsing errors, occur at compile time in traditional programming languages and at interpret time in JavaScript. When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and the rest of the code in other threads get executed assuming nothing in them depends on the code containing the error

- **Runtime Errors**
- Runtime errors, also called exceptions, occur during execution (after compilation/interpretation). Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.
- **Logical Errors**
- Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result as expected. You cannot catch those errors, because it depends on your business requirement, what type of logic you want to put in your program. JavaScript throws instances of the Error object when runtime errors occur. The following table lists predefined types of the Error object.

Error Object	Description
EvalError	Creates an instance representing an error that occurs regarding the global function **eval()**
RangeError	Creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range
ReferenceError	Creates an instance representing an error that occurs when de-referencing an invalid reference

SyntaxError	Creates an instance representing a syntax error that occurs while parsing the code
TypeError	Creates an instance representing an error that occurs when a variable or parameter is not of a valid type
URIError	Creates an instance representing an error that occurs when **encodeURI()** or **decodeURI()** are passed invalid parameters

- **ThrowingExceptions**
- An error (predefined or user defined) can be raised using the throw statement. Later these exceptions can be captured and you can take an appropriate action. Following is the syntax for the same.

Syntax: Throwing a generic exception

```
throw new Error([message])
OR
throw([message])
```

Syntax: Throwing a specific exception

```
throw new Error_name([message])
```


EXCEPTION HANDLING

- Exception handling is accomplished with a try…catch statement. When the program encounters an exception, the program will terminate in an unfriendly fashion. To safeguard against this unanticipated error, we can wrap our code in a try…catch statement.
- The try block must be followed by either exactly one catch block or one finally block (or one of both). When an exception occurs in the try block, the exception is placed in e and the catch block is executed. The optional finally block executes unconditionally after try/catch.

Following is the syntax for the same.

```
try {
// Code to run
[break;]
} catch ( e ) {
// Code to run if an exception occurs
```

```
[break;]
}[ finally {
// Code that is always executed regardless of
// an exception occurring
}]
```


Example

```
var a = 100;
var b = 0;
try
{
      if (b == 0 )
      {
          throw(“Divide by zero error.”);
      }
      else
      {
          var c = a / b;
      }
}
catch( e )
{
      console.log("Error: " + e );
}
```


Output

The following output is displayed on successful execution of the above code.

```
Error: Divide by zero error.
```

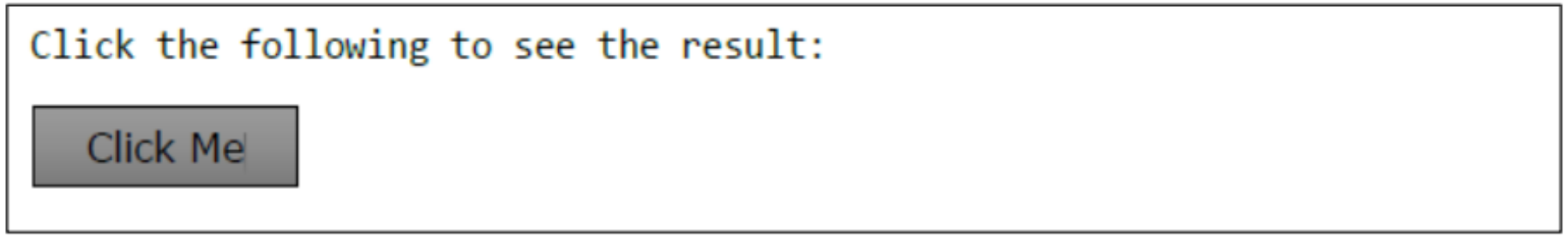

- Theonerror( ) Method
- The onerror event handler was the first feature to facilitate error handling in JavaScript. The error event is fired on the window object whenever an exception occurs on the page.

Example

```
<html>
<head>
<script type="text/javascript">
      window.onerror = function () {
      document.write ("An error occurred.");
      }
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```


Output

The following output is displayed on successful execution of the above code.



- The onerror event handler provides three pieces of information to identify the exact nature of the error:
- Error message: The same message that the browser would display for the given error.
- URL: The file in which the error occurred.
- Line number: The line number in the given URL that caused the error. The following example shows how to extract this information.

Example

```
<html>
<head>
<script type="text/javascript">
window.onerror = function (msg, url, line) {
     document.write ("Message : " + msg );
     document.write ("url : " + url );
     document.write ("Line number : " + line );
}
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```


- Custom Errors
- JavaScript supports the concept of custom errors. The following example explains the same.

Example 1: Custom Error with default message

```
function MyError(message) {
  this.name = 'CustomError';
  this.message = message || 'Error raised with default message';
}
try {
  throw new MyError();
} catch (e) {

console.log(e.name);
console.log(e.message);  // 'Default Message'
}
```


The following output is displayed on successful execution of the above code.

```
CustomError

Error raised with default message
```


Example 2: Custom Error with user-defined error message

```
function MyError(message) {
  this.name = 'CustomError';
  this.message = message || 'Default Error Message';

}

try {
  throw new MyError('Printing Custom Error message');
} catch (e) {
  console.log(e.name);
  console.log(e.message);
}
```

The following output is displayed on successful execution of the above code.

```
CustomError
Printing Custom Error message
```


INTRODUCTION TO JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application as text. It is language-independent and can be used with many programming languages, including JavaScript, Python, Java, and more.
- **JSON Syntax and Structure**
- JSON syntax is a subset of JavaScript object notation syntax. It follows these rules:
- **Data is represented in name/value pairs.**

- **Data is represented in name/value pairs.**

```Json
"name": "value"
```

- **Data is separated by commas.**

```Json
"name": "value",
"age": 30
```


Curly braces `{}` hold objects.

Json

```json
{
  "name": "John Doe",
  "age": 30,
  "city": "New York"
}
```

Square brackets `[]` hold arrays.

Json

```json
{
  "name": "John Doe",
  "age": 30,
  "cars": ["Ford", "BMW", "Fiat"]
}
```


5.5.2 Simple Example

Here's a simple example of a JSON object representing a person:

Json

```json
{
  "name": "Jane Doe",
  "age": 25,
  "email": "janedoe@example.com",
  "isStudent": true,
  "skills": ["JavaScript", "Python", "HTML", "CSS"],
  "address": {
    "street": "123 Main St",
    "city": "Somewhere",
    "state": "CA",
    "postalCode": "12345"
  }
}
```


- **Common Uses of JSON**
- **Data Interchange**:
  - JSON is commonly used to exchange data between a server and a client. It's often used in web applications to send and receive data in a structured format.
  - **Example**: An API might use JSON to send data from a server to a web browser, such as user information or product details.
- **Configuration Files**:
  - JSON is used for configuration files in various programming environments. It's a simple format that allows for easy parsing and modification.
  - **Example**: Application settings, environment configurations, and build scripts may use JSON files to store configuration data.

•**Storing Data**:
•JSON can be used to store data in a structured format. It's often used in databases, particularly NoSQL databases like MongoDB.
•**Example**: A document in a MongoDB collection might be stored in JSON format to represent complex data structures.
•**Serialization and Deserialization**:
•JSON is used for serializing and deserializing data. Serialization converts data structures into a JSON string, while deserialization converts a JSON string back into data structures.
•**Example**: A JavaScript object can be serialized into JSON for storage or transmission and deserialized back into a JavaScript object for use in an application.

•**APIs and Web Services**:
•JSON is widely used in APIs and web services to enable communication between different systems. It provides a standardized way to structure data for APIs.
•**Example**: RESTful APIs often use JSON to format the request and response data, making it easy for clients and servers to understand the exchanged information.

- **Benefits of JSON**
- **Lightweight**: JSON is a minimal and lightweight format, making it efficient for data transfer over the internet.
- **Readable**: JSON is easy to read and write for humans, with a simple syntax that is straightforward to understand.
- **Language-Independent**: JSON is language-agnostic, meaning it can be used with virtually any programming language. Many languages have built-in support for parsing and generating JSON.
- **Structured**: JSON provides a structured way to represent complex data, including nested objects and arrays, which is useful for representing hierarchical data.

- **Introduction to AJAX**
- **AJAX** (Asynchronous JavaScript and XML) is a technique used in web development to create dynamic and interactive web pages. It allows web pages to be updated asynchronously by exchanging data with a web server in the background. This means that parts of a web page can be updated without having to reload the entire page.
- **Key Concepts of AJAX**
- **Asynchronous**:
  - **Description**: AJAX operates asynchronously, meaning that the browser does not have to wait for a server response before continuing to execute other scripts. This allows for a smoother and more responsive user experience.
  - **Example**: Loading new content or submitting form data without refreshing the entire web page.

- **JavaScript and XML**:
- **JavaScript**: AJAX relies on JavaScript to make asynchronous HTTP requests to the server and handle the server's response.
- **XML**: Originally, AJAX used XML to format the data sent and received. However, JSON (JavaScript Object Notation) has become more popular due to its simplicity and ease of use.

HOW AJAX WORKS

- **Create an XMLHttpRequest Object**: This object is used to interact with the server and make HTTP requests.
- **Configure the Request**: Set up the request by specifying the HTTP method (e.g., GET, POST) and the URL to which the request is sent.
- **Send the Request**: Send the HTTP request to the server.
- **Handle the Response**: Define a callback function to handle the server's response. This function updates the web page content based on the data received from the server.
- **Basic AJAX Example**
- Here's a simple example of using AJAX to fetch data from a server and update the web page content without refreshing the page:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>AJAX Example</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
</head>
<body>
    <h1>AJAX Example</h1>
    <button id="loadButton">Load Data</button>
    <div id="content"></div>

    <script>
        $(document).ready(function() {
            $("#loadButton").click(function() {
                $.ajax({
                    url: "https://api.example.com/data", // Replace with your API endpoint
                    method: "GET",
                    success: function(data) {
                        $("#content").html(data);
                    },
                    error: function(xhr, status, error) {
                        console.error("Error:", error);
                    }
                });
            });
        });
    </script>
</body>
```


- **Benefits of Using AJAX**
- **Improved User Experience**: AJAX allows web pages to be more responsive and interactive by updating content without reloading the entire page.
- **Reduced Server Load**: By fetching only the necessary data, AJAX reduces the amount of data transferred between the client and server, leading to faster load times and reduced server load.
- **Enhanced Performance**: AJAX enables faster and more efficient data retrieval and display, improving the overall performance of web applications.