Optimizing JavaScript Functions: Role of Strings

Engage and Think

Imagine you are managing a design team assigned to create a mobile application for a travel company. The app aims to revolutionize how users interact with the company's products and services, incorporating personalized recommendations and intuitive user interfaces. As the design phase progresses, you recognize the complexity of the project and the need for a more creative approach and ideas.

How can generative AI be leveraged to enhance the design process and streamline the development of complex features?

Learning Objectives

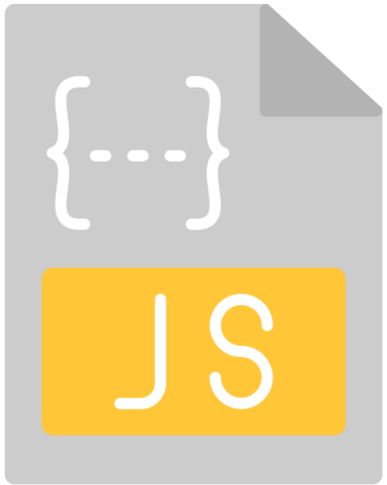
By the end of this lesson, you will be able to:

- Apply string manipulation techniques in JavaScript to format, concatenate, and extract substrings efficiently
- Utilize JavaScript string functions like split(), slice(), replace(), and toUpperCase() to handle text processing
- Apply JavaScript functions to modularize code, improve reusability, and manage function execution with parameters and return values
- Utilize string encoding and decoding techniques to ensure secure data transmission in JavaScript applications

Introduction to Strings

What Are Strings?

They are a primitive data type representing a sequence of characters, including letters, numbers, symbols, and whitespace.



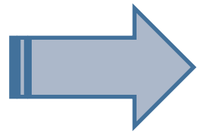
Strings can be created by enclosing a sequence of characters within either single quotes (' '), double quotes (" "), or template literals (` `).

Strings: Example

To create strings in JavaScript, users can use straightforward syntax with either single or double quotes or template literals, as shown below:

Example:

```
// String with single quotes
let singleQuotesString = 'Hello, World!’;

// String with double quotes
let doubleQuotesString = "Greetings,
Universe!";

// String with template literal
let templateLiteralString = `The result
of 2 + 2 is ${2 + 2}.`;
```



Output:

```
Hello, World!
Greetings, Universe!
The result of 2 + 2 is
4.
```

Template literals are useful for constructing strings with variable content. They embed expressions within the string to allow dynamic string creation.

What Are String Objects?

In JavaScript, string objects are instances of the String class created using the new String() constructor.

Example:

```
let strObj = new String('Hello');
```

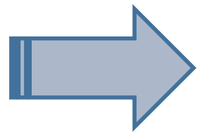
Unlike primitive strings, string objects store text as objects and come with built-in properties and methods.

String Object: Example

In JavaScript, strings are instances of the String object, and the creation of a new instance is achieved using the new keyword as demonstrated below:

Example:

```
// String with single quotes

let stringObject = new String('String
Object');

console.log(stringObject);
```



Output:

```
[String: 'String Object']
```

These strings have properties and methods that can be accessed and used.

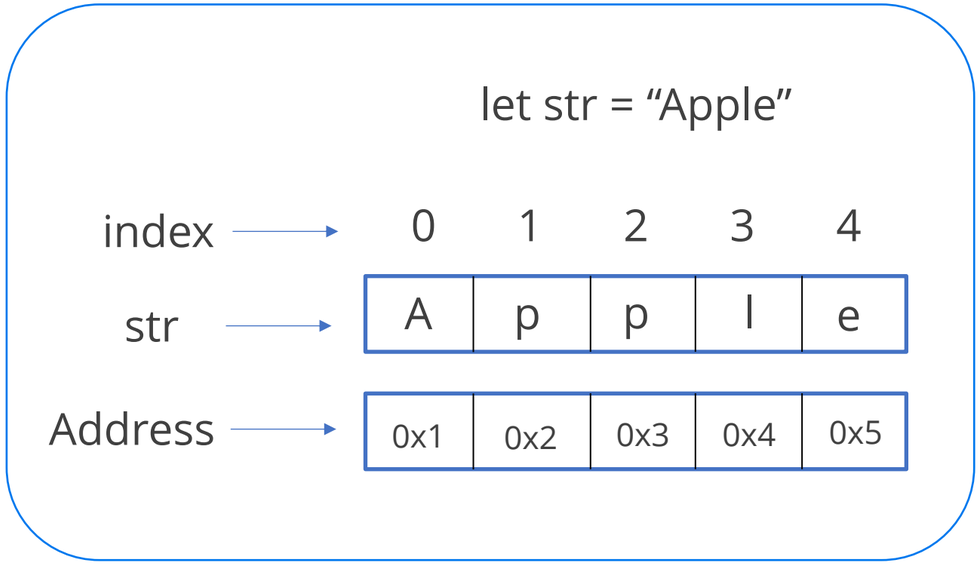
String Object: Limitations

The following limitations are associated with using string objects:

1	Immutability	String objects cannot be changed after creation.
2	Automatic conversion	Automatic conversion between string primitives and objects causes confusion and unexpected behavior.
3	Performance	String objects add overhead, making them unsuitable for performance-critical scenarios.
4	Complexity	String objects may introduce complexity in certain scenarios.

Strings: Indexing

It assigns a unique number to each character in a string, starting from 0, allowing easy access and manipulation. The following diagram shows the indexing of a string:



- Each character is associated with a specific index.
- The first character **‘A'** is at index 0, the second character **‘p'** is at index 1, and so on.
- The address refers to the memory location where individual characters within a string, like **'A'**, **'p'**, **'p'**, **'l’**, and **'e'**, are stored.

Properties of Strings

The string properties for effective data manipulation and type identification are:

1. **length:** Returns the number of characters in a string
2. **constructor**: Returns a reference to the constructor function that created the string object
3. **prototype:** Allows adding properties and methods to an object

Methods in Strings

The string object provides a set of built-in methods for manipulating and retrieving information from string values. These include the following:

Method	Description
charAt()	It provides the char value present at the specified index.
charCode()	It provides the Unicode value of a character present at the specified index.
concat()	It provides the combination of two or more strings.
indexOf()	It provides the position of a char value present in the given string.
lastIndexOf()	It provides the position of a char value present in the given string by searching for a character from the last position.
search()	It searches for a specified regular expression in each string and returns its index position if the match occurs.

Methods in Strings

The string object provides a set of built-in methods for manipulating and retrieving information from string values. These include the following:

Method	Description
match()	It searches for a specified regular expression in each string and returns the regular expression if a match occurs.
replace()	It replaces a given string with the specified replacement.
substr()	It fetches the part of the given string based on the specified starting position and the length.
substring()	It fetches a part of the given string based on the specified index.
slice()	It fetches a part of the given string and returns it as a new string.
toLowerCase()	It changes the uppercase letter in the given string to a lowercase letter.

Methods in Strings

The string object provides a set of built-in methods for manipulating and retrieving information from string values. These include the following:

Method	Description
toLocaleLowerCase()	It converts the characters in the given string into lowercase letters based on the host's current locale.
toUpperCase()	It converts the given string into an uppercase letter.
toLocaleUpperCase()	It converts the characters in the given string into uppercase letters based on the host's current locale.
toString()	It provides a string representing a particular object.
valueOf()	It provides the primitive value of the string object.
split()	It splits a string into a substring array and returns the newly created array.
trim()	It trims the spaces from both sides of a string.

Assisted Practice

Implementing String Methods **Duration: 15 Min.**

Problem statement:

You have been asked to implement JavaScript string methods to create, manipulate, and process strings effectively. This includes understanding different ways to define strings, accessing their properties, and utilizing various string methods for searching, modifying, and extracting content.

Outcome:

By the end of this task, you will be able to create and execute JavaScript programs that utilize string methods for operations like indexing, searching, concatenation, and case conversion. You will also understand how to modify strings using methods such as replace(), split(), and trim().

Note: Refer to the following demo document for detailed steps:
01_Implementing_String_Methods

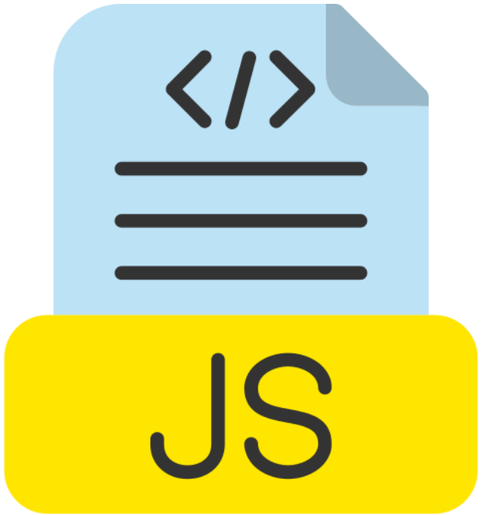
Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

What Is String Interpolation?

It serves as a powerful technique in JavaScript, enabling developers to seamlessly embed expressions or variables within strings.



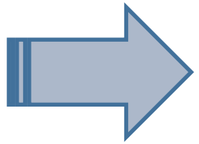
This facilitates the dynamic construction of strings, enhancing code readability and flexibility.

String Interpolation Using Template Literals

Introduced in ES6, template literals offer a cleaner and more readable way to embed expressions in strings compared to traditional concatenation.

Example:

```
// String Interpolation with Template
Literals
let name = "John";
let age = 30;

// Template Literal
let introduction = `Hello, my name is
${name} and I am ${age} years old.`;

// Output
console.log(introduction);
```



Output:

```
Hello, my name is John
and I am 30 years old.
```

Template literals allow direct embedding of expressions, making string manipulation simpler and more readable.

String Interpolation: Benefits

- Improved readability
- Simplified manipulation
- Enhanced multiline support
- Dynamic evaluation
- Reduced syntax errors
- Optimized performance
- Seamless JavaScript integration

Handling Strings Effectively

Beyond interpolation, there is effective string handling in JavaScript which requires managing special characters and ensuring accurate comparisons.

Let us explore how escape characters help represent special symbols in strings and how different comparison methods determine equality.

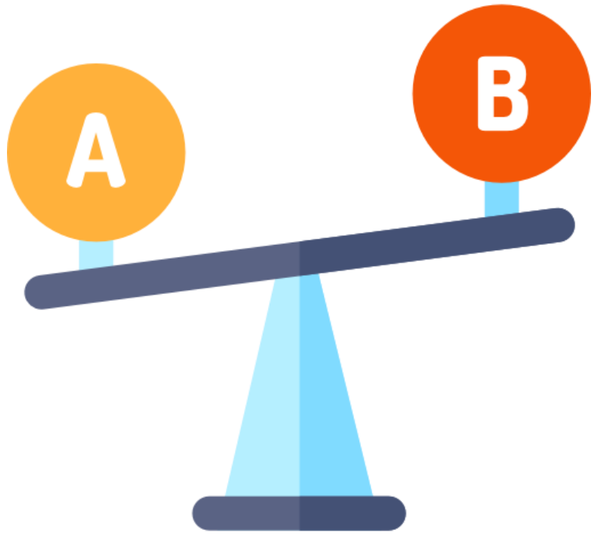
Escape Characters in JavaScript

Strings often include special characters like single quotes, double quotes, and line breaks, which require escape sequences to be represented correctly.

Escape sequences use a backslash (\) followed by a specific character to include special characters in strings, such as quotes or line breaks.

String Comparison

It is commonly used for checking equality and sorting, with methods like equality operators and localeCompare.



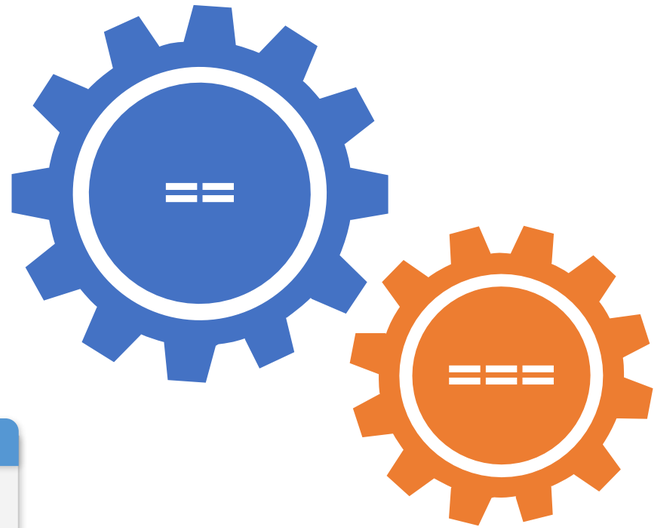
The primary methods for string comparison include using **equality operators** (== and ===) and the **localeCompare** method.

Equality Operators

Loose equality:
It compares strings without considering the data types, attempting conversion if necessary.

Example:

```
console.log(5 == "5");
// true
```



Example:

```
console.log(5 === "5");
// false
```

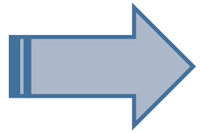
Strict equality:
It compares the value and data type, ensuring that operands are of the same type.

LocaleCompare Method

It provides a detailed way to compare strings, especially for sorting and supporting different languages.

Example:

```
let str1 = "apple";
let str2 = "banana";

let comparisonResult =
str1.localeCompare(str2);

console.log(comparisonResult);
```



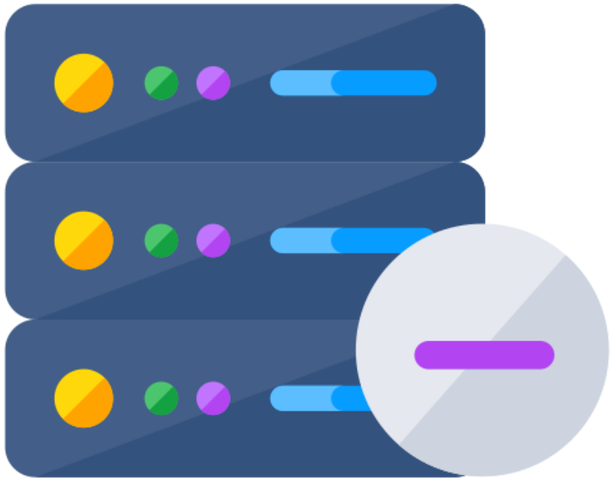
Output:

```
-1
```

In the example above, the localeCompare method returns -1, indicating that "apple" comes before "banana" in the sorting order.

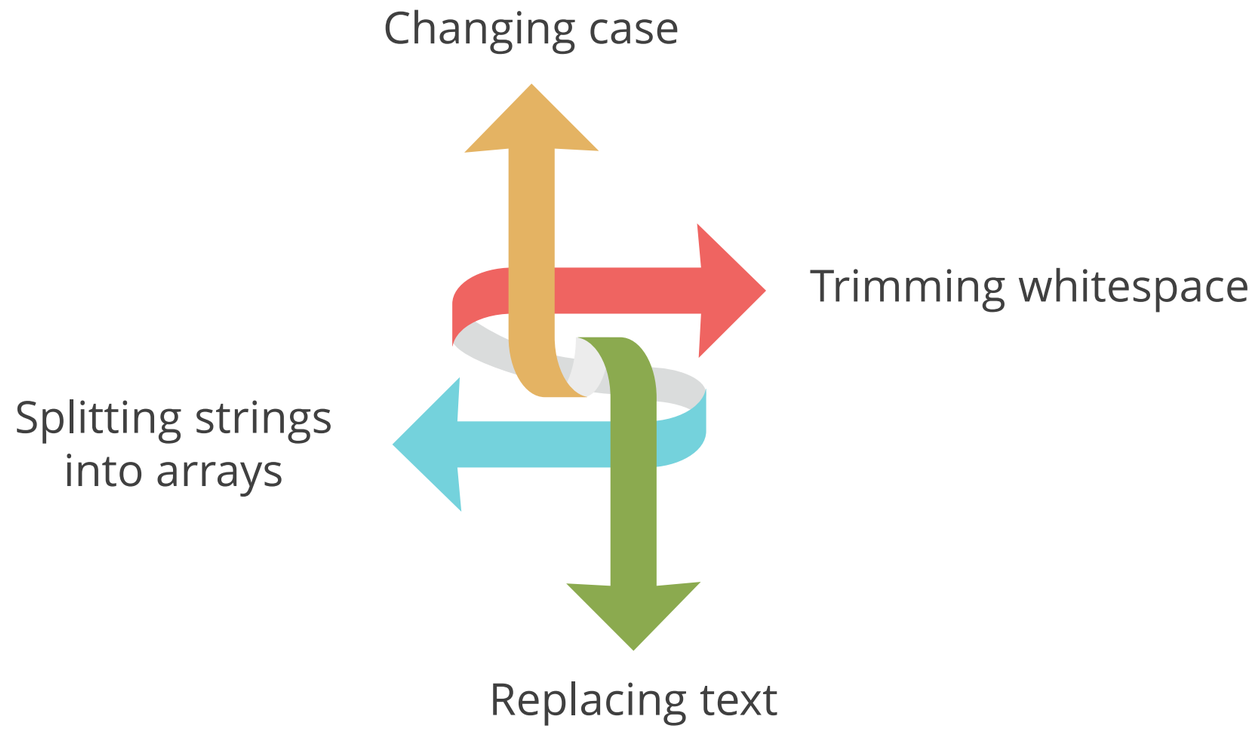
What Is String Manipulation in JavaScript?

It is an essential technique that allows developers to transform, modify, and optimize text data.



Various methods, from changing cases to splitting strings, play a crucial role in enhancing the code's flexibility and functionality.

String Manipulation Techniques



Once strings are modified using these techniques, the next step is iterating over them for further processing, such as data extraction or transformation.

String Iteration

It refers to the process of sequentially accessing each character in a string using loops (for, for...of, while) or built-in methods (split(), map(), forEach()).

String iteration is a fundamental operation in JavaScript programming, enabling efficient traversal, modification, and analysis of text data.

String Iteration: Methods

The following are some common methods for iterating through characters in a JavaScript string:

1. Using the traditional *for* loop
2. Using the charAt() method
3. Constructing a reversed string

A unique Unicode value represents each character processed in a string iteration.
Let us explore how JavaScript supports Unicode and character encoding.

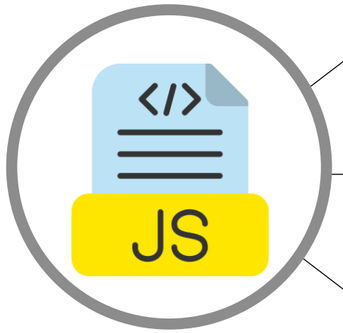
Unicode

It is a universal language for characters, covering everything from letters to symbols.

JavaScript naturally supports Unicode, using unique codes for each character.
This means users can easily include characters from various scripts.

Working With Unicode in JavaScript Strings

The following are the key use cases of Unicode in JavaScript strings:

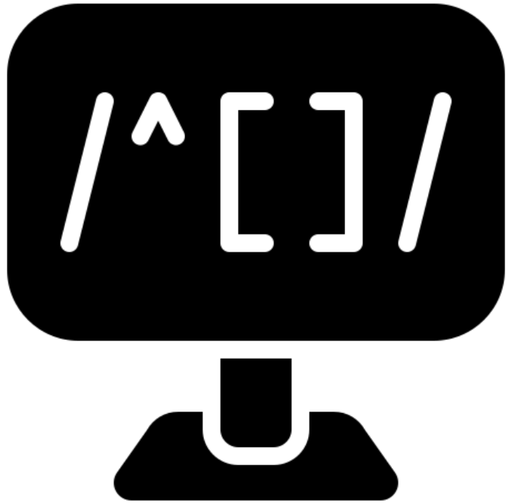


- Retrieve Unicode code points using codePointAt()
- Handle multi-byte characters and surrogate pairs
- Manipulate Unicode strings with normalize(), fromCodePoint(), and localeCompare()

Handling Unicode is essential in JavaScript, but efficiently searching, validating, and modifying text requires pattern matching. This is where regular expressions come in.

What Are Regular Expressions?

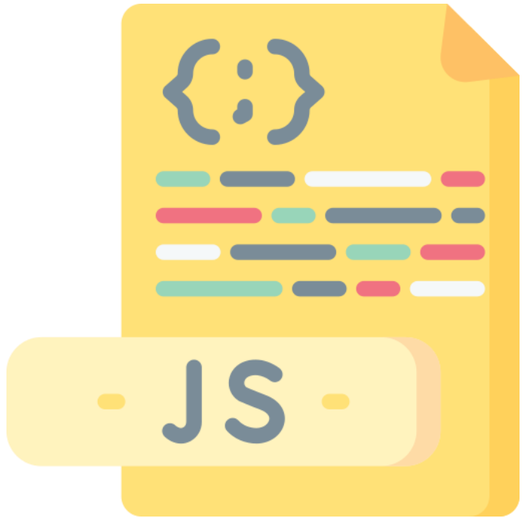
They are a powerful tool in programming, allowing efficient searching, matching, and manipulation of text patterns within strings.



Regular expressions define a pattern for matching strings, enabling advanced text searching, validation, and replacement from single characters to complex structures.

RegExp Object

In JavaScript, the RegExp object provides powerful methods for efficiently defining, searching, and manipulating text patterns.



Common methods like test(), match(), and replace() help validate inputs, extract data, and modify text dynamically.

Core Components of Regular Expressions

01 **Character classes ([a-z], \d):** Define sets of allowed characters

02 **Quantifiers (*, +, {n}):** Specify repetitions of characters or groups

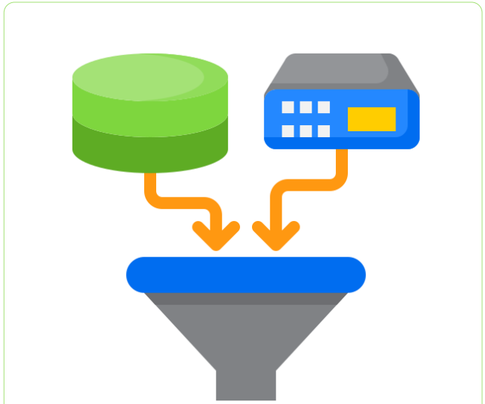
03 **Anchors (^, $):** Define start or end positions in a string

04 **Parentheses (()):** Group expressions and capture matches

Common Use Cases of Regular Expressions

Validation

Ensures input follows a pattern (for example, email and phone numbers)



Data extraction

Finds and retrieves structured data from text (for example, extracting dates)

Text manipulation

Modifies text dynamically (for example, replacing words)

Regular Expressions: Example

The following steps outline how to implement a regular expression in JavaScript to validate email addresses by defining and testing a matching pattern:

1. **Define the pattern:**
   Identify the required email format (for example, username, @ symbol, domain)

2. **Combine the components:**
   Use regular expression syntax to structure the full pattern

3. **Create the RegExp object:**
   Implement the pattern using JavaScript's RegExp constructor or literal notation

4. **Test the regular expression:**
   Validate different email samples to ensure correctness

Assisted Practice

Implementing Advanced String Operations **Duration: 15 Min.**

Problem statement:

You have been asked to implement advanced JavaScript string operations, including string interpolation, escape characters, comparison, and manipulation techniques. This involves iterating over strings, handling Unicode characters, and utilizing regular expressions for pattern matching.

Outcomes:

By the end of this task, you will be able to apply string interpolation, handle escape characters, manipulate strings using various methods, iterate through strings efficiently, and work with Unicode and regular expressions for pattern recognition.

Note: Refer to the following demo document for detailed steps:
02_Implementing_Advanced_String_Operations

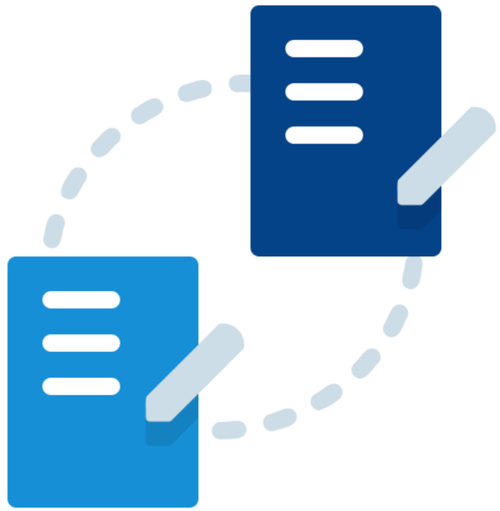
Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute JS file

String Conversions

In programming, handling different data types efficiently is crucial. String conversions allow seamless transformation of data to and from the string format, ensuring compatibility across operations.



String conversions facilitate the transformation of numbers and Booleans into strings and vice versa. This process is essential for manipulating, storing, and retrieving data from various sources.

String Conversions: Data Types to String

The data types that can be transformed into string representations in JavaScript are listed below:

01 Numbers (integers, floats)

02 Booleans (true, false)

03 Characters (single characters from strings)

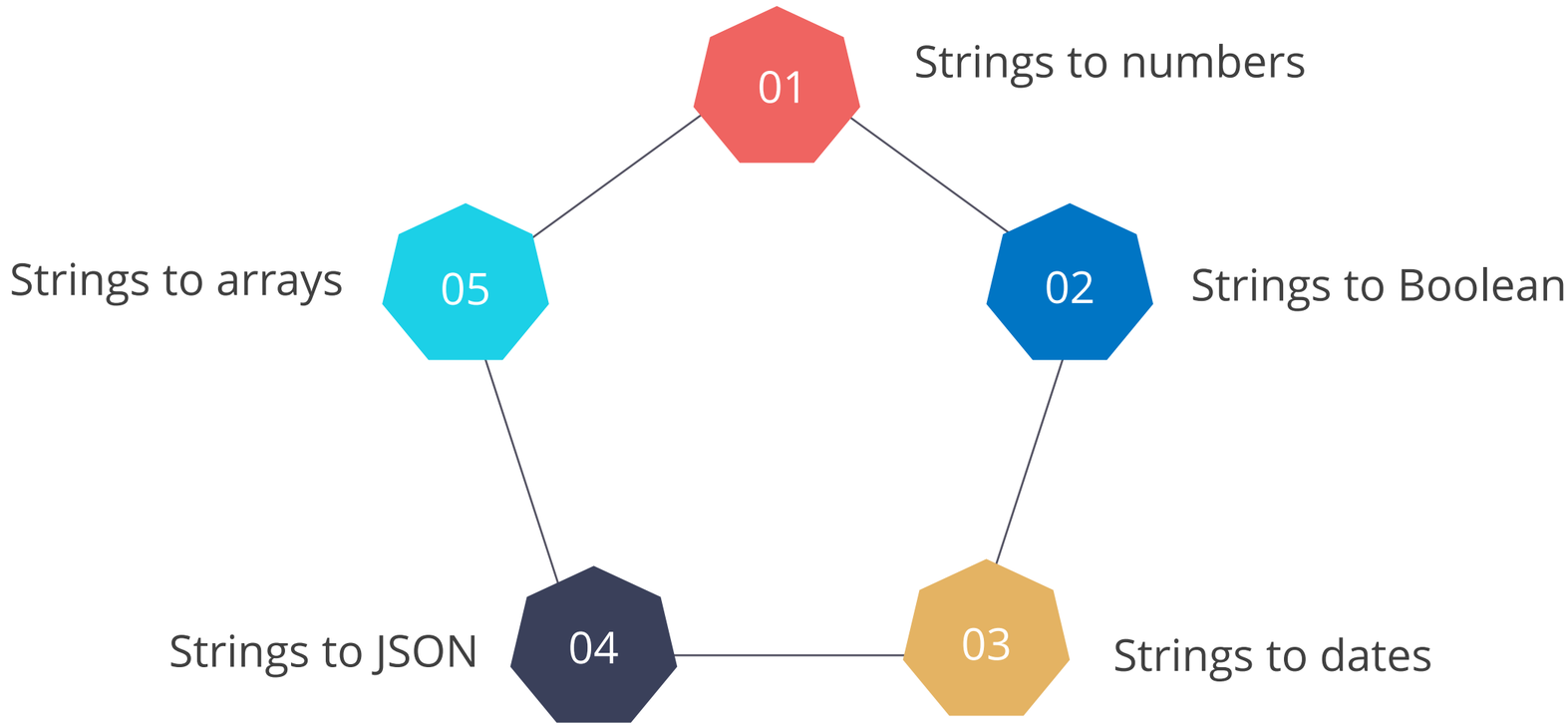
04 Arrays and collections (list of values

05 Objects (key-value pairs)

06 Dates and times (timestamps, date objects)

Parsing Strings to Other Data Types

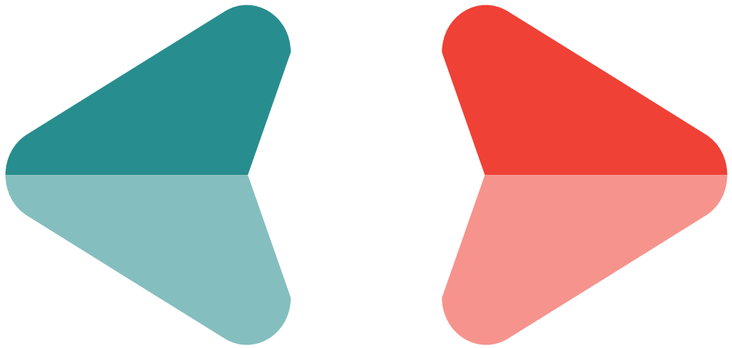
String parsing in JavaScript involves converting string representations of data back into their respective types. The following are common methods for parsing strings into structured data formats:



Common Challenges in String Conversions

Error handling:

Handle errors in string conversions to prevent parsing failures or incorrect data types



Locale considerations:

Account for locale-based number formats (for example, decimal separators) to ensure consistency in international applications

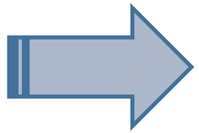
Since strings in JavaScript are immutable, modifying them requires creating new string instances. Let's explore methods for working with immutable strings effectively.

String Immutability

It refers to the unalterable nature of strings; once a string is declared in JavaScript, its characters cannot be changed directly.

Example:

```
// String immutability example

let greeting = "Hello";

greeting[0] = "J";

// This will not modify the string

console.log(greeting);
```



Output:

```
Hello
```

As shown, attempting to modify the first character (‘H’ → ‘J’) does not change the string.
Instead, JavaScript ignores the operation since strings are immutable.

String Immutability: String Manipulation Methods

Although JavaScript strings are immutable, which means modifications create a new string instead of changing the original one, the following methods allow effective string manipulation:

Concatenation: Merging two or more strings using + or concat()

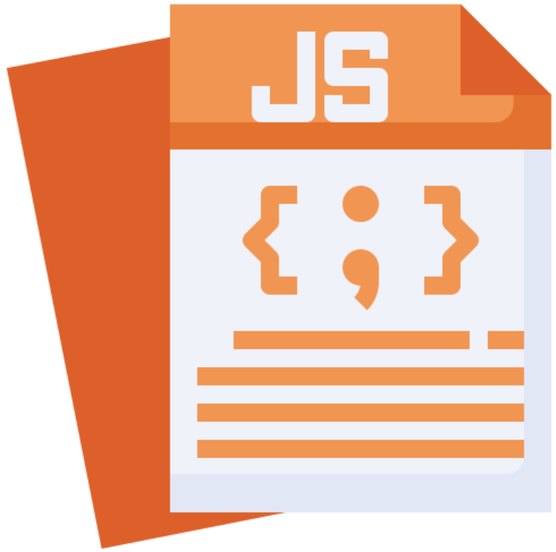
String methods: Using built-in methods like replace(), slice(), or substring()

Template literals: Using backticks (``) for dynamic and multi-line strings

String immutability provides key advantages such as predictable behavior in operations, stable references in memory, and enhanced support for functional programming principles.

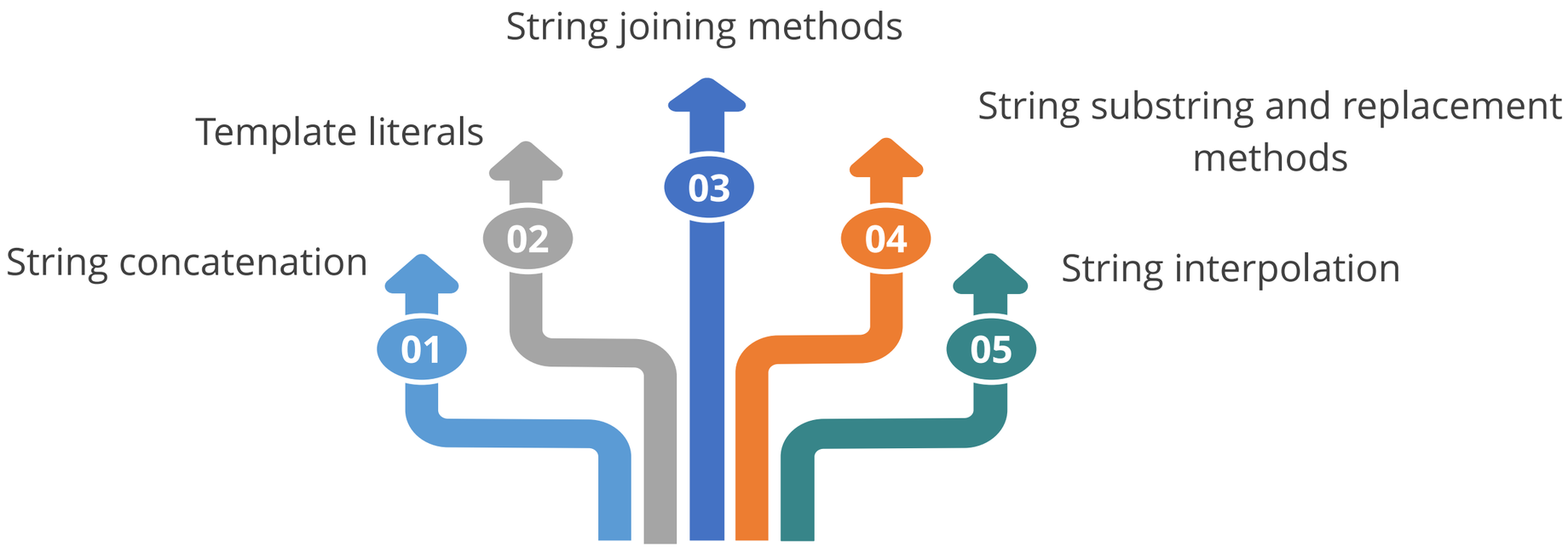
What Is String Formatting?

It involves structuring and presenting text to improve readability and maintain consistency in JavaScript applications.



Understanding and applying proper string formatting techniques for dates, numbers, and custom text ensures clarity and usability in applications.

Methods for String Formatting in JavaScript



These methods provide options for string formatting, allowing developers to choose the best approach for UI presentation, data processing, or other use cases.

String Encoding and Decoding

They are crucial for ensuring secure, efficient, and error-free data transfer in JavaScript applications. The following are the two ways of string encoding and decoding:

URL encoding and decoding:

Converts special characters in a string into a URL-safe format and decodes them back to the original representation

Base64 encoding and decoding:

Encodes binary data into an ASCII string format and restores it to its original binary representation upon decoding

URL Encoding and Decoding

URL encoding converts special characters into a safe format, preventing issues with URL structures to ensure data integrity. In JavaScript, this is commonly achieved using the encodeURIComponent function.

URL encoding

```
let encodedString =
encodeURIComponent(originalStrin
g);

console.log(encodedString);

// Output: "Hello%2C%20World%21"
```

URL decoding

```
let decodedString =
decodeURIComponent(encodedString
);

console.log(decodedString);

// Output: "Hello, World!"
```

To interpret encoded data correctly, URL decoding restores the original string format using the decodeURIComponent function.

Base64 Encoding and Decoding

Base64 encoding transforms binary data into a text-based format using a set of 64 characters, ensuring safe transmission and storage. Decoding reverses this process to restore the original data.

Base64 encoding

```
let base64Encoded =
btoa(binaryData);

console.log(base64Encoded);

// Output:
"VGhpcyBpcyBiaW5hcnkgZGF0YSE="
```

Base64 decoding

```
let decodedBinaryData =
atob(base64Encoded);

console.log(decodedBinaryData);

// Output: "This is binary data."
```

To access the original binary data, Base64 decoding is done using the atob function, which converts the encoded string back to its initial form.

Significance of String Encoding and Decoding

Security in data transmission

Compatibility in URLs

Binary data handling

Data integrity

JavaScript String Handling: Best Practices

Efficient and secure string manipulation is crucial in JavaScript development. Follow these best practices to optimize JavaScript string handling for efficiency and security:

Use template literals for efficient string concatenation

Consider built-in string methods over regular expressions for basic operations

Handle empty strings gracefully to prevent runtime errors

Ensure proper handling of Unicode characters for compatibility

Assisted Practice

Implementing String Manipulation and Optimization **Duration: 15 Min.**

Problem statement:

You have been asked to implement JavaScript string manipulation by converting data types, parsing formats, optimizing operations, handling encoding/decoding, sanitizing inputs, and applying best practices for secure and efficient processing.

Outcomes:

By the end of this task, you will be able to convert and parse strings, manipulate string content using various methods, optimize string operations, handle encoding and decoding, and apply best practices for secure and efficient string handling in JavaScript.

Note: Refer to the following demo document for detailed steps:
03_Implementing_String_Manipulation_and_Optimization

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Quick Check

Imagine you are developing a text-processing feature for a messaging app that needs to ensure user input is properly formatted. One requirement is to remove extra spaces and standardize capitalization before saving messages.

Which JavaScript string method would be most effective for trimming unnecessary spaces from user input?

A. toUpperCase()

B. trim()

C. replaceAll()

D. split()

Functions and Prototyping

What Is a Function?

It is a reusable block of code designed to perform a specific task. It executes only when called or invoked.

Syntax:

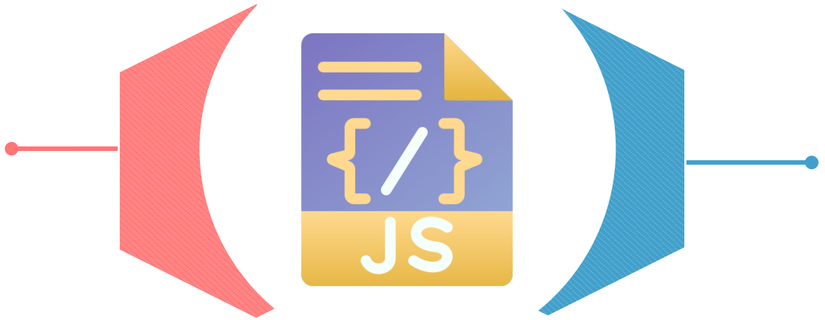
```
function functionName(parameter1
, parameter2, parameter3)
{
    statements 1;
    statements 2;
    statements 3;
return value;
}
```

In JavaScript, a function is declared using the function keyword, followed by a function name, parentheses (), and a block {} containing executable statements.

Function Parameters and Arguments

The properties of a function are:

Function parameters:
Placeholders for values that the function will receive when invoked



Function arguments:
Actual values passed to the function when it is called

Inside the function, parameters (which now hold the values of arguments) behave as local variables.

Benefits of Using Functions

Code reusability

Functions allow code reuse by defining logic once and invoking it multiple times.

Parameterization

Functions accept different arguments, producing varied outputs while using the same logic.

Modularity

Functions break down complex programs into smaller, manageable parts, improving maintainability.

To leverage the advantages of functions, we must first define them properly.
Let us look at how functions are structured and declared in JavaScript.

Function Definition

Also known as a function declaration, it is a way to define reusable code blocks in JavaScript.

Example:

```
function multiply(n1, n2) {
    return n1 * n2;
}

let result = multiply(5, 3);
console.log(result); // Output:
15
```

Every function definition starts with the function keyword, followed by a unique function name.

Function Calling

A function call passes arguments and receives a return value to execute a defined function. Functions can be called directly or using special methods like .call() to change the execution context.

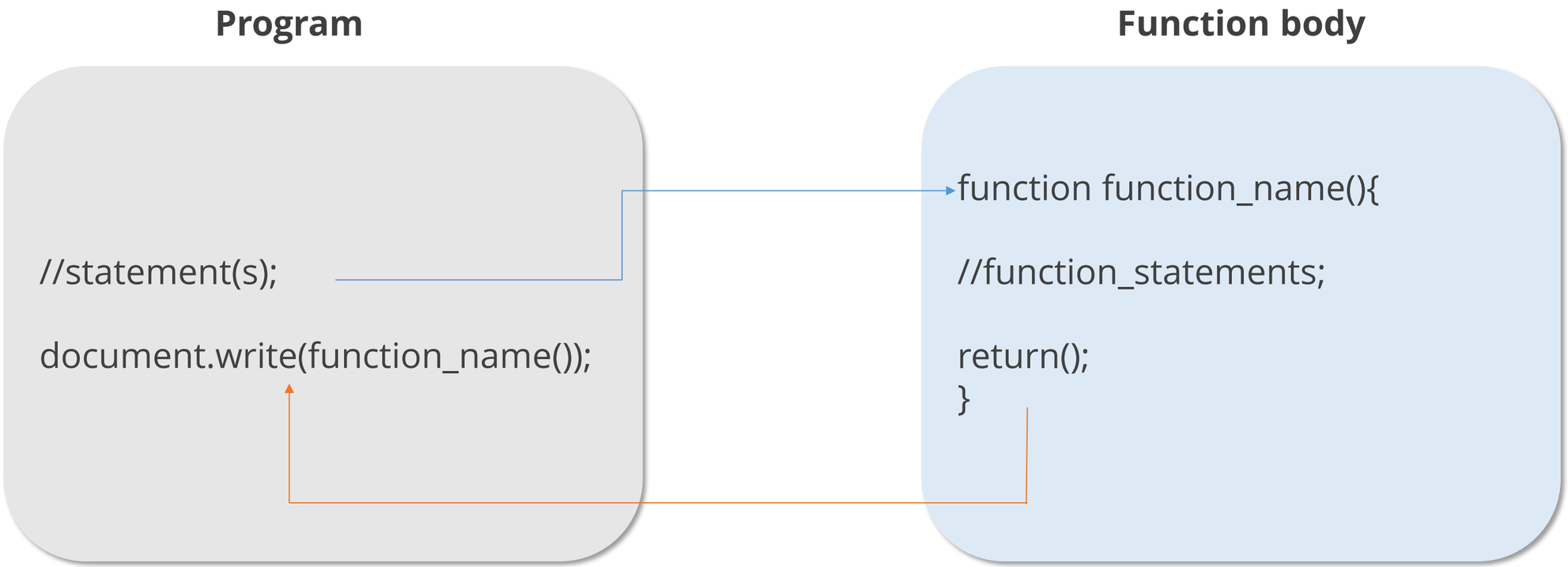
Example:

```
function add(a, b) {
    return a + b;
}

var result = add(14, 16);
console.log(result); // Output: 30
```

The function add(a, b) returns the sum of two numbers. Calling add(14, 16) stores 30 in the result, which is then logged into the console.

Function Execution

It begins with invocation, processes statements in the function body, and returns a result. The following diagram illustrates the flow within a program:



Functions as Objects in JavaScript

JavaScript functions are first-class objects, which means they can:

- Be stored in variables, passed as arguments, and returned from other functions
- Have properties and methods, making them powerful objects

Example:

```
function message() {
    console.log("Greetings
SimpliLearners!");
}

message.language = "JavaScript"; // Adding a
property to a function object

console.log(typeof message); // "function"
console.log(message.language); //
"JavaScript"
```

Passing Functions as Arguments

JavaScript allows functions to be passed as arguments, enabling dynamic execution. This approach enhances code flexibility and promotes modular programming.

This concept leads to callback functions, where one function executes after another, enhancing asynchronous programming in JavaScript.

Passing Functions as Arguments: Example

Using callback functions allows one function to execute another, enabling flexible and dynamic code execution in JavaScript.

Example:

```
function functionOne(x) {
    alert(x); // Displays the value of x in
an alert box
}

function functionTwo(var1, callback) {
    callback(var1); // Calls the passed
function with var1 as an argument
}

functionTwo(2, functionOne); // Passes 2 and
functionOne as arguments
```

The function **functionTwo** takes an argument and a callback function and calls the callback with the argument. Here, **functionOne** is passed as the callback, executing an alert with the value **2** when invoked.

Function Returning Function

The return statement passes information from inside a function back to the point in the main program where the function was called.

Example:

```
function sqr() {
   return function cal(x) {
      return x * x;
   };
}
var ans = sqr();
console.log(ans(5)); // Output: 25
```

Returning a function is useful in a prototype-based object model, allowing a sub-function to be returned to the main function, as shown in the example.

Function Constructor

The Function constructor creates a new function dynamically from a string, but it does not have access to the surrounding lexical scope.

Example:

```
const add = new Function('a', 'b', 'return a + b');
console.log(add(10, 6)); // Output: 16
```

Functions created using the Function constructor lack access to closures from their surrounding lexical scope and should be used cautiously due to security and performance concerns.

Function Constructors: Example

Example:

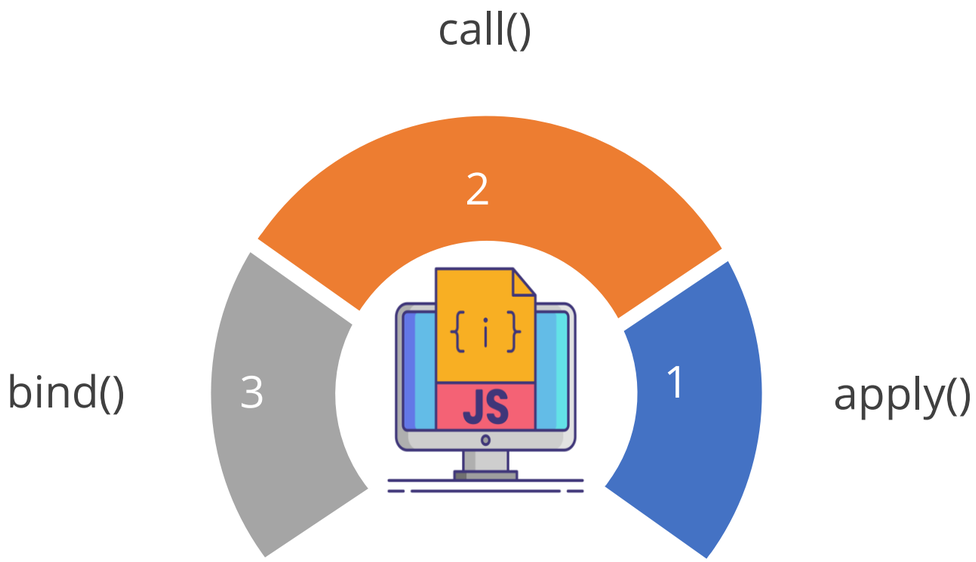
```
const person = {
  firstName: "Peter",
  lastName : "Parker",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " +
this.lastName;
  }};
```

- The *this* keyword refers to the object that is currently executing the code.
- The global object is referred to as *this* if the function is not a property of any object.
- The value of *this* can be changed using the call(), apply() and bind() methods, or arrow functions (es6).

To modify the execution context of a function, JavaScript provides methods like **bind()**, **call()**, and **apply()**, which allow us to control the value of 'this' explicitly.

Function Context Manipulation in JavaScript

The following are the methods to modify function execution context:



Each method offers a unique way to set the execution context of a function, enabling better control over how and where functions are invoked.

Bind()

This method allows you to create a new function with a fixed *this* value, ensuring consistent execution context across different calls.

Example:

```
let bike = {
    data:[
       {name:"Ducati", year:2021},
       {name:"Harley-Davidson", year:2021}
   ]
}
bike.showData = user.showData.bind (bike);
bike.showData ();
```

Call()

JavaScript's call() method enables explicit control over function execution by specifying the context of *this*, making it useful for borrowing methods and modifying scope dynamically.

Example:

```
function Elements () {
    var args = Array.prototype.slice.call (arguments);
    console.log (args);
 }

 Elements('Water', 'fire', 'wind', 'Earth');
```

Apply()

The apply() method works similarly to call(), but it takes an array of arguments instead of a list of arguments.

Example:

```
function Food(type){
    this.type = type;
}
function setFood(Variety){
    Food.apply(this, ["Vegetarian", "vegan", "seafood"]);
    this.Variety = Variety;
}
```

What Is a Prototype?

In JavaScript, objects can share methods and properties efficiently through prototypes.
Let’s explore how prototypes work and why they are fundamental to JavaScript’s inheritance model.

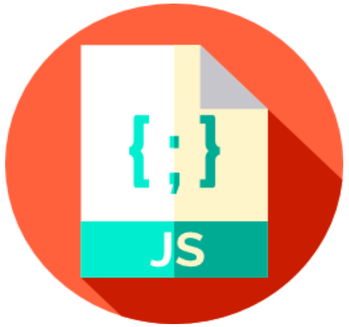
Example:

```
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};

const user = new Person("Alice");
user.greet(); // Output: Hello, my name is Alice
```

The code above defines the constructor function *Person(name)* and adds a greet method to its prototype, allowing all instances to share the method.

Prototype Mechanism in JavaScript

Prototypes in JavaScript provide a mechanism to define shared properties and methods for objects. Here are key aspects of how the prototype mechanism works and its significance in object inheritance:



01 All JavaScript objects inherit properties and methods from a prototype.

02 All function prototype properties are modifiable, but object prototype properties are not visible.

03 All functions include a prototype object by default.

Prototype Chaining in JavaScript

It enables objects to inherit properties and methods from other objects, forming a hierarchical structure that allows efficient method reuse.

Example:

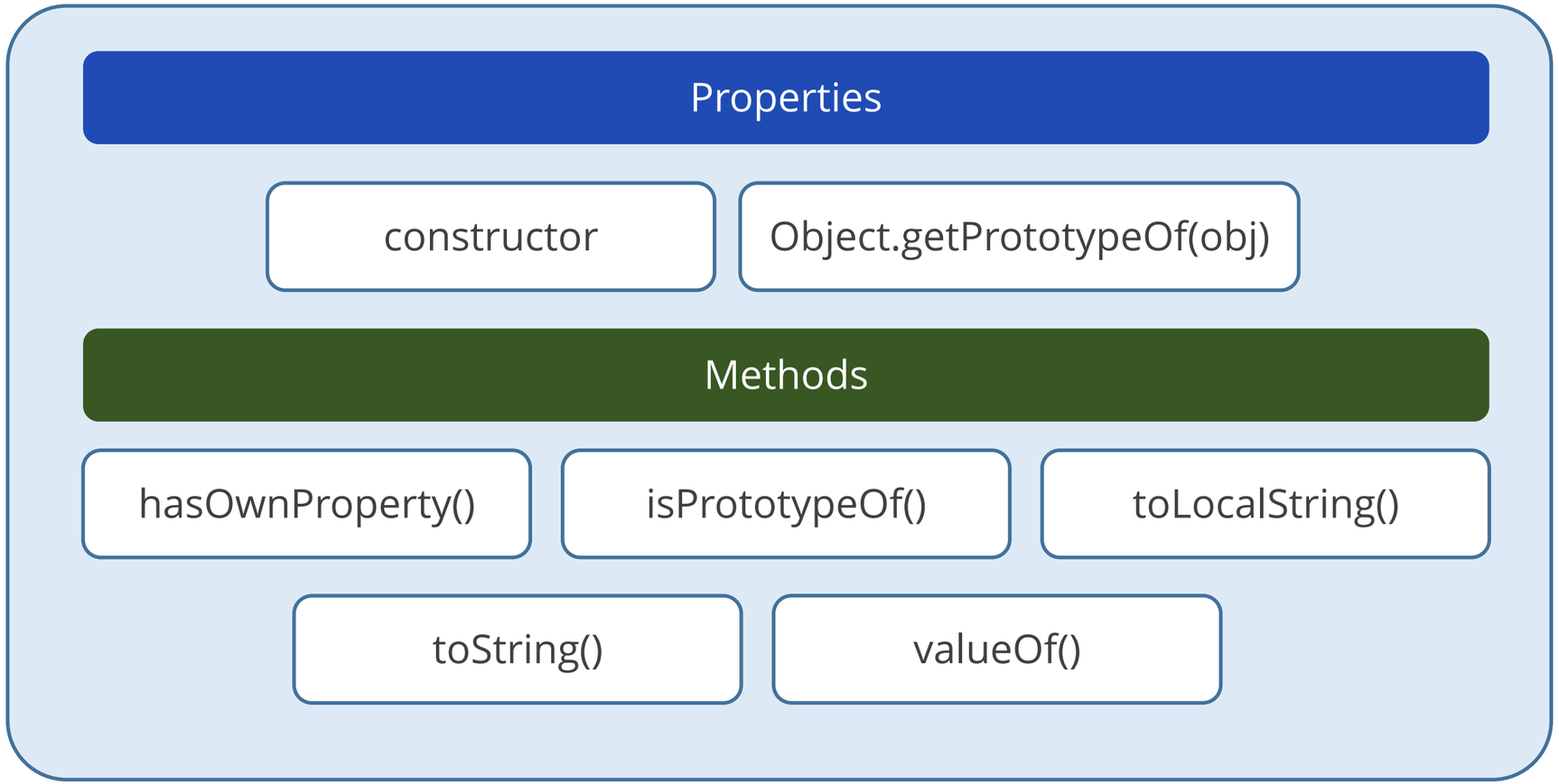
```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function () {
  console.log(`${this.name} makes a
sound.`);
};

const dog = new Animal("Buddy");
dog.speak(); // Output: Buddy makes a
sound.
```

- Animal.prototype holds the speak method, shared by all instances.
- dog.speak() looks for speak in dog, then in Animal.prototype.
- This shows prototype chaining, enabling method inheritance.

By leveraging prototype chaining, JavaScript optimizes memory usage and enables seamless method inheritance, ensuring a more efficient and structured object-oriented programming approach.

Prototype: Properties and Methods

JavaScript prototypes offers the following built-in properties and methods for object inheritance and dynamic property management:



Assisted Practice

Working with Functions **Duration: 15 Min.**

Problem statement:

You have been asked to implement JavaScript functions by creating an HTML file (index.html) and a JavaScript file (function.js). The task involves defining and executing functions for mathematical operations and temperature conversion, ensuring proper function calls and result validation in the browser console.

Outcome:

By the end of this task, you will be able to create and execute JavaScript functions, perform calculations using function parameters, and display results in the browser console for debugging and validation.

Note: Refer to the following demo document for detailed steps:
04_Working_with_Functions

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the index.html file

Assisted Practice

Implementing Functions and Prototyping **Duration: 15 Min.**

Problem statement:

You have been asked to implement JavaScript functions and prototypes by creating an HTML file (index.html) and a JavaScript file (functions_and_prototypes.js). This task involves defining a function constructor, adding a method to the prototype, and creating objects to demonstrate object-oriented programming concepts.

Outcome:

By the end of this task, you will be able to create JavaScript objects using function constructors, extend functionality with prototypes, and validate object properties and methods through the browser console.

Note: Refer to the following demo document for detailed steps:
05_Implementing_Functions_and_Prototyping

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the index.html file

Quick Check

You are developing a user profile system where a method inside an object prints a user's full name. However, when passing this method as a callback to another function, this keyword loses its reference to the original object. Which method should you use to ensure the function retains its original *this* context?

A. call()

B. apply()

C. bind()

D. toString()

Key Takeaways

- String formatting is essential for manipulating and arranging text to improve readability and presentation in programming languages like JavaScript.
- Function calls are crucial for understanding how expressions transfer control and arguments to functions, enabling modular and reusable code.
- String methods like replace(), split(), trim(), toUpperCase() enhance manipulation and formatting for practical tasks.
- Comparison methods such as equality operators and localeCompare() support efficient text evaluation and sorting.
- Regular expressions enable advanced pattern matching and text validation, commonly used in input checks and data extraction.

Programming and Managing the Behavior of Web Pages Dynamically

Project Agenda: To program and manage the behavior of web pages dynamically

Description: This assignment is designed to help understand working with JavaScript to develop a script that can validate the HTML content in web pages. Further, implement the script code in a separate .js file and include it in the HTML web page to work.

LESSON-END PROJECT

Programming and Managing the Behavior of Web Pages Dynamically

Steps to be performed:

1. Open VS Code and create a new folder
2. Create the initial HTML, CSS, and JS files
3. Create the meeting schedule files
4. Run and test the project

Expected deliverables: A functional JavaScript validation implementation that validates name and email on the client creation page and validates the meeting date on the schedule meeting page, ensuring correctness of user inputs with appropriate alerts.

LESSON-END PROJECT

Thank You